Vivekananda
College of Engineering & Technology

Nehru nagar post, Puttur, D.K. 5742013

Lecture Notes on

DAA

15CS43

Design and Analysis of Algorithms

(CBCS Scheme)

Prepared by

Mr. Harivinod N
Dept. of Computer Science and Engineering,
VCET Puttur

Jan 2017

Module-1 : Introduction to Algorithms

Contents

Course website: www.techjourney.in

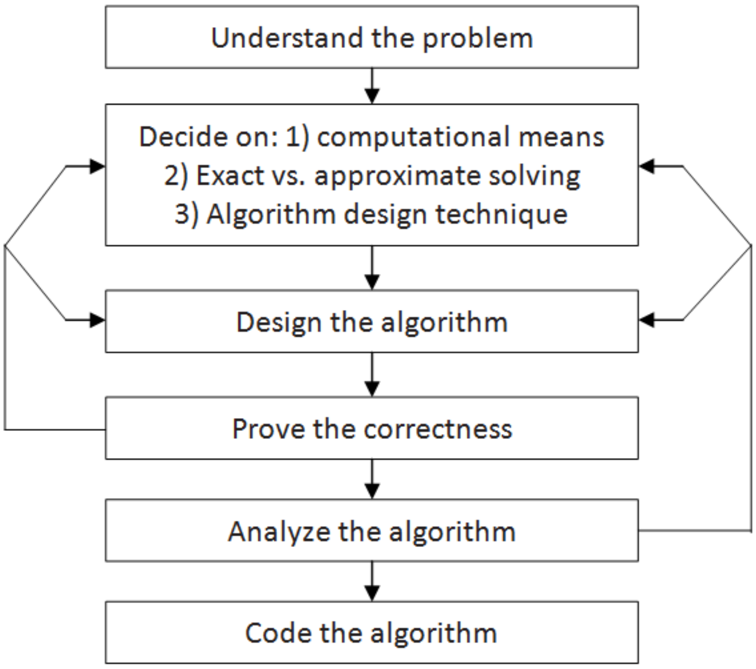
1. Introduction

1.1. What is an Algorithm?

An ***algorithm*** is a finite set of instructions to solve a particular problem. In addition, all algorithms must satisfy the following criteria:

a. ***Input***. Zero or more quantities are externally supplied.
b. ***Output***. At least one quantity is produced.
c. ***Definiteness***. Each instruction is clear and unambiguous. It must be perfectly clear what should be done.
d. ***Finiteness***. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
e. ***Effectiveness***. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion c; it also must be feasible.

Algorithm design and analysis process - We now briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm



- **Understanding the Problem -** From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.

- **Ascertaining the Capabilities of the Computational Device -** Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. Select appropriate model from sequential or parallel programming model.

- **Choosing between Exact and Approximate Problem Solving** - The next principal decision is to choose between solving the problem exactly and solving it approximately. Because, there are important problems that simply cannot be solved exactly for most of their instances and some of the available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.

- **Algorithm Design Techniques** - An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. They provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm.

- **Designing an Algorithm and Data Structures** - One should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes would run longer if we used a linked list instead of an array in its implementation. ***Algorithms + Data Structures = Programs***

- **Methods of Specifying an Algorithm-** Once you have designed an algorithm; you need to specify it in some fashion. These are the two options that are most widely used nowadays for specifying algorithms. Using a ***natural language*** has an obvious appeal; however, the inherent ambiguity of any natural language makes a concise and clear description of algorithms surprisingly difficult. ***Pseudocode*** is a mixture of a natural language and programming language like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.

- **Proving an Algorithm's Correctness -** Once an algorithm has been specified, you have to prove its correctness. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

- **Analyzing an Algorithm -** After correctness, by far the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses. Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder.

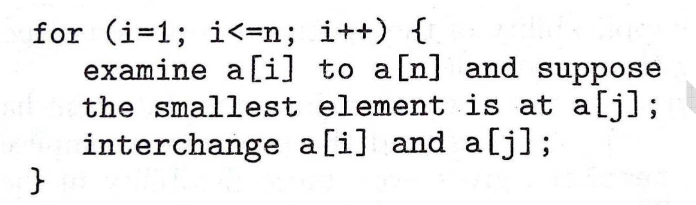
- **Coding an Algorithm -** Most algorithms are destined to be ultimately implemented as computer programs. Implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm's power by an inefficient implementation. Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode.

1.2. Algorithm Specification

An algorithm can be specified in

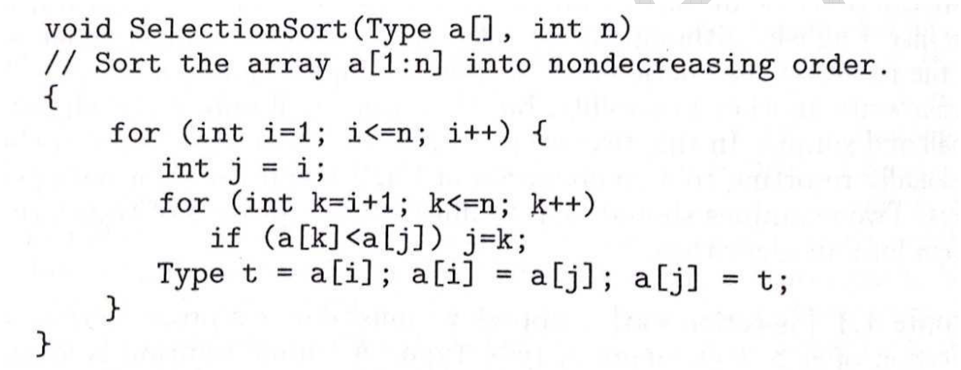
1) Simple English
2) Graphical representation like flow chart
3) Programming language like c++ / java
4) Combination of above methods.

Using the combination of simple English and C++, the algorithm for **selection sort** is specified as follows.



```
for (i=1; i<=n; i++) {
   examine a[i] to a[n] and suppose
   the smallest element is at a[j];
   interchange a[i] and a[j];
}
```

In C++ the same algorithm can be specified as follows



```
void SelectionSort(Type a[], int n)
// Sort the array a[1:n] into nondecreasing order.
{
    for (int i=1; i<=n; i++) {
       int j = i;
       for (int k=i+1; k<=n; k++)
          if (a[k]<a[j]) j=k;
       Type t = a[i]; a[i] = a[j]; a[j] = t;
    }
}
```

Here *Type* is a basic or user defined data type.

Recursive algorithms

An algorithm is said to be **recursive** if the same algorithm is invoked in the body (direct recursive). Algorithm ***A*** is said to be **indirect recursive** if it calls another algorithm which in turn calls A.

Example 1: Factorial computation   n! = n * (n-1)!

Example 2: Binomial coefficient computation

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}$$

Example 3: Tower of Hanoi problem
Example 4: Permutation Generator

1.3. Analysis Framework

General framework for analyzing the efficiency of algorithms is discussed here. There are two kinds of efficiency: **time efficiency** and **space efficiency**. Time efficiency indicates how fast an algorithm in question runs; space efficiency deals with the extra space the algorithm requires.

In the early days of electronic computing, both resources **time** and **space** were at a premium. Now the amount of extra space required by an algorithm is typically not of as much concern, In addition, the research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space. Therefore, following a well-established tradition of algorithm textbooks, we primarily concentrate on time efficiency.

Measuring an Input's Size

It is observed that almost all algorithms **run longer on larger inputs.** For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter ***n*** indicating the **algorithm's input size**.

There are situations, where the choice of a **parameter indicating an input size does matter**. The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

We should make a special note about measuring the size of inputs for algorithms involving **properties of numbers** (e.g., checking whether a given integer n is prime). For such algorithms, computer scientists prefer measuring size by the number ***b*** of bits in the ***n***'s binary representation: $b = \lfloor \log_2 n \rfloor + 1$. This metric usually gives a better idea about the efficiency of algorithms in question.

Units for Measuring Running lime

To measure an algorithm's efficiency, we would like to have a **metric that does not depend on these extraneous factors**. One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the **basic operation**, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

For example, most **sorting** algorithms work by **comparing elements** (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison.

As another example, algorithms for **matrix multiplication** and **polynomial evaluation** require two arithmetic operations: **multiplication and addition**.

Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let ***C(n)*** be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time ***T(n)*** of a program implementing this algorithm on that computer by the formula:

$$T(n) \approx c_{op}C(n)$$

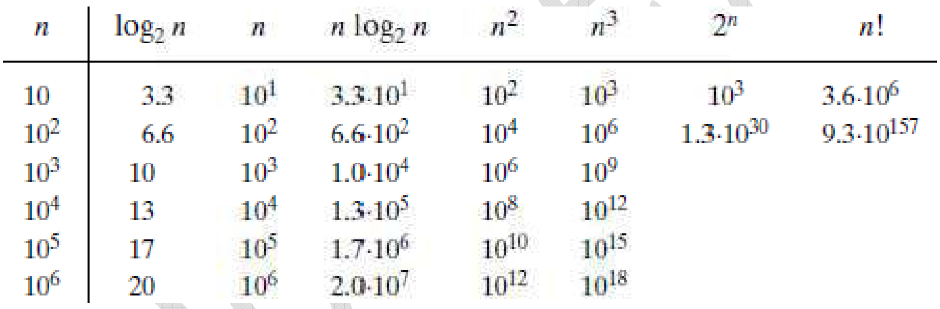
unless ***n*** is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time.

It is for these reasons that the efficiency analysis framework ignores multiplicative constants and concentrates on the count's **order of growth** to within a constant multiple for large-size inputs.

Orders of Growth

Why this emphasis on the count's order of growth for large input sizes? Because for large values of n, it is the function's order of growth that counts: just look at table which contains values of a few functions particularly important for analysis of algorithms.

Table: *Values of several functions important for analysis of algorithms*



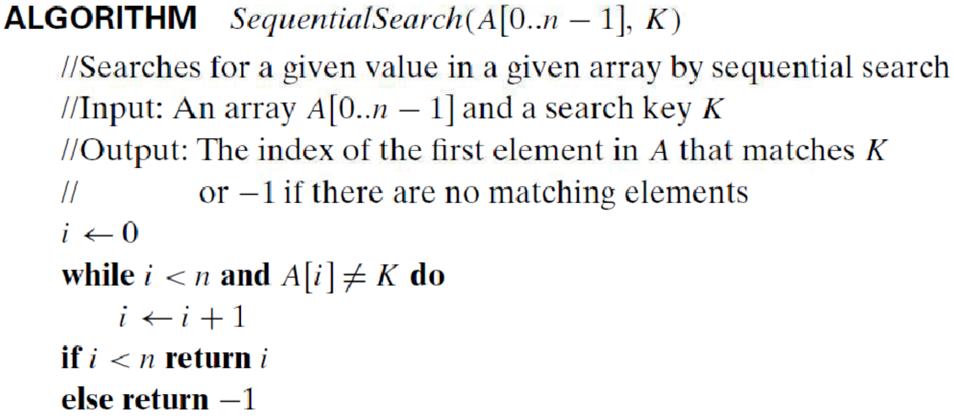
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

Worst-Case, Best-Case, and Average-Case Efficiencies

Definition: The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size n, for which the algorithm runs the longest among all possible inputs of that size.

Consider the algorithm for sequential search.



```
ALGORITHM  SequentialSearch(A[0..n − 1], K)
    //Searches for a given value in a given array by sequential search
    //Input: An array A[0..n − 1] and a search key K
    //Output: The index of the first element in A that matches K
    //         or −1 if there are no matching elements
    i ← 0
    while i < n and A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```

The running time of above algorithm can be quite different for the same list size n. In the worst case, when there are **no matching elements** or the first **matching element happens to be the last one on the list**, the algorithm makes the largest number of key comparisons among all possible inputs of size n: $\boldsymbol{C_{worst}(n) = n}$**.**

In general, we analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $\boldsymbol{C(n)}$ among all possible inputs of size *n* and then compute this worst-case value $\boldsymbol{C_{worst}(n)}$. The worst-case analysis provides algorithm's efficiency by bounding its running time from above. Thus it guarantees that for any instance of size ***n***, the running time will not exceed $\boldsymbol{C_{worst}(n)}$**,** its running time on the worst-case inputs.

Definition: The **best-case efficiency** of an algorithm is its efficiency for the best-case input of size n, for which the algorithm runs the fastest among all possible inputs of that size.

We determine the kind of inputs for which the count $\boldsymbol{C(n)}$ will be the smallest among all possible inputs of size ***n***. For example, for sequential search, best-case inputs are lists of size ***n*** with their first elements equal to a search key; $\mathbf{C_{best}(n) = 1}$**.**

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. Also, neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input. This information is provided by **average-case** efficiency.

Definition: the **average-case complexity** of an algorithm is the amount of time used by the algorithm, averaged over all possible inputs.

Let us consider again sequential search. The standard assumptions are that (a) the probability of a successful search is equal top ($0 \le p \le 1$) and (b) the probability of the first match occurring in the i^{th} position of the list is the same for every i. We can find the average number of key comparisons $\boldsymbol{C_{avg}(n)}$ as follows.

In the case of a successful search, the probability of the first match occurring in the i^{th} position of the list is **p/n** for every ***i***, and the number of comparisons made by the algorithm in such a situation is obviously *i*. In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being ***(1- p).*** Therefore,

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

Investigation of the average-case efficiency is considerably more difficult than investigation of the worst-case and best-case efficiencies. But there are many important algorithms for which the average case efficiency is much better than the overly pessimistic worst-case efficiency would lead us to believe. Note that average-case efficiency cannot be obtained by taking the average of the worst-case and the best-case efficiencies.

Summary of analysis framework

- Both time and space efficiencies are measured as functions of the algorithm's input size.
- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework's primary interest lies in the order of growth of the algorithm's running time (or extra memory units consumed) as its input size goes to infinity.

2. Performance Analysis

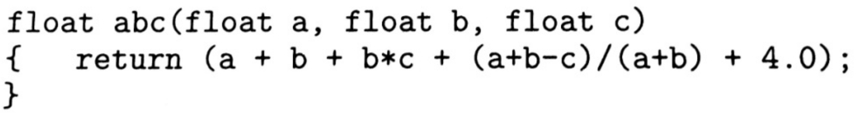
2.1 Space complexity

Total amount of computer memory required by an algorithm to complete its execution is called as **space complexity** of that algorithm. The Space required by an algorithm is the sum of following components

- A **fixed** part that is independent of the input and output. This includes memory space for codes, variables, constants and so on.
- A **variable** part that depends on the input, output and recursion stack. (We call these parameters as instance characteristics)

Space requirement *S(P)* of an algorithm *P*, ***S(P) = c + Sp*** where ***c*** is a constant depends on the fixed part, ***Sp*** is the instance characteristics

Example-1: Consider following algorithm **abc()**

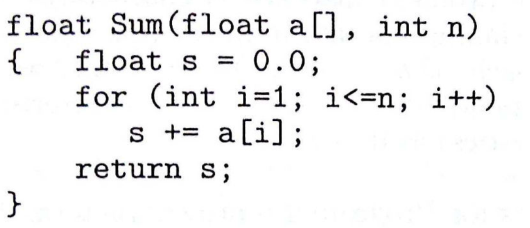


```
float abc(float a, float b, float c)
{   return (a + b + b*c + (a+b-c)/(a+b) + 4.0);
}
```

Here fixed component depends on the size of a, b and c. Also instance characteristics *Sp=0*

Example-2: Let us consider the algorithm to find sum of array.

For the algorithm given here the problem instances are characterized by ***n***, the number of elements to be summed. The space needed by ***a[ ]*** depends on ***n***. So the space complexity can be written as; $S_{sum}(n) \geq (n+3)$ n for a[ ], One each for n, i and s.



```
float Sum(float a[], int n)
{   float s = 0.0;
    for (int i=1; i<=n; i++)
       s += a[i];
    return s;
}
```

2.2 Time complexity

Usually, the execution time or run-time of the program is refereed as its time complexity denoted by t_p (instance characteristics). This is the sum of the time taken to execute all instructions in the program.

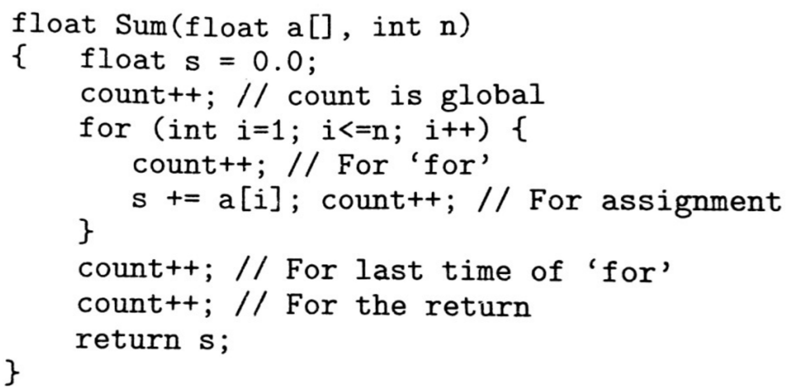
Exact estimation runtime is a complex task, as the number of instruction executed is dependent on the input data. Also different instructions will take different time to execute. So for the estimation of the time complexity **we count only the number of program steps**.

A program step is loosely defined as syntactically or semantically meaning segment of the program that has and execution time that is independent of instance characteristics. For example comment has zero steps; assignment statement has one step and so on.

We can determine the **steps needed by a program** to solve a particular problem instance in two ways.

In the **first method** we introduce a new variable ***count*** to the program which is initialized to zero. We also introduce statements to increment ***count*** by an appropriate amount into the program. So when each time original program executes, the ***count*** also incremented by the step count.

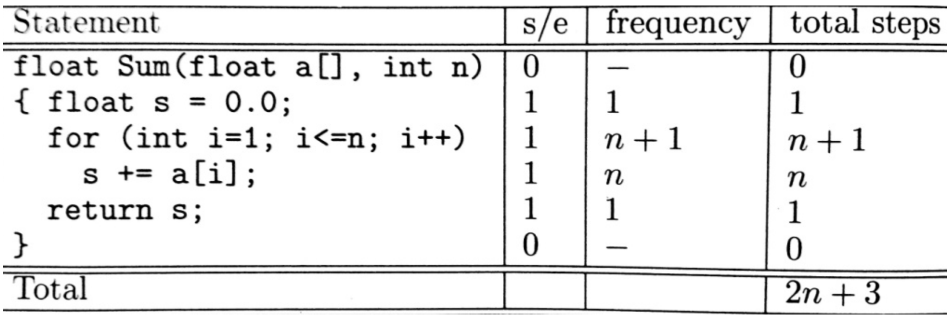
Example-1: Consider the algorithm ***sum( )***. After the introduction of the count the program will be as follows.



```
float Sum(float a[], int n)
{   float s = 0.0;
    count++; // count is global
    for (int i=1; i<=n; i++) {
       count++; // For 'for'
       s += a[i]; count++; // For assignment
    }
    count++; // For last time of 'for'
    count++; // For the return
    return s;
}
```

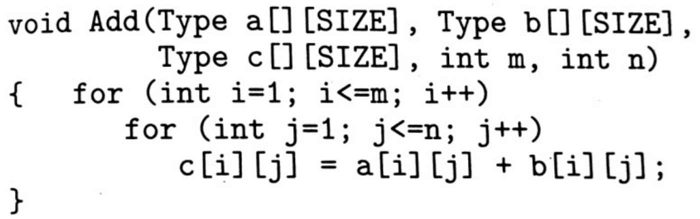
From the above we can estimate that invocation of ***sum( )*** executes total number of **2n+3** steps.

The **second method** to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. An example is shown below.

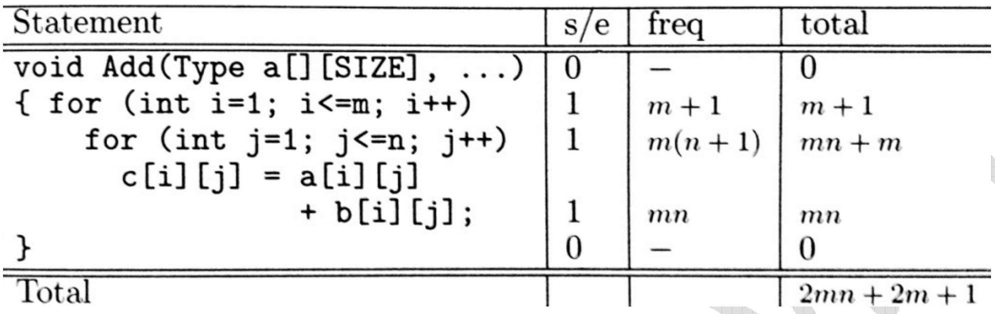


Statement	s/e	frequency	total steps
`float Sum(float a[], int n)`	0	–	0
`{ float s = 0.0;`	1	1	1
`  for (int i=1; i<=n; i++)`	1	$n+1$	$n+1$
`    s += a[i];`	1	n	n
`  return s;`	1	1	1
`}`	0	–	0
Total			$2n+3$

Example-2: matrix addition



```
void Add(Type a[][SIZE], Type b[][SIZE],
         Type c[][SIZE], int m, int n)
{   for (int i=1; i<=m; i++)
        for (int j=1; j<=n; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```



Statement	s/e	freq	total
`void Add(Type a[][SIZE], ...)`	0	–	0
`{ for (int i=1; i<=m; i++)`	1	$m+1$	$m+1$
`for (int j=1; j<=n; j++)`	1	$m(n+1)$	$mn+m$

`c[i][j] = a[i][j]`
`+ b[i][j];`
	1	mn	mn

`}`	0	–	0
Total			$2mn+2m+1$

The above thod is both excessively difficult and, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the **basic operation**, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

Trade-off

There is often a **time-space-tradeoff** involved in a problem, that is, it cannot be solved with few computing time and low memory consumption. One has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

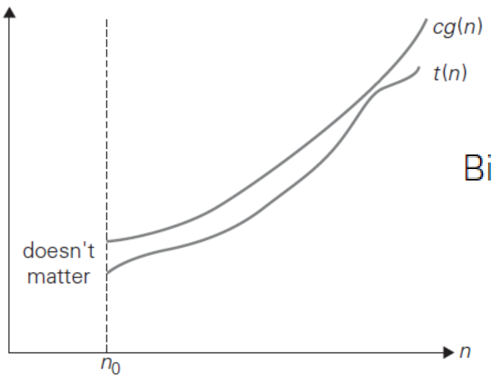
3. Asymptotic Notations

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, computer scientists use three notations: O(big oh), Ω(big omega), Θ (big theta) and *o*(little oh)

3.1. Big-Oh notation

Definition: A function *t(n)* is said to be in *O(g(n)),* denoted *t(n)* ∈*O(g(n)),* if t (n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \le c\, g(n) \text{ for all } n \ge n_0.$$



Big-oh notation: $t(n) \in O(g(n))$.

Informally, *O(g(n))* is the set of all functions with a lower or same order of growth as *g(n)*

Examples: $n \in O(n^2)$, $100n + 5 \in O(n^2)$, $\frac{1}{2}n(n-1) \in O(n^2)$.

$n^3 \notin O(n^2)$, $0.00001n^3 \notin O(n^2)$, $n^4 + n + 1 \notin O(n^2)$.

As another example, let us formally prove **$100n + 5 \in O(n^2)$**

$$100n + 5 \le 100n + n \text{ (for all } n \ge 5) = 101n \le 101n^2. \ (c=101,\ n_0=5)$$

Note that the definition gives us a lot of freedom in choosing specific values for constants ***c*** and ***n_0***.

Example: To prove $n^2 + n = O(n^3)$

Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$

Notice that if $n \ge 1$, $n \le n^3$ is clear.

Also, notice that if $n \ge 1$, $n^2 \le n^3$ is clear.

Therefore,

$$n^2 + n \le n^3 + n^3 = 2n^3$$

We have just shown that

$$n^2 + n \le 2n^3 \text{ for all } n \ge 1$$

Thus, we have shown that $n^2 + n = O(n^3)$
(by definition of Big-O, with $n_0 = 1$, and $c = 2$.)

Strategies for Big-O Sometimes the easiest way to prove that f(n) = O(g(n)) is to take c to be the sum of the positive coefficients of f(n). We can usually ignore the negative coefficients.

Example: To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \ge n_0 = 1$,

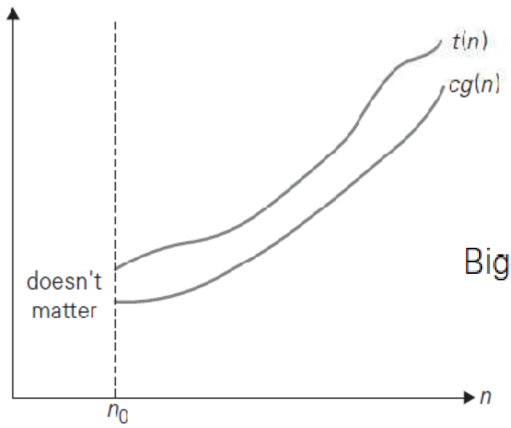
$$5\,n^2 + 3\,n + 20 \le 5\,n^2 + 3\,n^2 + 20\,n^2 = 28\,n^2,$$

thus $5n^2 + 3n + 20 = O(n^2)$.

3.2. Omega notation

Definition: A function t(n) is said to be in Ω(g(n)), denoted ***t(n) ∈Ω(g(n)),*** if t(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq c\, g(n) \text{ for all } n \geq n_0.$$



Big-omega notation: $t(n) \in \Omega(g(n))$.

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$**:**
$n^3 \geq n^2$ for all $n \geq 0$,
i.e., we can select c = 1 and $n_0 = 0$.

Example: $n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$, but $100n + 5 \notin \Omega(n^2)$.

Example: To prove $n^3 + 4n^2 = \Omega(n^2)$

Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$

It is not too hard to see that if $n \geq 0$,

$$n^3 \leq n^3 + 4n^2$$

We have already seen that if $n \geq 1$,

$$n^2 \leq n^3$$

Thus when $n \geq 1$,

$$n^2 \leq n^3 \leq n^3 + 4n^2$$

Therefore,

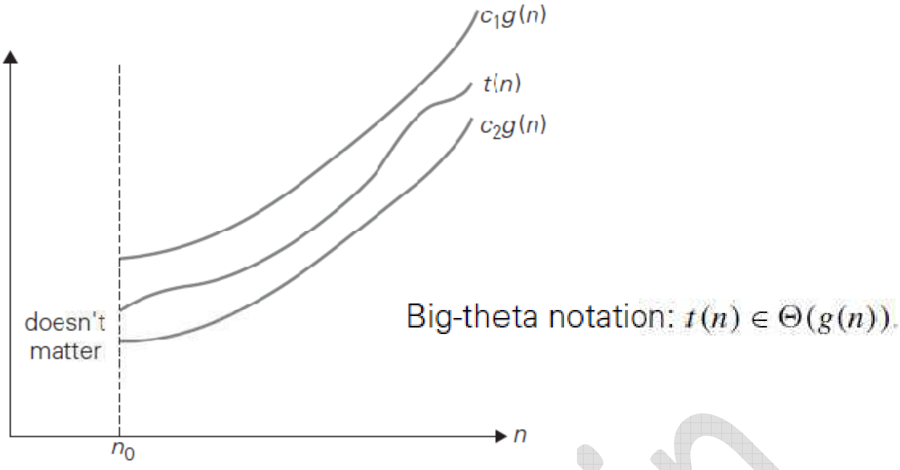
$$1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1$$

Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Ω, with $n_0 = 1$, and $c = 1$.)

3.3. Theta notation

A function t(n) is said to be in **Θ(g(n)),** denoted **t(n) ∈ Θ(g(n)),** if t (n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2\, g(n) \leq t(n) \leq c_1 g(n) \quad \text{for all } n \geq n_0.$$



Big-theta notation: $t(n) \in \Theta(g(n))$.

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \ge \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \ge 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Example: $n^2 + 5n + 7 = \Theta(n^2)$

When $n \ge 1$,

$$n^2 + 5n + 7 \le n^2 + 5n^2 + 7n^2 \le 13n^2$$

When $n \ge 0$,

$$n^2 \le n^2 + 5n + 7$$

Thus, when $n \ge 1$

$$1n^2 \le n^2 + 5n + 7 \le 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

Strategies for Ω and Θ

- Proving that a f(n) = Ω(g(n)) often requires more thought.
  - Quite often, we have to pick c < 1.
  - A good strategy is to pick a value of c which you think will work, and determine which value of n_0 is needed.
  - Being able to do a little algebra helps.
  - We can sometimes simplify by ignoring terms of f(n) with the positive coefficients.
- The following theorem shows us that proving f(n) = Θ(g(n)) is nothing new:

  Theorem: f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).

  Thus, we just apply the previous two strategies.

Show that $\frac{1}{2}n^2 + 3n = \Theta(n^2)$

Notice that if $n \geq 1$,

$$\frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n^2 = \frac{7}{2}n^2$$

Thus,

$$\frac{1}{2}n^2 + 3n = O(n^2)$$

Also, when $n \geq 0$,

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 + 3n$$

So

$$\frac{1}{2}n^2 + 3n = \Omega(n^2)$$

Since $\frac{1}{2}n^2 + 3n = O(n^2)$ and $\frac{1}{2}n^2 + 3n = \Omega(n^2)$,

$$\frac{1}{2}n^2 + 3n = \Theta(n^2)$$

Show that $(n \log n - 2n + 13) = \Omega(n \log n)$

Proof: We need to show that there exist positive constants c and n_0 such that

$$0 \leq c\,n \log n \leq n \log n - 2\,n + 13 \text{ for all } n \geq n_0.$$

Since $\quad n \log n - 2\,n \leq n \log n - 2\,n + 13,$

we will instead show that

$$c\,n \log n \leq n \log n - 2\,n,$$

which is equivalent to

$$c \leq 1 - \frac{2}{\log n}, \text{ when } n > 1.$$

If $n \geq 8$, then $2/(\log n) \leq 2/3$, and picking $c = 1/3$ suffices. Thus if $c = 1/3$ and $n_0 = 8$, then for all $n \geq n_0$, we have

$$0 \leq c\,n \log n \leq n \log n - 2\,n \leq n \log n - 2\,n + 13.$$

Thus $(n \log n - 2\,n + 13) = \Omega(n \log n)$.

Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

We need to find positive constants c_1, c_2, and n_0 such that

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0$$

Dividing by n^2, we get

$$0 \le c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

$c_1 \le \frac{1}{2} - \frac{3}{n}$ holds for $n \ge 10$ and $c_1 = 1/5$

$\frac{1}{2} - \frac{3}{n} \le c_2$ holds for $n \ge 10$ and $c_2 = 1$.

Thus, if $c_1 = 1/5$, $c_2 = 1$, and $n_0 = 10$, then for all $n \ge n_0$,

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0.$$

Thus we have shown that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

3.4. Little Oh The function **f(n) = o(g(n))** [i.e f of n is a little oh of g of n] if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example: The function $3n + 2 = o(n^2)$ since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log \log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \ne o(n)$. $6 * 2^n + n^2 \ne o(2^n)$. □

For comparing the order of growth limit is used

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

If the case-1 holds good in the above limit, we represent it by little-oh.

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2 - n}{n^2} = \frac{1}{2}\lim_{n\to\infty}(1 - \frac{1}{n}) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$. ■

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty}\frac{(\log_2 e)\,\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e\,\lim_{n\to\infty}\frac{1}{\sqrt{n}} = 0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called ***little-oh notation***: $\log_2 n \in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.) ■

3.5. Basic asymptotic Efficiency Classes

Class	*Name*	*Comments*
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

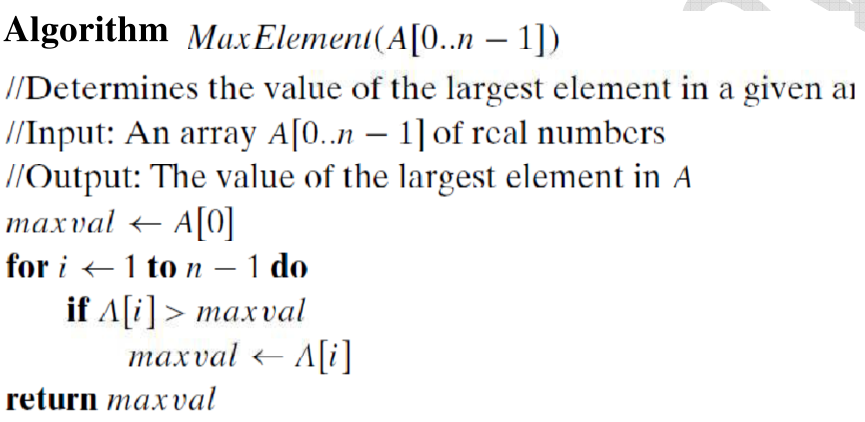
3.6. Mathematical Analysis of Non-recursive & Recursive Algorithms

Analysis of Non-recursive Algorithms

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closedform formula for the count or, at the very least, establish its order of growth.

Example-1: To find maximum element in the given array



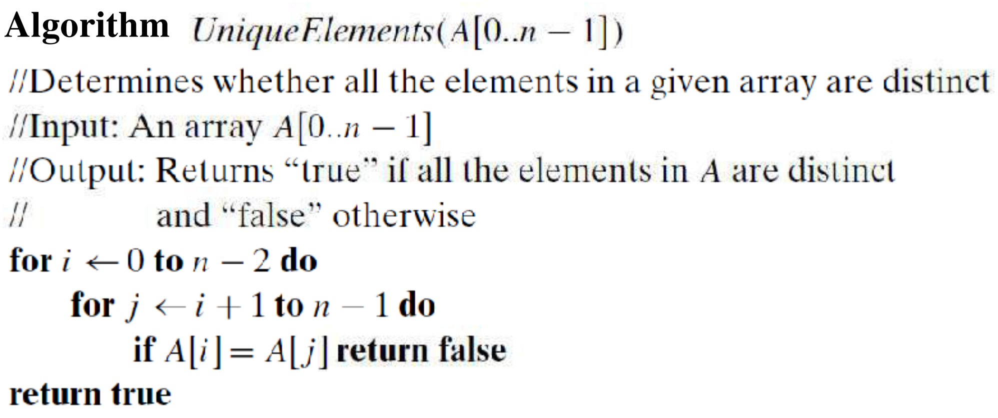
```
Algorithm  MaxElement(A[0..n − 1])
//Determines the value of the largest element in a given a
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Here comparison is the basic operation.

Note that number of comparisions will be same for all arrays of size n. Therefore, no need to distinguish worst, best and average cases.

Total number of basic operations (comparison) are, $C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$

Example-2: To check whether all the elements in the given array are distinct



```
Algorithm  UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Here basic operation is comparison. The maximum no. of comparisons happen in the worst case. (i.e. all the elements in the array are distinct and algorithms return *true*).

Total number of basic operations (comparison) in the worst case are,

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

Other than the worst case, the total comparisons are **less than** $\boldsymbol{\frac{1}{2}n^2}$. (For example if the first two elements of the array are equal, only one comparison is computed). So in general **C(n) =O(n^2)**

Example-3: To perform matrix multiplication

```
Algorithm MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two square matrices of order n by the definition-based algorithm
    //Input: Two n × n matrices A and B
    //Output: Matrix C = AB
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```

Number of basic operations (multiplications) is

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

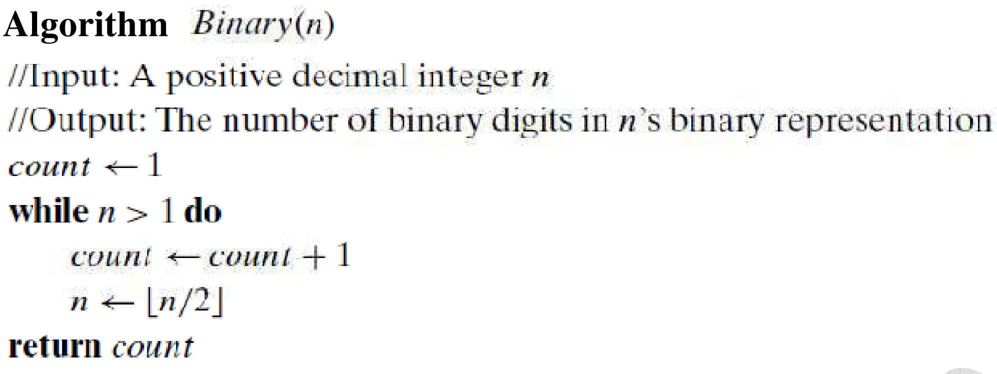
Total running time:

$$T(n) \approx c_m M(n) = c_m n^3$$

Suppose if we take into account of addition; Algoritham also have same number of additions *A(n) = n^3*

Total running time: $T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a)n^3.$

Example-4: To count the bits in the binary representation



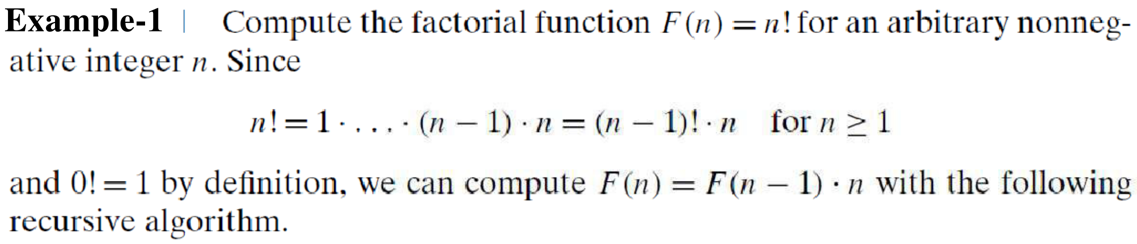
```
Algorithm  Binary(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

The basic operation is count=count + 1 repeats $\lfloor \log_2 n \rfloor + 1$ no. of times

Analysis of Recursive Algorithms

General plan for analyzing the time efficiency of recursive algorithms

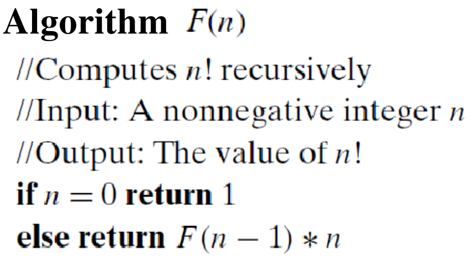
1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
4. Solve the recurrence or, at least, ascertain the order of growth of its solution.



Example-1 | Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n. Since

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.



```
Algorithm  F(n)
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) ∗ n
```

Since the function *F(n)* is computed according to the formula

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$$

The number of multiplications *M(n)* needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

Such equations are called **recurrence Relations**

Condition that makes the algorithm stop ***if n = 0 return 1.*** Thus recurrence relation and initial condition for the algorithm's number of multiplications M(n) can be stated as

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

We can use backward substitutions method to solve this

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3. \\ &\ldots \\ &= M(n-i) + i = \cdots = M(n-n) + n = n. \end{aligned}$$

Example-2: Tower of Hanoi puzzle. In this puzzle, There are ***n*** disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

The problem has an elegant recursive solution, which is illustrated in Figure.

- To move n>1 disks from peg 1 to peg 3 (with peg 2 as auxiliary),
  - we first move recursively n-1 disks from peg 1 to peg 2 (with peg 3 as auxiliary),
  - then move the largest disk directly from peg 1 to peg 3, and,
  - finally, move recursively n-1 disks from peg 2 to peg 3 (using peg 1 as auxiliary).
- If n = 1, we move the single disk directly from the source peg to the destination peg.

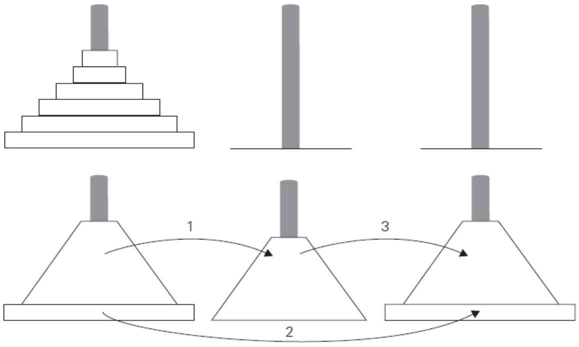


Figure: Recursive solution to the Tower of Hanoi puzzle

The number of moves ***M(n)*** depends only on n. The recurrence equation is

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

We have the following recurrence relation for the number of moves M(n):

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1$$
$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n-1)+1 && \text{sub. } M(n-1)=2M(n-2)+1 \\ &= 2[2M(n-2)+1]+1 = 2^2M(n-2)+2+1 && \text{sub. } M(n-2)=2M(n-3)+1 \\ &= 2^2[2M(n-3)+1]+2+1 = 2^3M(n-3)+2^2+2+1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be 2^4 M(n − 4) + 2^3 + 2^2 + 2 + 1, and generally, after *i* substitutions, we get

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1$$

Since the initial condition is specified for n = 1, which is achieved for i = n - 1, we get the following formula for the solution to recurrence,

$$\begin{aligned} M(n) &= 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 \end{aligned}$$

Alternatively, by counting the number of nodes in the tree obtained by recursive calls, we can get the total number of calls made by the Tower of Hanoi algorithm:

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure } \quad) = 2^n - 1$$

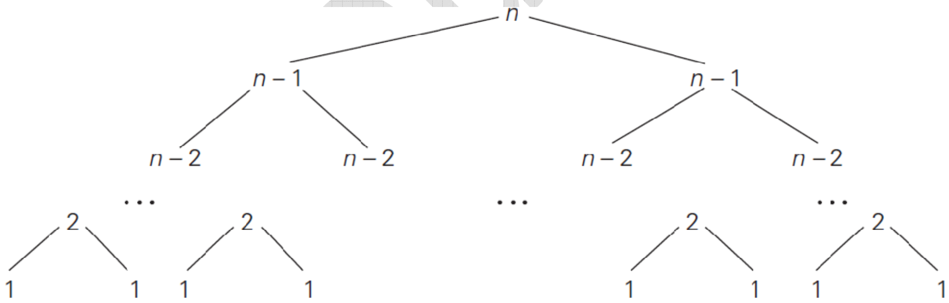


Figure: Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

Example-3

```
ALGORITHM BinRec(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    if $n = 1$ return 1
    else return $BinRec(\lfloor n/2 \rfloor) + 1$
```

The recurrence relation can be written as

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

Also note that A(1) = 0.

The standard approach to solving such a recurrence is to **solve it only for n = 2^k** and then take advantage of the theorem called the **smoothness rule** which claims that under very broad assumptions the order of growth observed for n = 2^k gives a correct answer about the order of growth for all values of n.

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1\\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1\\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots\\
&\dots\\
&= A(2^{k-i}) + i\\
&\dots\\
&= A(2^{k-k}) + k.
\end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

4. Important Problem Types

In this section, we are going to introduce the most important problem types: Sorting, Searching, String processing, Graph problems, Combinatorial problems.

4.1. Sorting

The sorting problem is to rearrange the items of a given list in **non-decreasing order.** As a practical matter, we usually need to sort lists of numbers, characters from an alphabet or character strings.

Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations. Some of the algorithms are simple but relatively slow, while others are faster but more complex; some work better on randomly ordered inputs, while others do better on almost-sorted lists; some are suitable only for lists residing in the fast memory, while others can be adapted for sorting large files stored on a disk; and so on.

Two properties of sorting algorithms deserve special mention. A sorting algorithm is called ***stable*** if it preserves the relative order of any two equal elements in its input. The second notable feature of a sorting algorithm is the amount of ***extra memory*** the algorithm requires. An algorithm is said to be in-place if it does not require extra memory, except, possibly, for a few memory units.

4.2. Searching

The searching problem deals with finding a given value, called a search key, in a given set. (or a multiset, which permits several elements to have the same value). There are plenty of searching algorithms to choose from. They range from the straightforward **sequential search** to a spectacularly efficient but limited **binary search** and algorithms based on representing the underlying set in a different form more conducive to searching. The latter algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.

4.3. String Processing

In recent decades, the rapid proliferation of applications dealing with non-numerical data has intensified the interest of researchers and computing practitioners in string-handling algorithms. A string is a sequence of characters from an alphabet. String-processing algorithms have been important for computer science in conjunction with computer languages and compiling issues.

4.4. Graph Problems

One of the oldest and most interesting areas in algorithmics is graph algorithms. Informally, a graph can be thought of as a collection of points called **vertices**, some of which are connected by line segments called **edges**. Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games. Studying different technical and social aspects of the Internet in particular is one of the active areas of current research involving computer scientists, economists, and social scientists.

4.5. Combinatorial Problems

Generally speaking, combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint. Their difficulty stems from the following facts. First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances. Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.

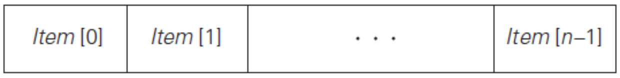
5. Fundamental Data Structures

Since the vast majority of algorithms of interest operate on data, particular ways of organizing data play a critical role in the design and analysis of algorithms. A **data structure** can be defined as a particular scheme of organizing related data items.

5.1. Linear Data Structures

The two most important elementary data structures are the array and the linked list.

A (one-dimensional) **array** is a sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.



Array of n elements.

A **linked list** is a sequence of zero or more elements called nodes, each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list. In a **singly linked list**, each node except the last one contains a single pointer to the next element. Another extension is the structure called the **doubly linked list**, in which every node, except the first and the last, contains pointers to both its successor and its predecessor.

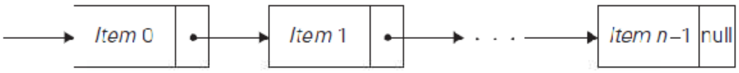


FIGURE 1.4 Singly linked list of n elements.

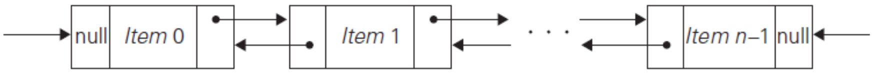


FIGURE 1.5 Doubly linked list of n elements.

A **list** is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element. Two special types of lists, stacks and queues, are particularly important.

A **stack** is a list in which insertions and deletions can be done only at the end. This end is called the top because a stack is usually visualized not horizontally but vertically—akin to a stack of plates whose "operations" it mimics very closely.

A **queue**, on the other hand, is a list from which elements are deleted from one end of the structure, called the front (this operation is called dequeue), and new elements are added to the other end, called the rear (this operation is called enqueue). Consequently, a queue operates in a "first-in–first-out" (FIFO) fashion—akin to a queue of customers served by a single teller in a bank. Queues also have many important applications, including several algorithms for graph problems.

Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such applications is called a **priority queue**. A priority queue is a collection of data items from a totally ordered universe (most often, integer or real numbers). The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element.

5.2. Graphs

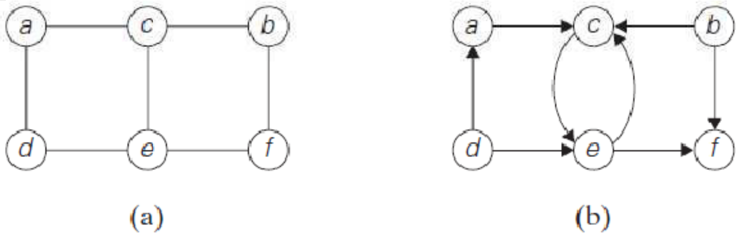
A graph is informally thought of as a collection of points in the plane called "vertices" or nodes," some of them connected by line segments called "edges" or "arcs." A graph G is called **undirected** if every edge in it is undirected. A graph whose every edge is directed is called **directed**. Directed graphs are also called **digraphs**.

The graph depicted in Figure (a) has six vertices and seven undirected edges:

V = {a, b, c, d, e, f }, E = {(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)}.

The digraph depicted in Figure 1.6b has six vertices and eight directed edges:

V = {a, b, c, d, e, f }, E = {(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)}.



(a) Undirected graph. (b) Digraph.

Graph Representations - Graphs for computer algorithms are usually represented in one of two ways: the ***adjacency matrix*** and ***adjacency lists***.

The **adjacency matrix** of a graph with n vertices is an **n x n** boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i^{th} row and the j^{th}

column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge.

The **adjacency lists** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge).

$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 0 & 0 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 0 & 1 \\ c & 1 & 1 & 0 & 0 & 1 & 0 \\ d & 1 & 0 & 0 & 0 & 1 & 0 \\ e & 0 & 0 & 1 & 1 & 0 & 1 \\ f & 0 & 1 & 0 & 0 & 1 & 0 \end{array}$$

(a)

$\boxed{a} \to c \to d$

$\boxed{b} \to c \to f$

$\boxed{c} \to a \to b \to e$

$\boxed{d} \to a \to e$

$\boxed{e} \to c \to d \to f$

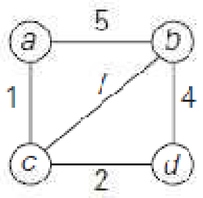
$\boxed{f} \to b \to e$

(b)

GURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure

Weighted Graphs: A weighted graph (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called weights or costs.

Among the many properties of graphs, two are important for a great number of applications: connectivity and acyclicity. Both are based on the notion of a path. A path from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.



(a)

$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & \infty & 5 & 1 & \infty \\ b & 5 & \infty & / & 4 \\ c & 1 & 7 & \infty & 2 \\ d & \infty & 4 & 2 & \infty \end{array}$$

(b)

$\boxed{a} \to b, 5 \to c, 1$

$\boxed{b} \to a, 5 \to c, 7 \to d, 4$

$\boxed{c} \to a, 1 \to b, 7 \to d, 2$

$\boxed{d} \to b, 4 \to c, 2$

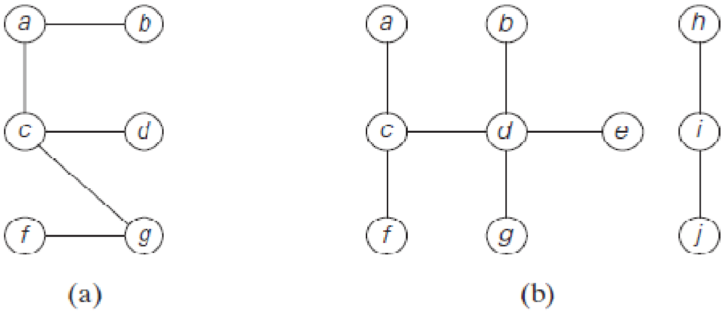
(c)

(a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

A graph is said to be **connected** if for every pair of its vertices u and v there is a path from u to v. Graphs with several connected components do happen in real-world applications. It is important to know for many applications whether or not a graph under consideration has cycles. A **cycle** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once.

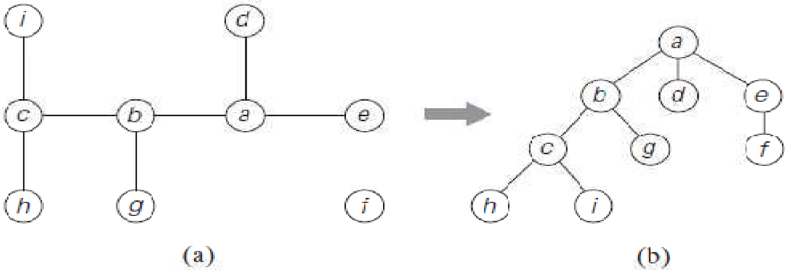
5.3. Trees

A **tree** (more accurately, a free tree) is a connected acyclic graph. A graph that has no cycles but is not necessarily connected is called a **forest**: each of its connected components is a tree. Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices: |E| = |V| - 1



(a) Tree (b) Forest

Rooted Trees: Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree. A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on.

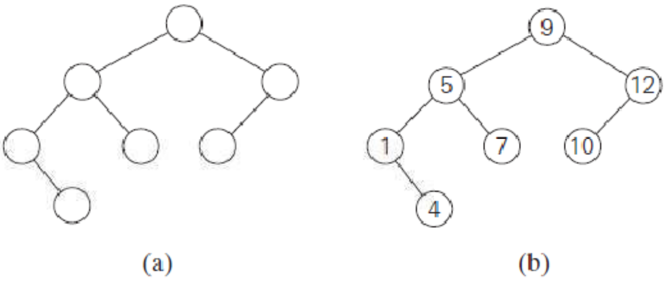


(a) Free tree. (b) Its transformation into a rooted tree.

The **depth** of a vertex v is the length of the simple path from the root to v. The **height** of a tree is the length of the longest simple path from the root to a leaf.

Ordered Trees- An ordered tree is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right. A **binary tree** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a left child or a right child of its parent; a binary tree may also be empty.

If a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called **binary search trees**. Binary trees and binary search trees have a wide variety of applications in computer science.



2 (a) Binary tree. (b) Binary search tree.

5.4. Sets and Dictionaries

A **set** can be described as an unordered collection (possibly empty) of distinct items called **elements** of the set. A specific set is defined either by an explicit listing of its elements (e.g., S = {2, 3, 5, 7}) or by specifying a property that all the set's elements and only they must satisfy (e.g., S = {n: n is a prime number smaller than 10}).

The most important **set operations** are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets.

Sets can be **implemented** in computer applications in two ways. The first considers only sets that are subsets of some large set U, called the universal set. If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a **bit vector**, in which the i^{th} element is 1 if and only if the i^{th} element of U is included in set S.

The second and more common way to represent a set for computing purposes is to use the **list** structure to indicate the set's elements. This is feasible only for finite sets. The requirement for uniqueness is sometimes circumvented by the introduction of a multiset, or bag, an unordered collection of items that are not necessarily distinct. Note that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

Dictionary: In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the **dictionary**. An efficient implementation of a dictionary has to strike a compromise between the efficiency of searching and the efficiencies of the other two operations. They range from an unsophisticated use of arrays (sorted or not) to much more sophisticated techniques such as hashing and balanced search trees.

A number of applications in computing require a dynamic partition of some n-element set into a collection of disjoint subsets. After being initialized as a collection of n one-element subsets, the collection is subjected to a sequence of intermixed union and search operations. This problem is called the **set union** problem.

*****

Vivekananda
College of Engineering & Technology

Lecture Notes
on

15CS43
Design and Analysis of Algorithms
(CBCS Scheme)

Prepared by

Mr. Harivinod N
Assistant Professor,
Dept. of Computer Science and Engineering,
VCET Puttur

Feb 2017

Module-2 : Divide and Conquer

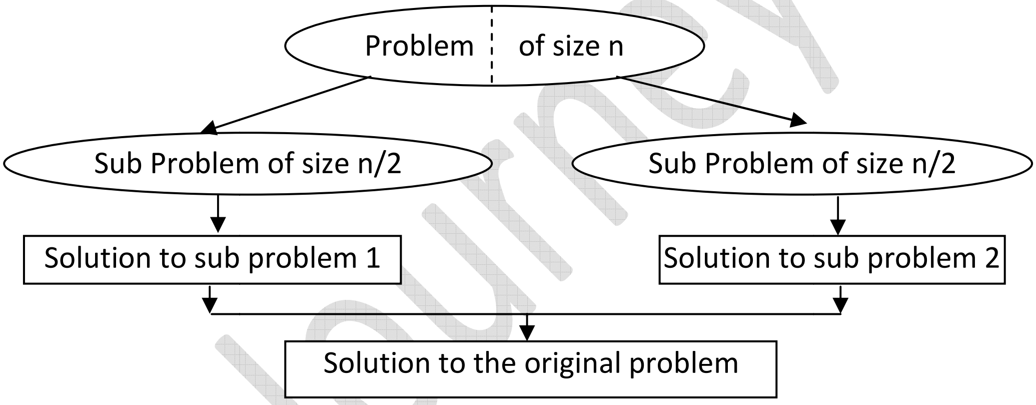
Contents

1. General method:

Divide and Conquer is one of the best-known general algorithm design technique. It works according to the following general plan:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

A typical case with k=2 is diagrammatically shown below.


Problem of size n
Sub Problem of size n/2
Sub Problem of size n/2
Solution to sub problem 1
Solution to sub problem 2
Solution to the original problem

Control Abstraction for divide and conquer:

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances $P_1, P_2, \ldots, P_k$, $k \geq 1$;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC($P_1$),DAndC($P_2$),...,DAndC($P_k$));
    }
}
```

In the above specification,

- Initially ***DAndC(P)*** is invoked, where 'P' is the problem to be solved.
- ***Small (P)*** is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this so, the function '***S'*** is invoked. Otherwise, the problem P is divided into smaller sub problems. These sub problems P_1, P_2 ...P_k are solved by recursive application of ***DAndC***.
- ***Combine*** is a function that determines the solution to P using the solutions to the 'k' sub problems.

2. Recurrence equation for Divide and Conquer:

If the size of problem 'p' is n and the sizes of the 'k' sub problems are n_1, n_2 ….n_k, respectively, then the computing time of divide and conquer is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

Where,

- T(n) is the time for divide and conquer method on any input of size n and
- g(n) is the time to compute answer directly for small inputs.
- The function f(n) is the time for dividing the problem 'p' and combining the solutions to sub problems.

For divide and conquer based algorithms that produce sub problems of the same type as the original problem, it is very natural to first describe them by using recursion.

More generally, an instance of size **n** can be divided into **b** instances of size **n/b**, with **a** of them needing to be solved. (Here, a and b are constants; **a>=1 and b > 1**.). Assuming that size **n** is a power of **b** (i.e. n = b^k), to simplify our analysis, we get the following recurrence for the running time T(n):

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \qquad \text{..... (1)}$$

where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Substitution Method - One of the methods for solving the recurrence relation is called the substitution method. This method repeatedly makes substitution for each occurrence of the function T in the right hand side until all such occurrences disappear.

Master Theorem - The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the master theorem.

It states that, in recurrence equation **T(n) = aT(n/b) + f (n)**, If f (n)$\in\Theta$ (n^d) where d $\geq$ 0 then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the *O* and Ω notations, too.

For example, the recurrence for the number of additions A(n) made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size n = 2^k is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, a = 2, b = 2, and d = 0; hence, since a >b^d,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Problems on Substitution method & Master theorem to solve the recurrence relation

Exercise 3.1 Horowitz

Solve following recurrence relation.

$T(n) = 2\,T(n/2) + n$, $T(1) = 2$. using substitution method

Soln! $T(n) = 2\,T(n/2) + n$.

$= 2\left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 4\,T\left(\frac{n}{4}\right) + 2n$

$= 4\left[2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n = 8\,T\left(\frac{n}{8}\right) + 3n$

$\vdots$

$= 2^i\, T\left(\frac{n}{2^i}\right) + in$; $1 \leq i \leq \log_2 n$

The maximum value of $i = \log_2 n$ [∵ then only we get $T(1)$].

$= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + n \cdot \log_2 n$

$= n \cdot T(1) + n \log_2 n$

$= 2n + n \log_2 n$

$= \underline{\underline{\theta(n \log_2 n)}}$

Solution using master theorem

Here $a=2$, $b=2$, $f(n) = n = \theta(n^1) \Rightarrow d = 1$

Also we see that $a = b^d$ $[2 = 2^1]$

∴ As per case-2 of master theorem.

$T(n) = \theta(n^d \cdot \log_2 n)$

$T(n) = \underline{\underline{\theta(n \log_2 n)}}$

Exercise 3.2: Solve by substitution method.

$a=1,\ b=2,\ f(n)=c$

Soln: $T(n) = T(\frac{n}{2}) + c$

$= [T(\frac{n}{4})+c]+c = T(\frac{n}{4})+2c$

$= [T(\frac{n}{8})+c]+c = T(\frac{n}{8})+3c$

$\vdots$

$= T(\frac{n}{2^i}) + ic \quad ; \quad 1 \le i \le \log_2 n.$

$= T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \cdot c$

$= T(1) + c.\log_2 n$

Assuming $T(1) = K$ some constant

$T(n) = c\log_2 n + K.$

$T(n) = \underline{\underline{\theta(\log_2 n)}}.$

Soln Using master theorem

Here $a=1,\ b=2,\ f(n)=c=\theta(1) = \theta(n^0).$

$\Rightarrow d=0.$

As $a=b^d\ [1=2^0]$, case-2 of master theorem is applied.

$T(n) = \theta(n^d.\log_2 n)$

$T(n) = \underline{\underline{\theta(\log_2 n)}}.$

Exercise 3.3 Solve recurrence relation

$a=2,\ b=2,\ f(n)=cn.$

Soln: $T(n) = 2\cdot T(\frac{n}{2}) + cn = 2[2\cdot T(\frac{n}{4}) + c\frac{n}{2}] + cn$

$= 4\cdot T(\frac{n}{4}) + 2\cdot cn = 4[2\cdot T(\frac{n}{8}) + c\cdot\frac{n}{4}] + 2cn$

$= 8\cdot T(\frac{n}{8}) + 3cn$

$\vdots$

$= 2^i T(\frac{n}{2^i}) + icn; \quad 1 \le i \le \log_2 n.$

$\vdots$

$= 2^{\log_2 n}\cdot T(\frac{n}{2^{\log n}}) + \log_2 n\cdot c\cdot n$

$= n\cdot T(1) + c\cdot n\log_2 n$

Assuming $T(1)=K$; some constant

$T(n) = c\cdot n\log_2 n + K\cdot n.$

$= \theta(n\log_2 n)$

Soln Using master theorem

$f(n) = cn = \theta(n) = \theta(n^1) \Rightarrow d=1$

Here $a=2,\ b=2,$

Here $a=b^d\ [2=2^1]$, case-2 of master theorem

$T(n) = \theta(n^d \log_b n)$

$= \theta(n\cdot\log_2 n)$

Ex.3.5 Solve $T(n) = 9\cdot T(\frac{n}{3}) + 4n^6$; $n \geq 3$
n is power of 3

Soln $T(n) = 9\cdot T(\frac{n}{3}) + 4n^6$

$= 9[9\cdot T(\frac{n}{9}) + 4(\frac{n}{3})^6] + 4\cdot n^6$

$= 81\cdot T(\frac{n}{9}) + \frac{9\cdot 4\cdot n^6}{3^6} + 4\cdot n^6$

$= 81\cdot T(\frac{n}{9}) + \frac{4\cdot n^6}{3^4} + 4n^6$

$= 3^4\cdot T(\frac{n}{3^2}) + \left(\frac{1+3^4}{3^4}\right)4n^6$

$= 3^4\cdot T(\frac{n}{3^2}) + K\cdot n^6$ | K is a constant

$= \vdots$

$= 3^{2i}\cdot T(\frac{n}{3^i}) + Kn^6$; $1 \leq i \leq \log_3 n$

$= 3^{2\cdot\log_3 n}\cdot T\left(\frac{n}{3^{\log_3 n}}\right) + Kn^6$

$= 3^2\cdot 3^{\log_3 n}\cdot T(1) + Kn^6$

Assuming T(1) is constant c

$= 9c\cdot n + Kn^6$

$= \theta(n^6)$

Soln using master theorem

$a=9$, $b=3$, $f(n) = 4n^6 = \theta(n^6) \Rightarrow d=6$.

Since $a<b^d$ ($9<3^6$)

$T(n) = \theta(n^d) = \theta(n^6)$.

(*) Solve $T(n) = 2\cdot T(\frac{n}{2})+1$

Soln $T(n) = 2\cdot[2\cdot T(\frac{n}{4})+1]+1 = 4\,T(\frac{n}{4}) + (2+1)$

$= 4[2\cdot T(\frac{n}{8})+1] + (2+1)$

$= 8\cdot T(\frac{n}{8}) + (4+2+1)$

$\vdots$

$= 2^i T(\frac{n}{2^i}) + (2^{i-1} + 2^{i-2} + \dots 2+1)$

$(\le i \le \log_2 n.$

$= 2^{\log_2 n}\cdot T(1) + (2^i - 1)$

$= n\cdot T(1) + 2^{\log_2 n} - 1$

Assuming $T(1)=1$

$= n+n-1$

$= 2n-1$

$= \Theta(n)$

Soln Using master theorem

$a=2,\ b=2,\ f(n)=1$
$= \Theta(1) = \Theta(n^0) \Rightarrow d=0.$

Since $a > b^d$ $(2 > 2^0)$, case-3 is applied

$T(n) = \Theta(n^{\log_b a})$
$= \Theta(n^{\log_2 2})$
$= \Theta(n)$

$\circledast$ solve $T(n) = T(\frac{n}{2}) + n$

$$T(n) = T(\tfrac{n}{2}) + n$$
$$= T(\tfrac{n}{4}) + \tfrac{n}{2} + n$$
$$= T(\tfrac{n}{8}) + \tfrac{n}{4} + \tfrac{n}{2} + n$$
$$\vdots$$
$$= T\left(\frac{n}{2^i}\right) + \left(\frac{n}{2^{i-1}} + \frac{n}{2^{i-2}} + \cdots \frac{n}{4} + \frac{n}{2} + n\right)$$

$1 \le i \le \log_2 n$

$$= T(1) + \left(\frac{n}{2^{\log_2 n - 1}} + \frac{n}{2^{\log_2 n - 2}} + \cdots \frac{n}{4} + \frac{n}{2} + n\right)$$
$$= T(1) + \frac{n}{(2^{\log_2 n}/2)} + \frac{n}{\left(\frac{2^{\log_2 n}}{2^2}\right)} + \cdots \frac{n}{4} + \frac{n}{2} + n$$

Assume $T(1)=1$,

$$= 1 + 2 + 2^2 + \cdots 2^{\log_2 n - 2} + 2^{\log_2 n - 1} + 2^{\log_2 n}$$
$$= 2^{\log_2 n + 1} - 1 \quad (\because 1 + 2 + 2^2 \cdots 2^k = 2^{k+1} - 1)$$
$$= 2^{\log_2 n} \cdot 2 - 1$$
$$= n \cdot 2 - 1 = \underline{\underline{2n-1}} \in \underline{\underline{\theta(n)}}$$

Soln using master theorem

Here $a=1$, $b=2$ $f(n) = n = \theta(n^1) \Rightarrow d=1$.

Since $a < b^d$ $(1 < 2^1)$

$$T(n) = \theta(n^d)$$
$$= \underline{\underline{\theta(n)}}$$

3. Binary Search

Problem definition: Let a_i, $1 \leq i \leq n$ be a list of elements that are sorted in non-decreasing order. The problem is to find whether a given element x is present in the list or not. If x is present we have to determine a value j (element's position) such that a_j=x. If x is not in the list, then j is set to zero.

Solution: Let **P = (n, a_i…a_l, x)** denote an arbitrary instance of search problem where ***n*** is the number of elements in the list, **a_i…a_l** is the list of elements and **x** is the key element to be searched for in the given list. **Binary search** on the list is done as follows:

Step 1: Pick an index *q* in the middle range [i, *l*] i.e. $q=\lfloor (n+1)/2 \rfloor$ and compare x with a_q.

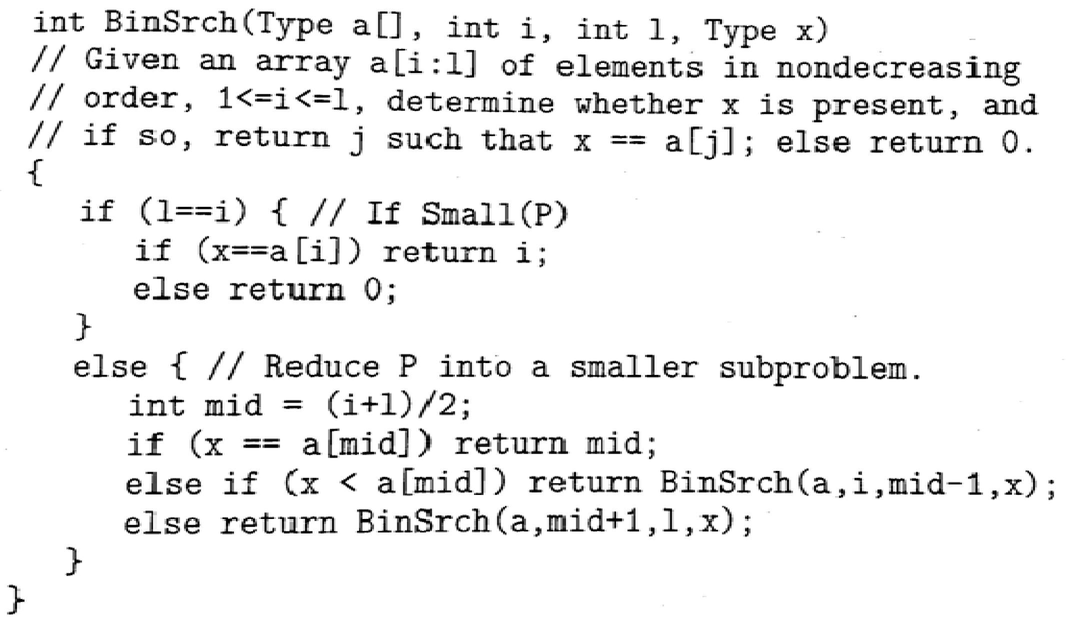
Step 2: if x = a_q i.e key element is equal to mid element, the problem is immediately solved.

Step 3: if x < a_q in this case x has to be searched for only in the sub-list $a_i, a_{i+1}, \ldots\ldots, a_{q-1}$. Therefore problem reduces to **(q-i, a_i…a_{q-1}, x).**

Step 4: if x > a_q , x has to be searched for only in the sub-list $a_{q+1}, \ldots,, a_l$. Therefore problem reduces to **(*l*-i, a_{q+1}…a_l, x).**

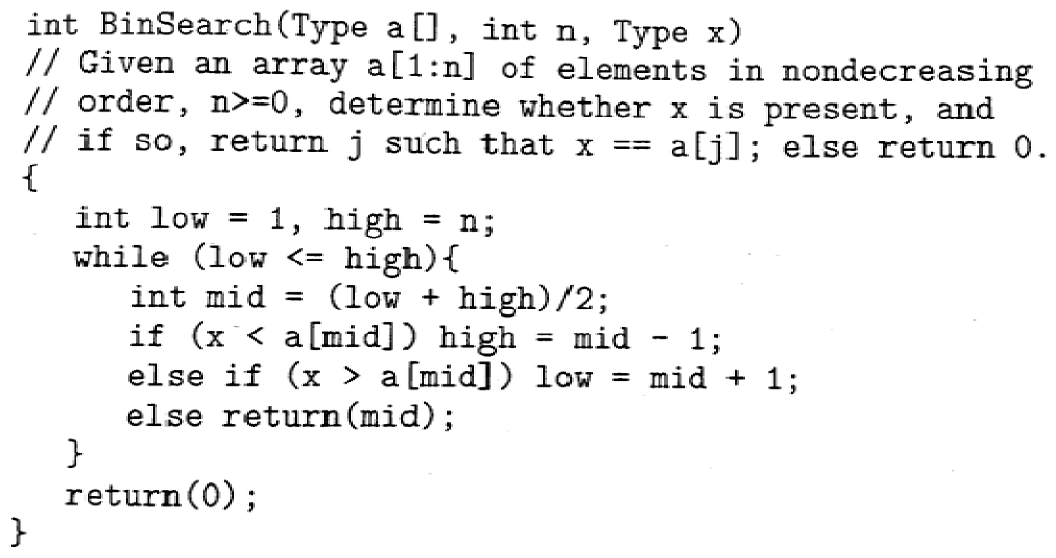
For the above solution procedure, the Algorithm can be implemented as recursive or non-recursive algorithm.

Recursive binary search algorithm

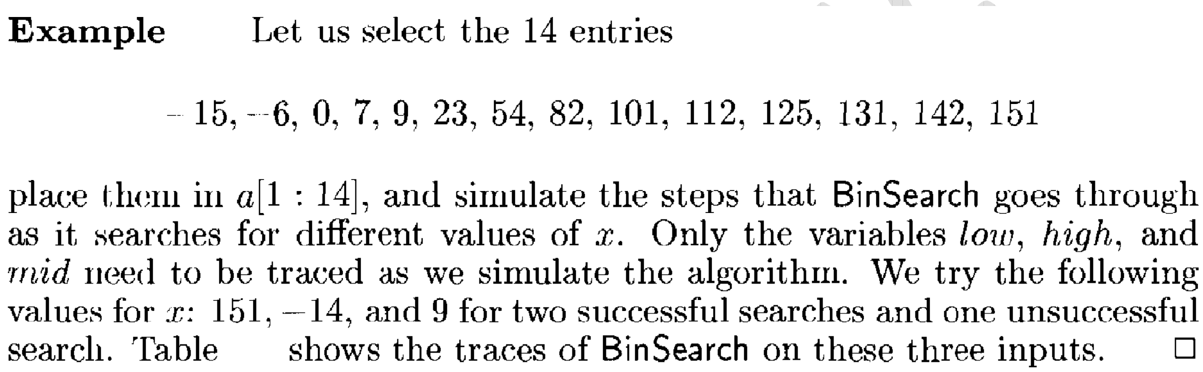


```
int BinSrch(Type a[], int i, int l, Type x)
// Given an array a[i:l] of elements in nondecreasing
// order, 1<=i<=l, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
   if (l==i) { // If Small(P)
      if (x==a[i]) return i;
      else return 0;
   }
   else { // Reduce P into a smaller subproblem.
      int mid = (i+l)/2;
      if (x == a[mid]) return mid;
      else if (x < a[mid]) return BinSrch(a,i,mid-1,x);
      else return BinSrch(a,mid+1,l,x);
   }
}
```

Iterative binary search:



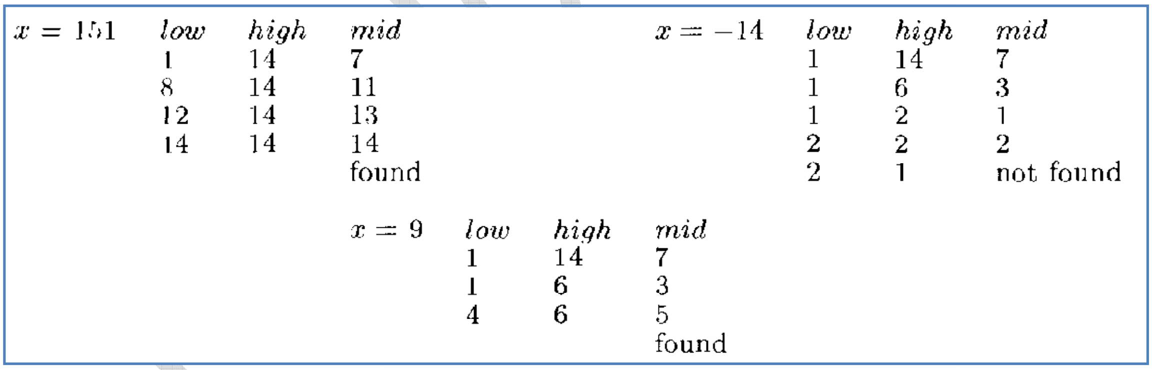
```
int BinSearch(Type a[], int n, Type x)
// Given an array a[1:n] of elements in nondecreasing
// order, n>=0, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
    int low = 1, high = n;
    while (low <= high){
       int mid = (low + high)/2;
       if (x < a[mid]) high = mid - 1;
       else if (x > a[mid]) low = mid + 1;
       else return(mid);
    }
   return(0);
}
```



Example Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

place them in $a[1:14]$, and simulate the steps that BinSearch goes through as it searches for different values of x. Only the variables *low*, *high*, and *mid* need to be traced as we simulate the algorithm. We try the following values for x: 151, -14, and 9 for two successful searches and one unsuccessful search. Table shows the traces of BinSearch on these three inputs. □



$x = 151$	*low*	*high*	*mid*	$x = -14$	*low*	*high*	*mid*
	1	14	7		1	14	7
	8	14	11		1	6	3
	12	14	13		1	2	1
	14	14	14		2	2	2
			found		2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Analysis

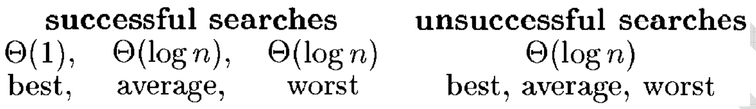
In binary search the basic operation is key comparison. Binary Search can be analyzed with the best, worst, and average case number of comparisons. The numbers of comparisons for the recursive and iterative versions of Binary Search are the same, if comparison counting is relaxed slightly. For Recursive Binary Search, count each pass through the if-then-else block as one comparison. For Iterative Binary Search, count each pass through the while block as one comparison. Let us find out how many such key comparison does the algorithm make on an array of **n** elements.

Best case – Θ(1) In the best case, the key is the middle in the array. A constant number of comparisons (actually just 1) are required.

Worst case - Θ(log$_2$ n) In the worst case, the key does not exist in the array at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling (log$_2$ n) times. Thus, $\lceil \log_2 n \rceil$ comparisons are required.

Sometimes, in case of the successful search, it may take maximum number of comparisons. $\lceil \log_2 n \rceil$. So worst case complexity of successful binary search is Θ (log$_2$ n).

Average case - Θ (log$_2$ n) To find the average case, take the sum of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in array will be searched for, and that the probabilities of searching for each element are uniform.



successful searches			unsuccessful searches
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

How to compute Average case complexity?

Space Complexity - The space requirements for the recursive and iterative versions of binary search are different. Iterative Binary Search requires only a constant amount of space, while Recursive Binary Search requires space proportional to the number of comparisons to maintain the recursion stack.

Advantages: Efficient on very big list, Can be implemented iteratively/recursively.

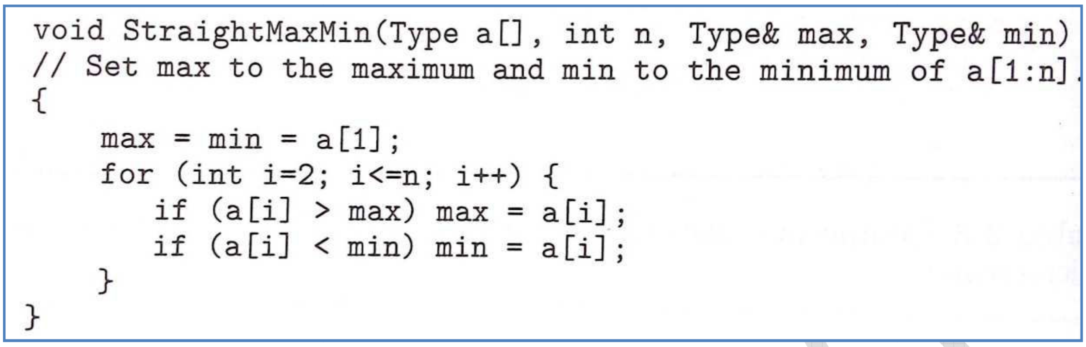
Limitations:

- Interacts poorly with the memory hierarchy
- Requires sorted list as an input
- Due to random access of list element, needs arrays instead of linked list.

4. Finding the maximum and minimum

Problem statement: Given a list of n elements, the problem is to find the maximum and minimum items.

StraightMaxMin: A simple and straight forward algorithm to achieve this is given below.



```
void StraightMaxMin(Type a[], int n, Type& max, Type& min)
// Set max to the maximum and min to the minimum of a[1:n].
{
    max = min = a[1];
    for (int i=2; i<=n; i++) {
       if (a[i] > max) max = a[i];
       if (a[i] < min) min = a[i];
    }
}
```

Explanation:

- ***StraightMaxMin*** requires 2(n-1) comparisons in the best, average & worst cases.
- By realizing the comparison of a[i]>max is false, improvement in a algorithm can be done. Hence we can replace the contents of the for loop by,
  `If(a[i]>Max) then Max = a[i]; Else if (a[i]< min) min=a[i]`
- On the average a[i] is > max half the time. So, the avg. no. of comparison is 3n/2-1.

Algorithm based on Divide and Conquer strategy

Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem. Here 'n' is the no. of elements in the list (a[i],….,a[j]) and we are interested in finding the maximum and minimum of the list. If the list has more than 2 elements, P has to be divided into smaller instances.

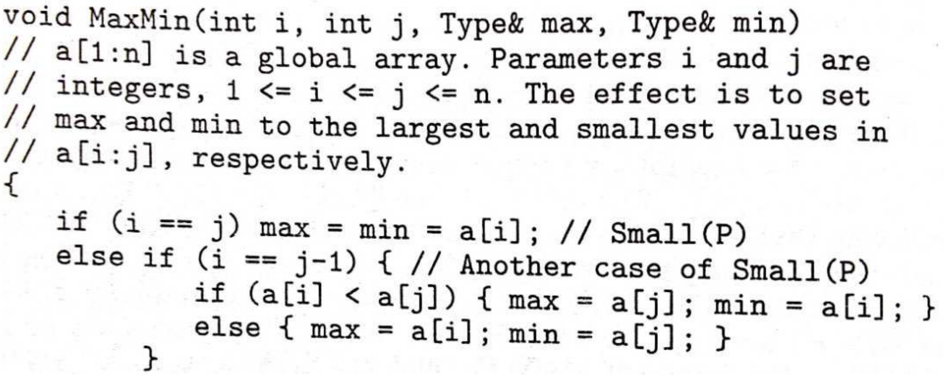
For example, we might divide 'P' into the 2 instances,

P1=([n/2],a[1],……..a[n/2])

P2= (n-[n/2], a[[n/2]+1],….., a[n])

After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm:



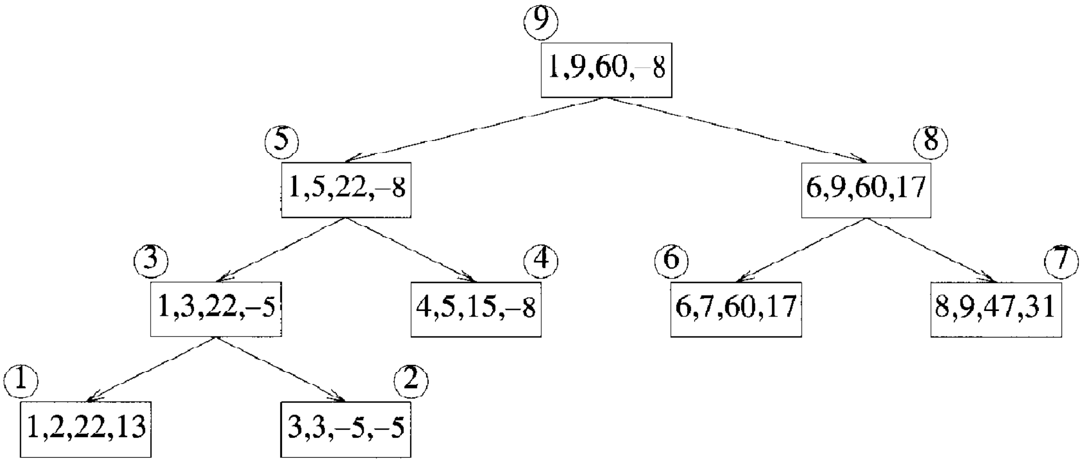
```
void MaxMin(int i, int j, Type& max, Type& min)
// a[1:n] is a global array. Parameters i and j are
// integers, 1 <= i <= j <= n. The effect is to set
// max and min to the largest and smallest values in
// a[i:j], respectively.
{
    if (i == j) max = min = a[i]; // Small(P)
    else if (i == j-1) { // Another case of Small(P)
            if (a[i] < a[j]) { max = a[j]; min = a[i]; }
            else { max = a[i]; min = a[j]; }
         }
else { // If P is not small
          // divide P into subproblems.
       // Find where to split the set.
          int mid=(i+j)/2; Type max1, min1;
       // Solve the subproblems.
         MaxMin(i, mid, max, min);
         MaxMin(mid+1, j, max1, min1);
       // Combine the solutions.
         if (max < max1) max = max1;
         if (min > min1) min = min1;
    }
}
```

Example:

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i, j, max, and min. On the array $a[\,]$ above, the tree of recursive calls of MaxMin is as follows



Analysis - Time Complexity

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \qquad (3.3)$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the straight forward method (2n-2) this method saves 25% in comparisons.

Space Complexity

Compared to the straight forward method, the MaxMin method requires extra stack space for i, j, max, min, max1 and min1. Given n elements there will be $\lfloor log_2 n \rfloor + 1$ levels of recursion and we need to save seven values for each recursive call. (6 + 1 for return address).

5. Merge Sort

Merge sort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A [O ... n - 1] by dividing it into two halves A [0 .. $\lfloor n/2 \rfloor$-1] and A [$\lfloor n/2 \rfloor$ .. n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

```
ALGORITHM  Mergesort($A[0..n-1]$)
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)
        Mergesort($C[0..\lceil n/2 \rceil - 1]$)
        Merge($B, C, A$)    //see below
```

The merging of two sorted arrays can be done as follows.

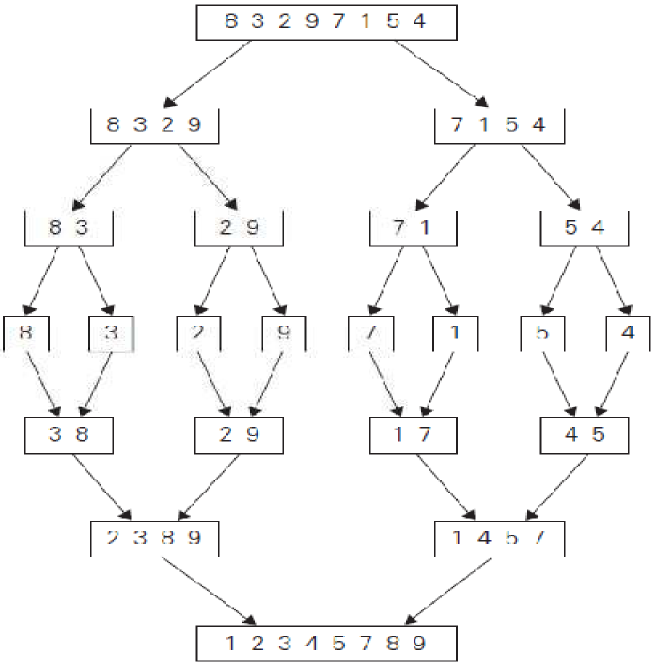
- Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed

- After that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

```
ALGORITHM  $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
    $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$
```

Example:

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in the figure



Analysis

Here the basic operation is key comparison. As merge sort execution does not depend on the order of the data, best case and average case runtime are the same as worst case runtime.

Worst case: During key comparison, neither of the two arrays becomes empty before the other one contains just one element leads to the worst case of merge sort. Assuming for simplicity that total number of elements **n** is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

where, $C_{merge}(n)$ is the number of key comparison made during the merging stage.

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$.
Now,

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Solving the recurrence equation using **master theorem:**
Here a = 2, b = 2, f (n) = n, d = 1. Therefore $2 = 2^1$, case 2 holds in the master theorem
C_{worst} (n) = Θ (n^d log n) = Θ (n^1 log n) = Θ (n log n) Therefore **C_{worst}(n) = Θ (n log n)**

Advantages:

- Number of comparisons performed is nearly optimal.
- For large n, the number of comparisons made by this algorithm in the average case turns out to be about 0.25n less and hence is also in *Θ(n* log *n)*.
- Mergesort will never degrade to O (n^2)
- Another advantage of mergesort over quicksort and heapsort is its **stability**. (A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.)

Limitations:

- The principal shortcoming of mergesort is the linear amount [O(n)] of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

Variations of merge sort

1. The algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. (If n is not a power of 2, only slight bookkeeping complications arise.) This avoids the time and space overhead of using a stack to handle recursive calls.
2. We can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called multiway mergesort.

6. Quick sort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides (or partitions) them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently (e.g., by the same method).

In quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the sub problems.

```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //          indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ←Partition(A[l..r])  //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

Partitioning

We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot. We use the sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.

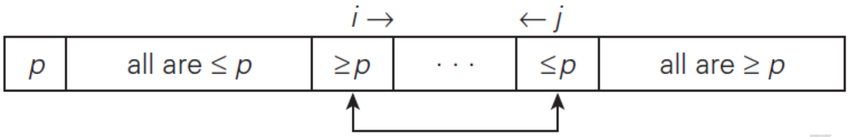
Select the subarray's first element: p = A[l]. Now scan the subarray from both ends, comparing the subarray's elements to the pivot.

- The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
- The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the

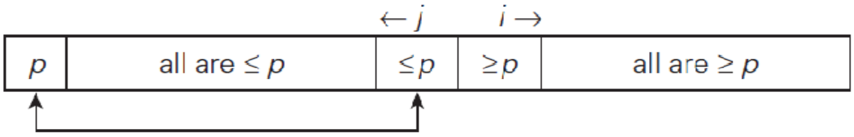
subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.

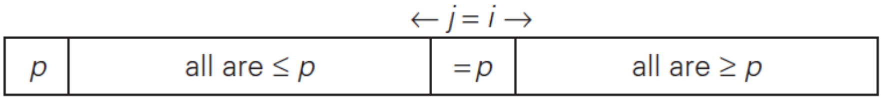
1. If scanning indices i and j have not crossed, i.e., i < j, we simply exchange A[i] and A[j] and resume the scans by incrementing I and decrementing j, respectively:



2. If the scanning indices have crossed over, i.e., i > j, we will have partitioned the subarray after exchanging the pivot with A[j]:



3. If the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p. Thus, we have the subarray partitioned, with the split position s = i = j :



We can combine this with the case-2 by exchanging the pivot with A[j] whenever i≥j

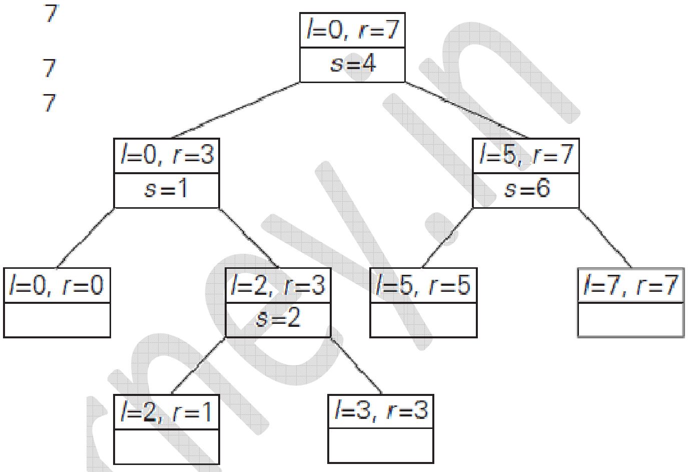
ALGORITHM *HoarePartition*($A[l..r]$)
//Partitions a subarray by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right indices l and r $(l<r)$
//Output: Partition of $A[l..r]$, with the split position returned as this function's value

```
$p \leftarrow A[l]$
$i \leftarrow l;\ j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ until $A[i] \geq p$
    repeat $j \leftarrow j - 1$ until $A[j] \leq p$
    swap($A[i]$, $A[j]$)
until $i \geq j$
swap($A[i]$, $A[j]$)    //undo last swap when $i \geq j$
swap($A[l]$, $A[j]$)
return $j$
```

Note that index *i* can go out of the subarray's bounds in this pseudocode.

Example: Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained.

0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{ij}{4}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9



Analysis

Best Case - Here the basic operation is key comparison. Number of key comparisons made before a partition is achieved is n + 1 if the scanning indices cross over and n if they coincide. If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence,

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

Worst Case – In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays.

Indeed, if A[0..n − 1] is a strictly increasing array and we use A[0] as the pivot, the left-to-right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0: So, after making n + 1 comparisons to get to this partition and exchanging the pivot A[0] with itself, the algorithm will be left with the strictly increasing array A[1..n − 1] to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one A[n−2 .. n−1] has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Average Case - Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s (0 ≤ s ≤ n−1) after n+1comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and n − 1− s elements, respectively. Assuming that the partition split can happen in each position s with the same probability 1/n, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Thus, on the average, quicksort makes only 39% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it usually runs faster than mergesort on randomly ordered arrays of nontrivial sizes. This certainly justifies the name given to the algorithm by its inventor.

Variations

Because of quicksort’s importance, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- Better pivot selection methods such as randomized quicksort that uses a random element or the median-of-three method that uses the median of the leftmost, rightmost, and the middle element of the array
- Switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- Modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot

Limitations

- It is not stable.
- It requires a stack to store parameters of subarrays that are yet to be sorted.
- While Performance on randomly ordered arrays is known to be sensitive not only to implementation details of the algorithm but also to both computer architecture and data type.

7. Stassen's Matrix multiplication

Direct Method: Suppose we want to multiply two n x n matrices, A and B. Their product, C=AB, will be an n by n matrix and will therefore have n^2 elements. The number of multiplications involved in producing the product in this way is $\Theta(n^3)$

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

Divide and Conquer method

Multiplication of 2 × 2 matrices: By using divide-and-conquer approach we can reduce the number of multiplications. Such an algorithm was published by V. Strassen in 1969. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2 × 2 matrices A and B with **just seven multiplications** as opposed to the eight required by the brute-force algorithm. This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$\begin{aligned} m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}), \\ m_2 &= (a_{10} + a_{11}) * b_{00}, \\ m_3 &= a_{00} * (b_{01} - b_{11}), \\ m_4 &= a_{11} * (b_{10} - b_{00}), \\ m_5 &= (a_{00} + a_{01}) * b_{11}, \\ m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}), \\ m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}). \end{aligned}$$

Thus, to multiply two 2×2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions.

Multiplication of n × n matrices – Let A and B be two n × n matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four n/2 × n/2 submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C00 can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. If the seven products of n/2 × n/2 matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Analysis

Here the basic operation is ***multiplication.*** If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n × n matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \cdots$$
$$= 7^iM(2^{k-i}) \cdots = 7^kM(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

This implies ***$M(n) = \Theta(n^{2.807})$*** which is smaller than n^3 required by the brute-force algorithm.

8. Advantages and Disadvantages of Divide & Conquer

Advantages

- **Parallelism:** Divide and conquer algorithms tend to have a lot of inherent parallelism. Once the division phase is complete, the sub-problems are usually independent and can therefore be solved in parallel. This approach typically generates more enough concurrency to keep the machine busy and can be adapted for execution in multi-processor machines.
- **Cache Performance:** divide and conquer algorithms also tend to have good cache performance. Once a sub-problem fits in the cache, the standard recursive solution reuses the cached data until the sub-problem has been completely solved.

- It allows solving **difficult** and often impossible looking problems like the Tower of Hanoi. It reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.
- Another advantage to this paradigm is that it often **plays a part in finding other efficient algorithms**, and in fact it was the central role in finding the quick sort and merge sort algorithms.

Disadvantages

- One of the most common issues with this sort of algorithm is the fact that the **recursion is slow**, which in some cases outweighs any advantages of this divide and conquer process.
- Another concern with it is the fact that sometimes it can become more **complicated than a basic iterative approach**, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.
- Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times. It is solved again. In cases like these, it can often be easier to identify and save the solution to the repeated sub problem, which is commonly referred to as memorization.

9. Decrease and Conquer Approach

Decrease-and-conquer is a general algorithm design technique, based on exploiting a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (usually recursively) or bottom up.

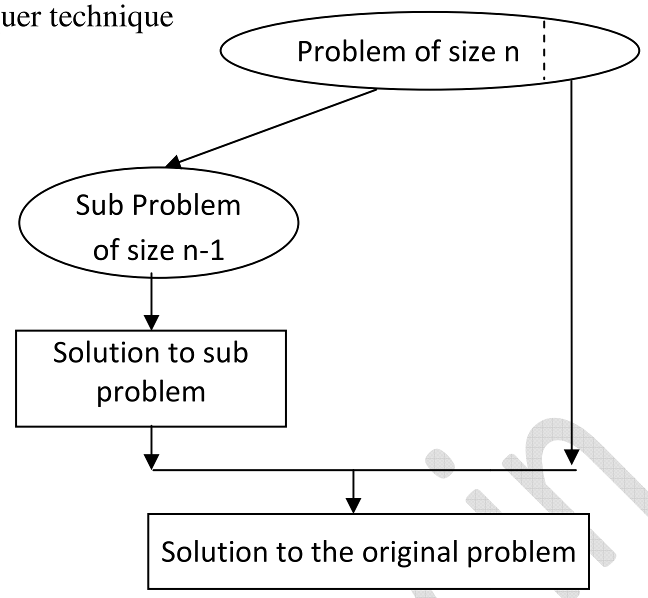
There are three major variations of decrease-and-conquer:

- decrease-by-a-constant, most often by one (e.g., insertion sort)
- decrease-by-a-constant-factor, most often by the factor of two (e.g., binary search)
- variable-size-decrease (e.g., Euclid’s algorithm)

In the **decrease-by-a-constant** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one although other constant size reductions do happen occasionally.

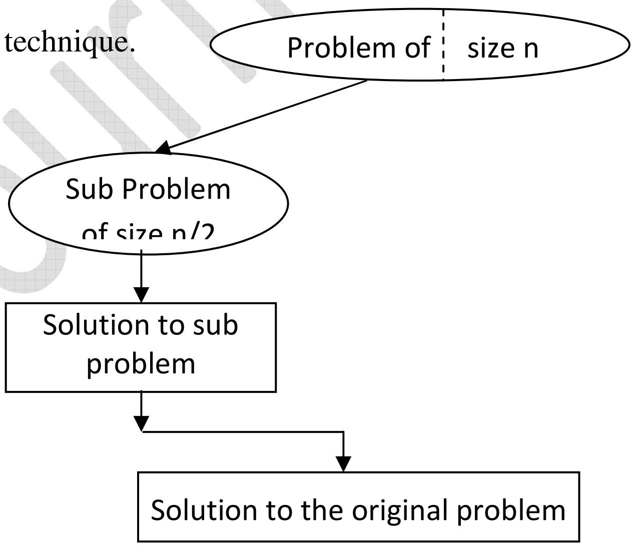
Figure: Decrease-(by one)-and-conquer technique

Example: $a^n = a^{n-1} \times a$



The **decrease-by-a-constant-factor** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.

Figure: Decrease-(by half)-and-conquer technique.



Example:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

Finally, in the **variable-size-decrease** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another.

Example: Euclid's algorithm for computing the greatest common divisor. It is based on the formula. *gcd(m, n) = gcd(n, m mod n).*

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

10. Topological Sort

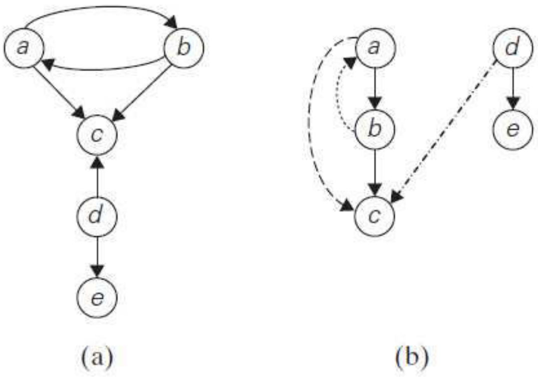
Background

A directed graph, or digraph for short, is a graph with directions specified for all its edges. The adjacency matrix and adjacency lists are the two principal means of representing a digraph.

There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure, the depth-first search forest (Figure b) exhibits all four types of edges possible in a DFS forest of a directed graph:

- ***tree edges*** (*ab*, *bc, de*),
- ***back edges*** (*ba)* from vertices to their ancestors,
- ***forward edges*** (*ac)* from vertices to their descendants in the tree other than their children, and
- ***cross edges*** (*dc*), which are none of the aforementioned types.



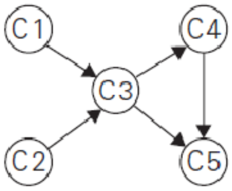
(a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at *a*.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, *a, b, a* is a directed cycle in the digraph in Figure given above. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

Motivation for topological sorting

Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements.



In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. In other words, can you find such an ordering of this digraph's vertices? This problem is called **topological sorting**.

Topological Sort

For topological sorting to be possible, a digraph in question must be a dag. i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution.

There are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem. The first one is based on depth-first search; the second is based on a direct application of the decrease-by-one technique.

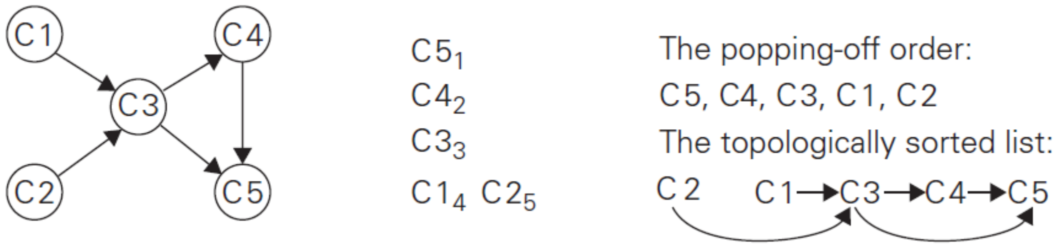
Topological Sorting based on DFS

Method

1. Perform a DFS traversal and note the order in which vertices become dead-ends
2. Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Illustration

a) Digraph for which the topological sorting problem needs to be solved.
b) DFS traversal stack with the subscript numbers indicating the popping off order.
c) Solution to the problem. Here we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.



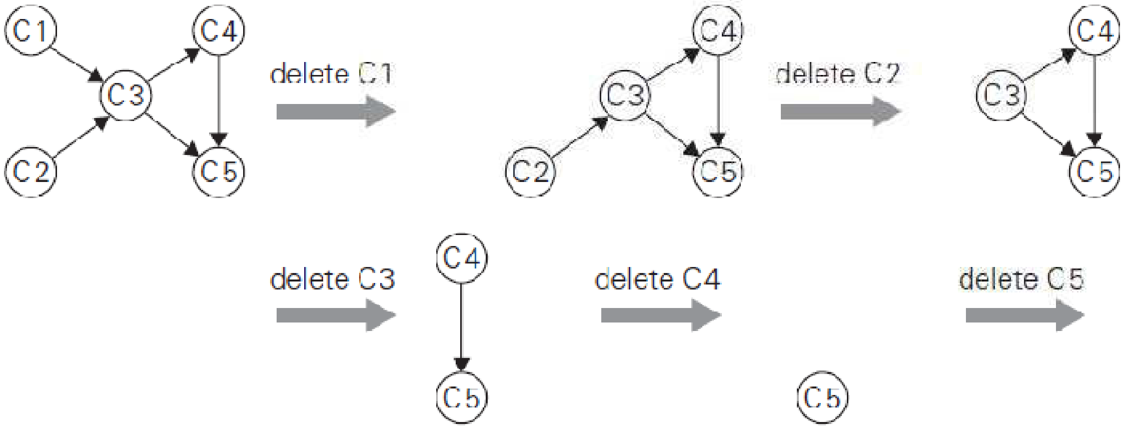
(a) (b) (c)

Topological Sorting using decrease-and-conquer technique:

Method: The algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique:

1. Repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved.)
2. The order in which the vertices are deleted yields a solution to the topological sorting problem.

Illustration - Illustration of the source-removal algorithm for the topological sorting problem is given here. On each iteration, a vertex with no incoming edges is deleted from the digraph.



The solution obtained is C1, C2, C3, C4, C5

Note: The solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several **alternative solutions**.

Applications of Topological Sorting

- Instruction scheduling in program compilation
- Cell evaluation ordering in spreadsheet formulas,
- Resolving symbol dependencies in linkers.

***

Vivekananda
College of Engineering & Technology
Nehru Nagar Post, Puttur, D.K. 574203

Lecture Notes on

15CS43

Design and Analysis of Algorithms

(CBCS Scheme)

Prepared by

Mr. Harivinod N
Dept. of Computer Science and Engineering,
VCET Puttur

Mar 2017

Module-3 : Greedy Method

Contents

1. Introduction to Greedy method

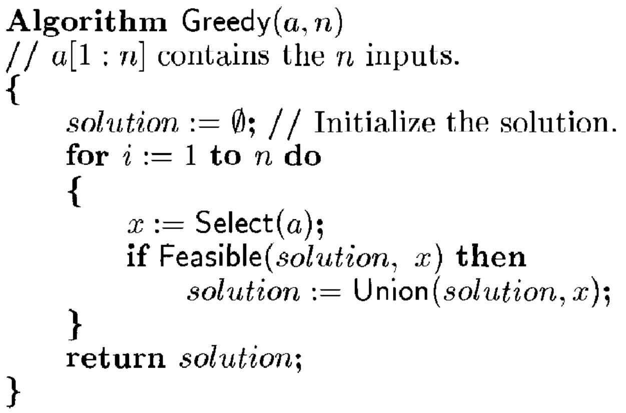
1.1 General method

The greedy method is the straight forward design technique applicable to variety of applications.

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step the choice made must be:

- ***feasible***, i.e., it has to satisfy the problem's constraints
- ***locally optimal***, i.e., it has to be the best local choice among all feasible choices available on that step
- ***irrevocable***, i.e., once made, it cannot be changed on subsequent steps of the algorithm

As a rule, greedy algorithms are both intuitively appealing and simple. Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem. What is usually more difficult is to prove that a greedy algorithm yields an optimal solution (when it does).



```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Greedy method control abstraction for the subset paradigm

1.2. Coin Change Problem

Problem Statement: Given coins of several denominations find out a way to give a customer an amount with **fewest** number of coins.

Example: if denominations are 1, 5, 10, 25 and 100 and the change required is 30, the solutions are,

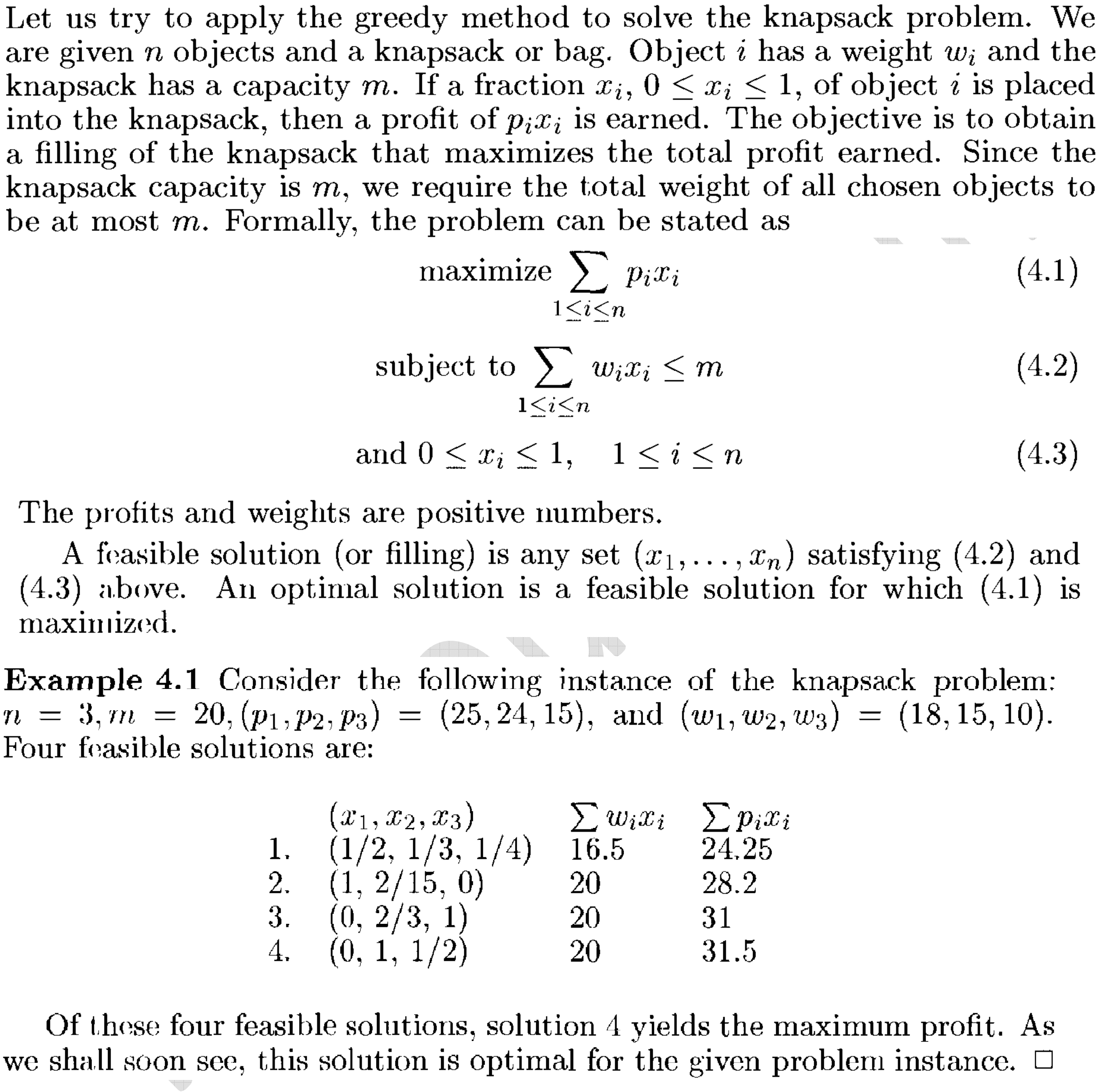
Amount : 30

Solutions : 3 x 10 (3 coins), 6 x 5 (6 coins)
1 x 25 + 5 x 1 (6 coins) **1 x 25 + 1 x 5 (2 coins)**

The last solution is the **optimal** one as it gives us change only with 2 coins.

Solution for coin change problem using greedy algorithm is very intuitive and called as cashier's algorithm. Basic principle is: **At every iteration for search of a coin, take the largest coin which can fit into remain amount to be changed at that particular time.** At the end you will have optimal solution.

1.3. Knapsack Problem



Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m \tag{4.2}$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

There are several greedy methods to obtain the feasible solutions.

a) At each step fill the knapsack with the object with **largest profit -** If the object under consideration does not fit, then the fraction of it is included to fill the knapsack. This method does not result optimal solution. As per this method the solution to the above problem is as follows;

Select Item-1 with profit p_1=25, here w_1=18, x_1=1. Remaining capacity = 20-18 = 2

Select Item-2 with profit p_1=24, here w_2=15, x_1=2/15. Remaining capacity = 0

Total profit earned = 28.2. This results 2nd solution in the example 4.1

b) At each step fill the object with **smallest weight**

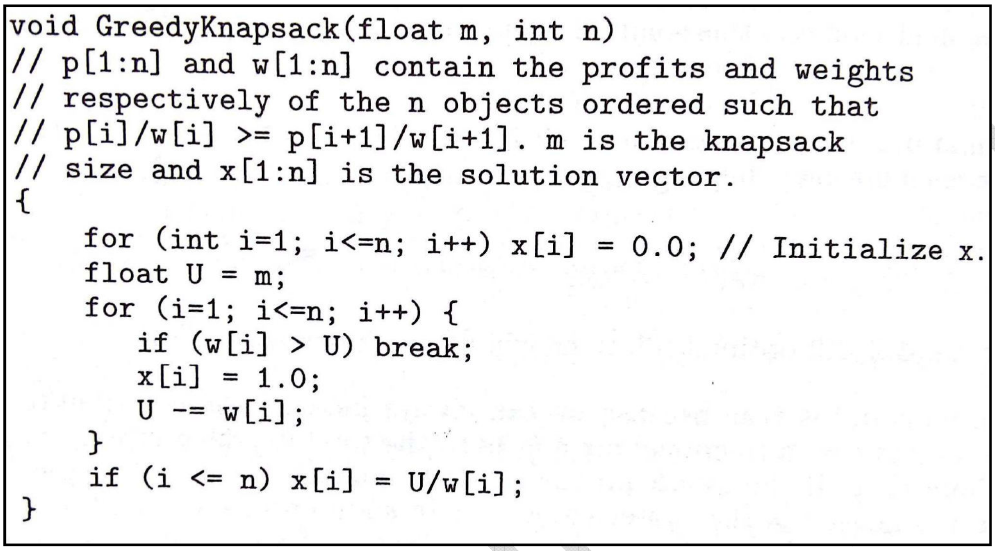
This results 3rd solution in the example 4.1

c) At each step include the object with **maximum profit/weight ratio**

This results 4th solution in the example 4.1

This greedy approach always results ***optimal solution***.

Algorithm: The algorithm given below assumes that the objects are sorted in non-increasing order of profit/weight ratio

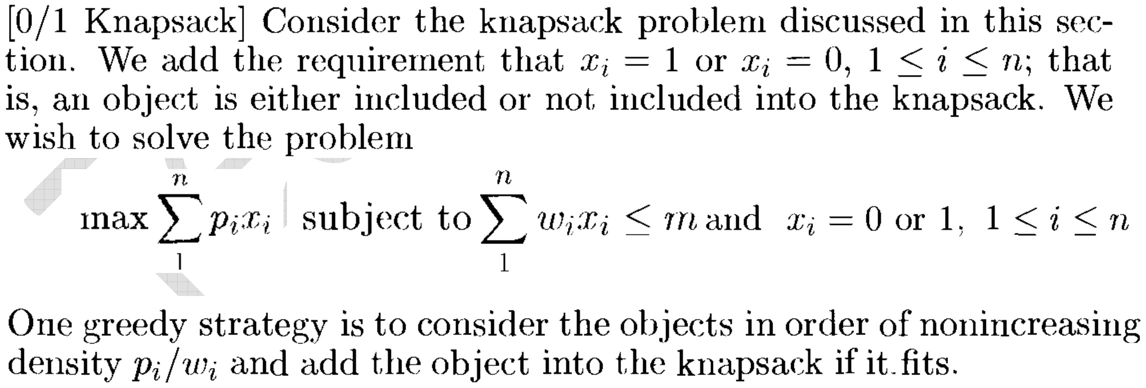


```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
     for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
     float U = m;
     for (i=1; i<=n; i++) {
        if (w[i] > U) break;
        x[i] = 1.0;
        U -= w[i];
     }
     if (i <= n) x[i] = U/w[i];
}
```

Analysis:

Disregarding the time to initially sort the object, each of the above strategies use O(n) time,

<u>0/1 Knapsack problem</u>



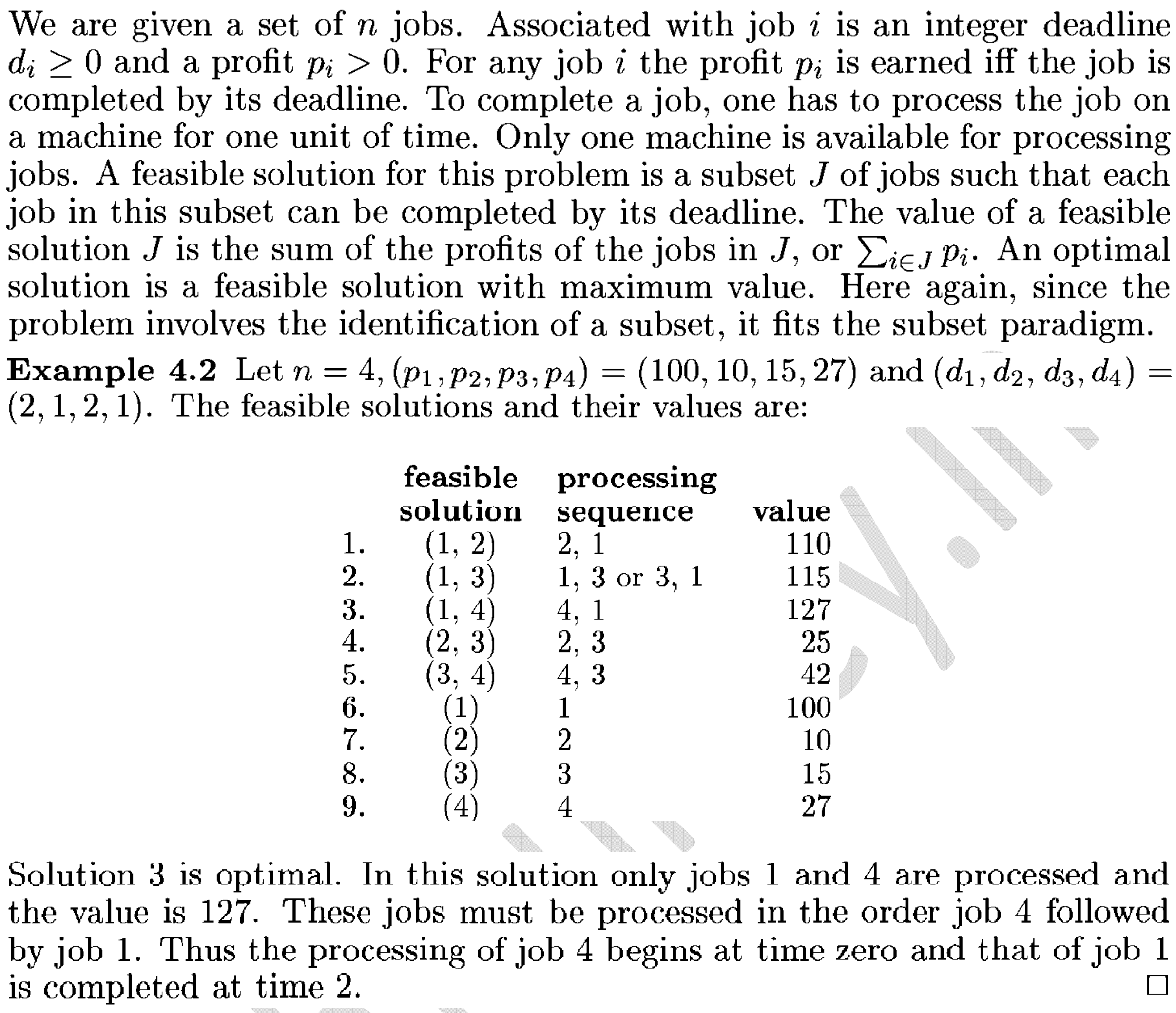
[0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \le i \le n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i \quad \text{subject to} \sum_{1}^{n} w_i x_i \le m \text{ and } \quad x_i = 0 \text{ or } 1, \; 1 \le i \le n$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits.

Note: The greedy approach to solve this problem does not necessarily yield an optimal solution

1.4. Job sequencing with deadlines



We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

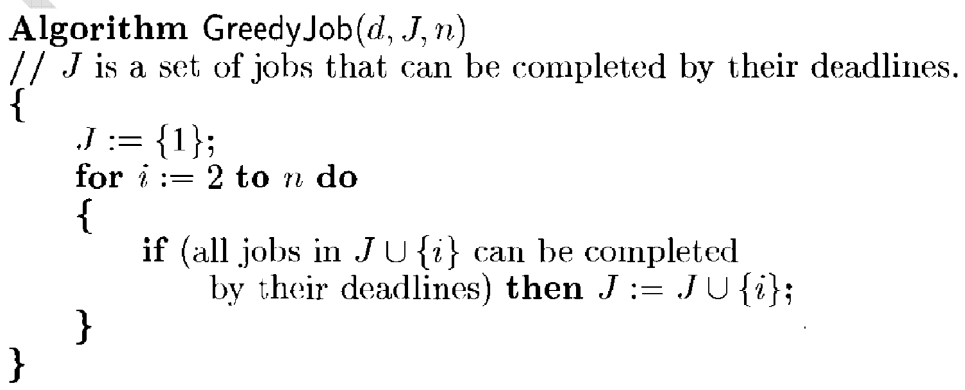
Example 4.2 Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	**processing sequence**	**value**
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

The greedy strategy to solve job sequencing problem is, "At each time select the job that that satisfies the constraints and gives **maximum profit.** i.e consider the jobs in the non decreasing order of the p_i's"

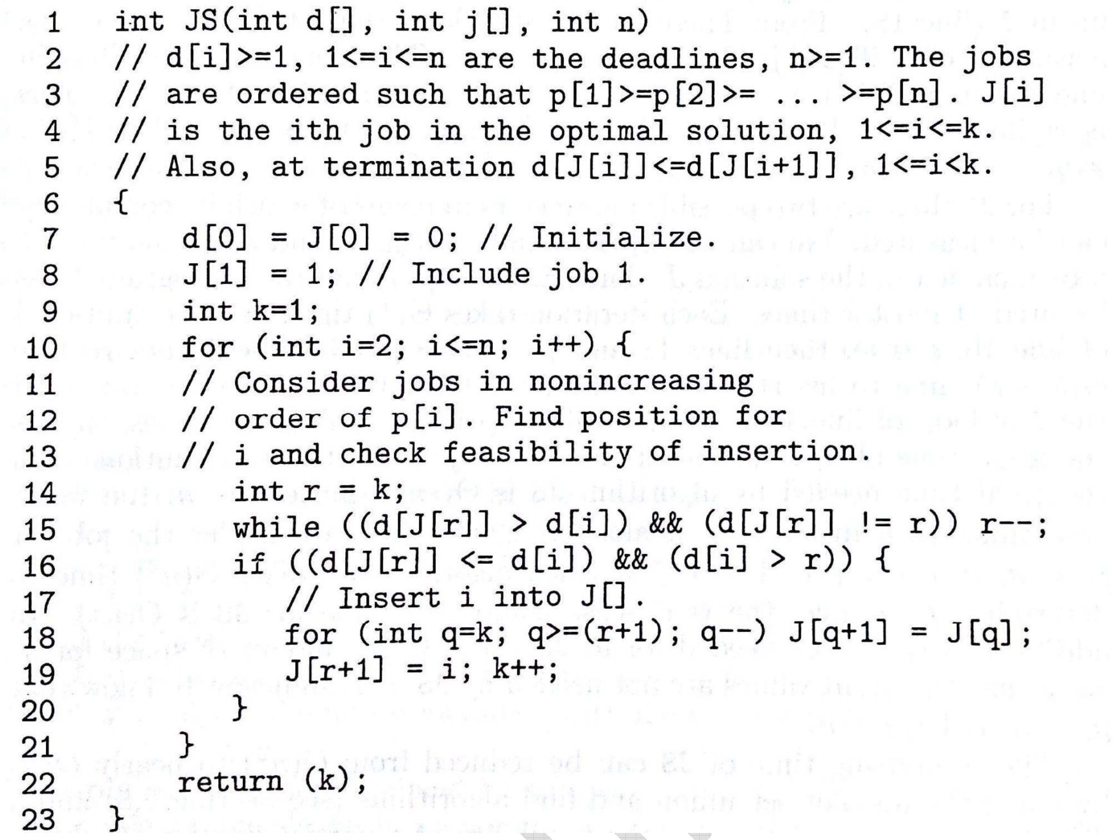
By following this procedure, we get the 3rd solution in the example 4.3. It can be proved that, this greedy strategy always results optimal solution



```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J := J ∪ {i};
    }
}
```

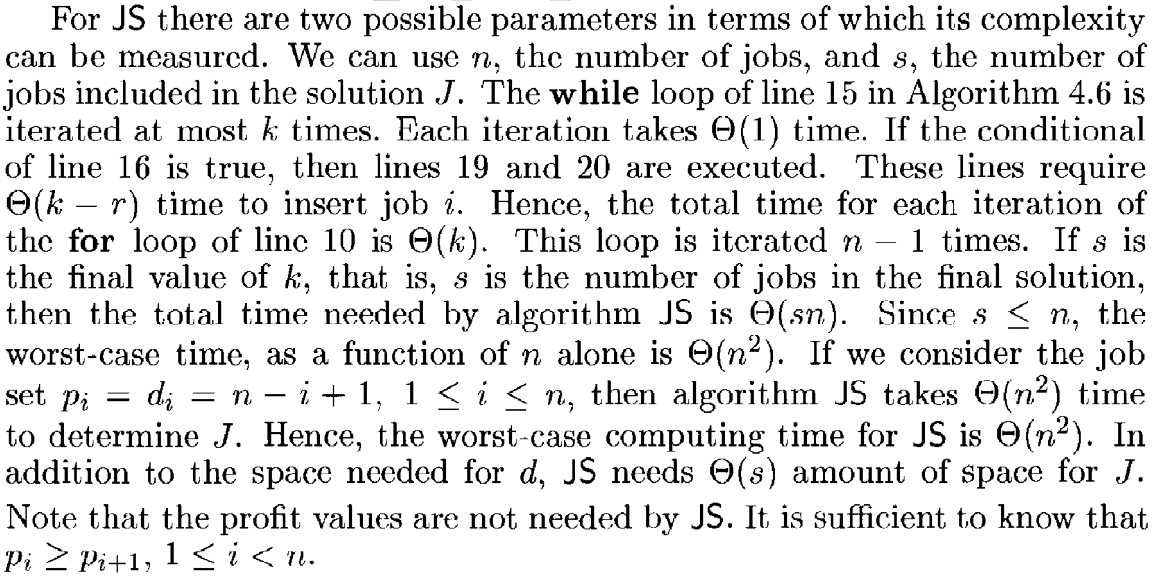
High level description of job sequencing algorithm

Algorithm/Program 4.6: Greedy algorithm for sequencing unit time jobs with deadlines and profits



```
 1   int JS(int d[], int j[], int n)
 2   // d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs
 3   // are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i]
 4   // is the ith job in the optimal solution, 1<=i<=k.
 5   // Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
 6   {
 7        d[0] = J[0] = 0; // Initialize.
 8        J[1] = 1; // Include job 1.
 9        int k=1;
10        for (int i=2; i<=n; i++) {
11        // Consider jobs in nonincreasing
12        // order of p[i]. Find position for
13        // i and check feasibility of insertion.
14           int r = k;
15           while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
16           if ((d[J[r]] <= d[i]) && (d[i] > r)) {
17              // Insert i into J[].
18              for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
19              J[r+1] = i; k++;
20           }
21        }
22        return (k);
23   }
```

Analysis:



For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k-r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n-1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J. Note that the profit values are not needed by JS. It is sufficient to know that $p_i \geq p_{i+1}$, $1 \leq i < n$.

Fast Job Scheduling Algorithm

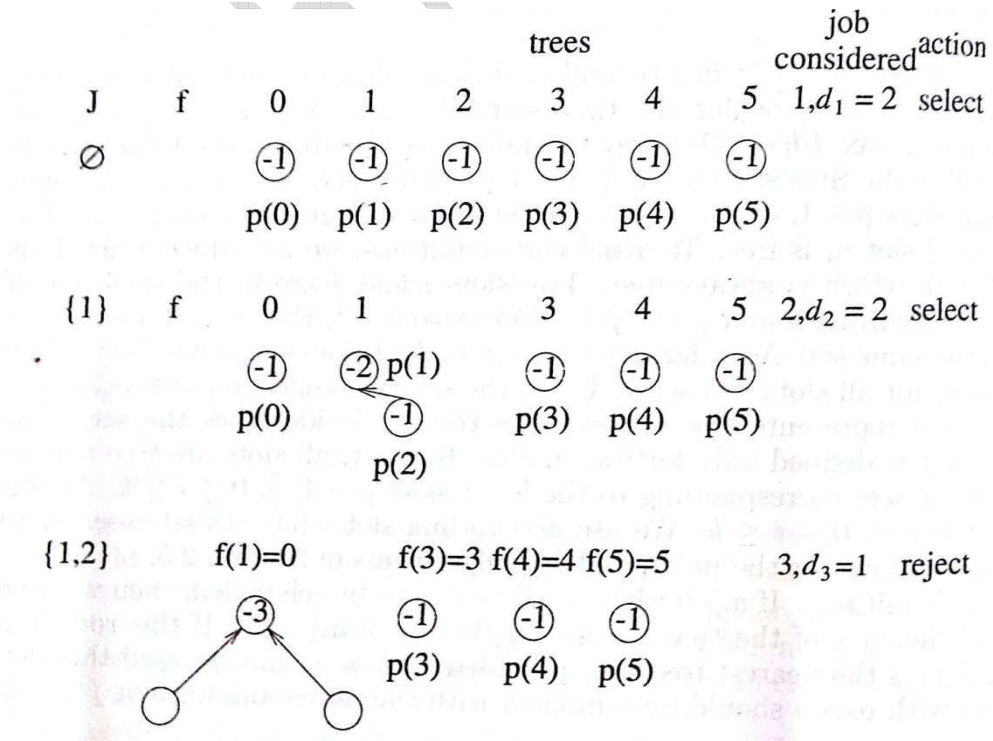
The computing time of JS can be reduced from $O(n^2)$ to nearly $O(n)$ by using the disjoint set union and find algorithms and a different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs, then we can determine the processing times for each of the jobs using the rule: if job i hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the largest integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined above, then it cannot be included in J.

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Example 4.4 The trees defined by the $p(i)$'s for the first three iterations in Example 4.3 are shown in Figure 4.4. □



Algorithm: Fast Job Sheduling

```
Algorithm FJS(d, n, b, j)
// Find an optimal solution J[1 : k]. It is assumed that
// p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
{
    // Initially there are b + 1 single node trees.
    for i := 0 to b do f[i] := i;
    k := 0; // Initialize.
    for i := 1 to n do
    { // Use greedy rule.
        q := CollapsingFind(min(n, d[i]));
        if (f[q] ≠ 0) then
        {
            k := k + 1; J[k] := i; // Select job i.
            m := CollapsingFind(f[q] − 1);
            WeightedUnion(m, q);
            f[q] := f[m]; // q may be new root.
        }
    }
}
```

Analysis

The fast algorithm appears as FJS (Algorithm 4.7). Its computing time is readily observed to be $O(n\alpha(2n, n))$ (recall that $\alpha(2n, n)$ is the inverse of Ackermann's function defined in Section 2.5). It needs an additional $2n$ words of space for f and p.

2. Minimum cost spanning trees

Definition: A **spanning tree** of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. A **minimum spanning tree** of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the **weights** on all its edges. The **minimum spanning tree problem** is the problem of finding a minimum spanning tree for a given weighted connected graph.

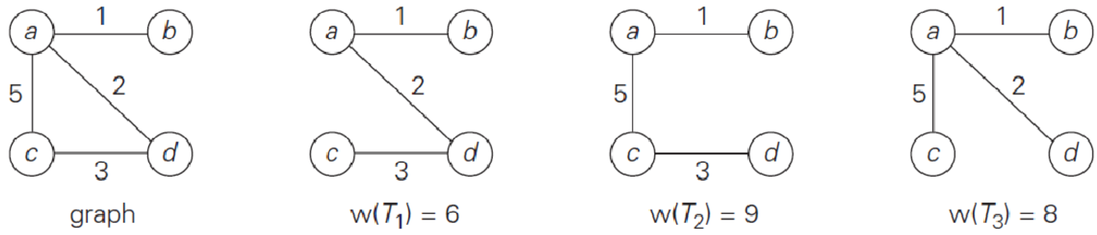


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

2.1. Prim's Algorithm

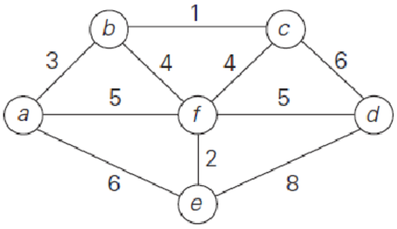
Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub-trees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration it expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n - 1, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges.

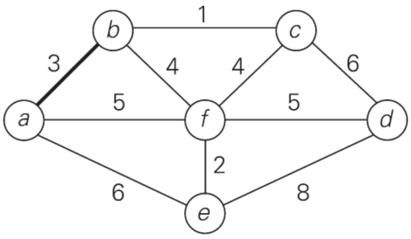
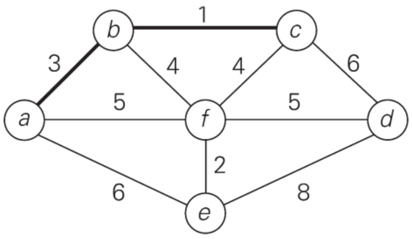
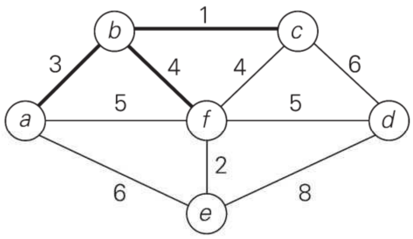
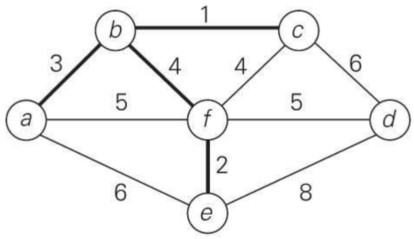
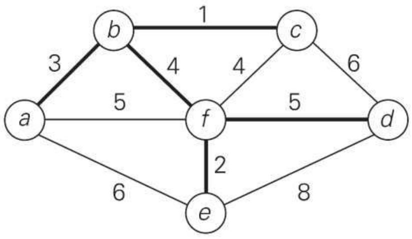
```
ALGORITHM Prim(G)
    //Prim's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    $V_T \leftarrow \{v_0\}$    //the set of tree vertices can be initialized with any vertex
    $E_T \leftarrow \varnothing$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
        such that $v$ is in $V_T$ and $u$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

Correctness

Prim's algorithm always yields a minimum spanning tree.

Example: An example of prim's algorithm is shown below. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.



Tree vertices	*Remaining vertices*	*Illustration*
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Analysis of Efficiency

The efficiency of Prim's algorithm depends on the data structures chosen for the **graph** itself and for the **priority queue** of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

1. If a graph is represented by its **weight matrix** and the priority queue is implemented as an **unordered array**, the algorithm's running time will be in **$\Theta(|V|^2)$**. Indeed, on each of the $|V| - 1$iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

We can implement the priority queue as a **min-heap**. (A min-heap is a complete binary tree in which every element is less than or equal to its children.) Deletion of the smallest element from and insertion of a new element into a min-heap of size n are **O(log n)** operations.

2. If a graph is represented by its **adjacency lists** and the priority queue is implemented as a **min-heap**, the running time of the algorithm is in **$O(|E| \log |V|)$.**

This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation. Hence, the running time of this implementation of Prim's algorithm is in

$(|V| - 1+ |E|)\ O(\log |V|) = O(|E| \log |V|)$ because, in a connected graph, $|V| - 1 \leq |E|$.

2.2. Kruskal's Algorithm

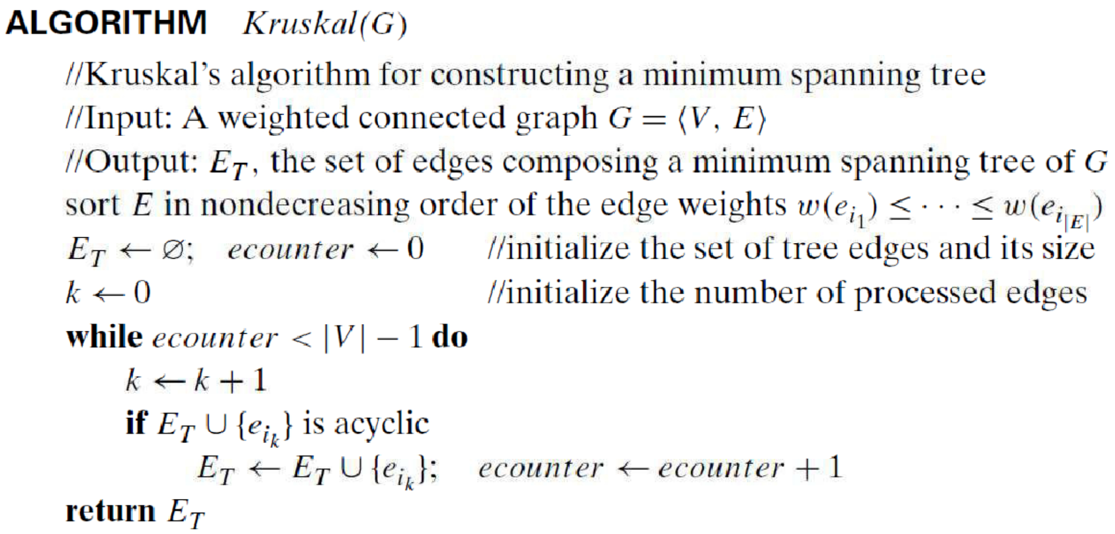
Background

Kruskal's algorithm is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named Kruskal's algorithm, after Joseph Kruskal.

Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G = (V, E) as an acyclic sub graph with |V | - 1 edges for which the **sum of the edge weights is the smallest**. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of sub graphs, which are always **acyclic** but are not necessarily connected on the intermediate stages of the algorithm.

Working

The algorithm begins by **sorting** the graph's edges in **non decreasing** order of their **weights**. Then, starting with the empty sub graph, it scans this sorted list adding the next edge on the list to the current sub graph if such an inclusion **does not create a cycle** and simply **skipping the edge otherwise.**



```
ALGORITHM  Kruskal(G)
    //Kruskal's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
    $E_T \leftarrow \varnothing$;   $ecounter \leftarrow 0$     //initialize the set of tree edges and its size
    $k \leftarrow 0$                         //initialize the number of processed edges
    while $ecounter < |V| - 1$ do
        $k \leftarrow k + 1$
        if $E_T \cup \{e_{i_k}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{i_k}\}$;   $ecounter \leftarrow ecounter + 1$
    return $E_T$
```

The fact that E_T ,the set of edges composing a minimum spanning tree of graph G actually a tree in Prim's algorithm but generally just an acyclic sub graph in Kruskal's algorithm.

Kruskal's algorithm is **not simpler** because it has to check whether the addition of the next edge to the edges already selected would **create a cycle**.

We can consider the algorithm's operations as a progression through **a series of forests** containing all the vertices of a given graph and some of its edges. The initial forest consists of |V| trivial trees, each comprising a **single vertex of the graph**. The **final forest** consists of a **single tree,** which is a **minimum spanning tree** of the graph**.** On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

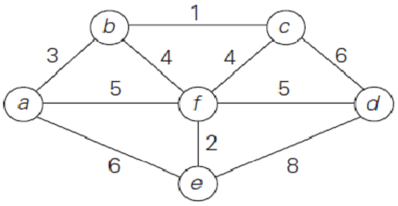
Analysis of Efficiency

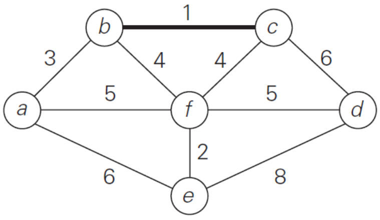
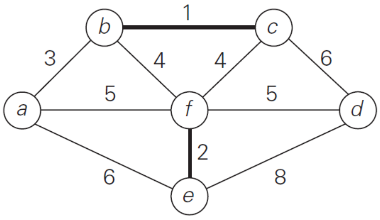
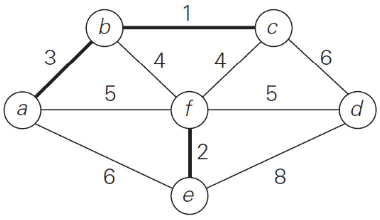
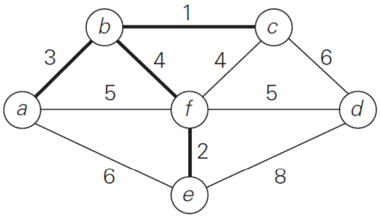
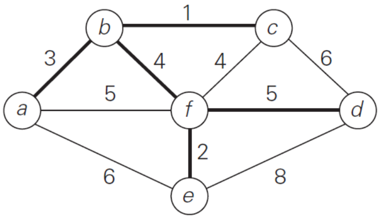
The crucial check whether two vertices belong to the same tree can be found out using **union-find algorithms.**

Efficiency of Kruskal's algorithm is based on the time needed for **sorting the edge weights** of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in **O (|E| log |E|).**

Illustration

An example of Kruskal's algorithm is shown below. The selected edges are shown in bold.



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

3. Single source shortest paths

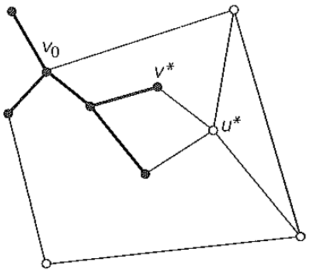
Single-source shortest-paths problem is defined as follows. For a given vertex called the *source* in a weighted connected graph, the problem is to find shortest paths to all its other vertices. The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

3.1. Dijkstra's Algorithm

Dijkstra's Algorithm is the best-known algorithm for the single-source shortest-paths problem. This algorithm is applicable to undirected and directed graphs with nonnegative weights only.

Working - Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source.

- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.
- In general, before its i^{th} iteration commences, the algorithm has already identified the shortest paths to *i*-1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph shown in the figure.

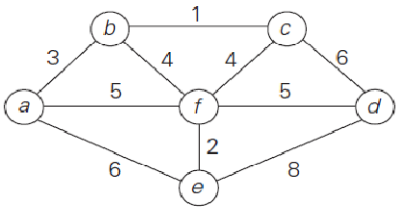


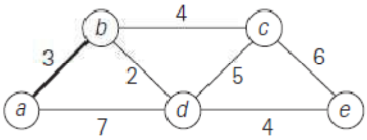
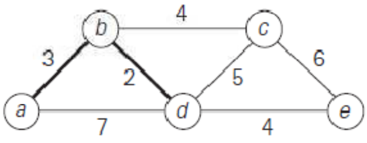
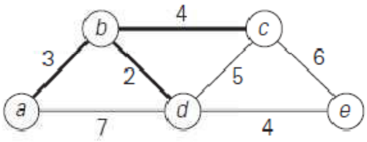
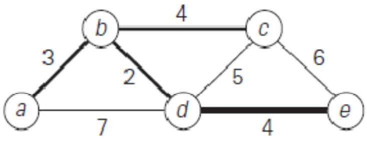
- Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i. The set of vertices adjacent to the vertices in *Ti* can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.
- To identify the i^{th} nearest vertex, the algorithm computes, for every fringe vertex *u,* the sum of the distance to the nearest tree vertex *v* (given by the weight of the edge (*v,* u)) and the length *d.,* of the shortest path from the source to *v* (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.
- To facilitate the algorithm's operations, we label each vertex with two labels.
  - The numeric label **d** indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, d indicates the length of the shortest path from the source to that vertex.
  - The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the source s and vertices that are adjacent to none of the current tree vertices.)

With such labeling, finding the next nearest vertex u* becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily.

- After we have identified a vertex u* to be added to the tree, we need to perform two operations:
    - Move $u*$ from the fringe to the set of tree vertices.
    - For each remaining fringe vertex u that is connected to $u*$ by an edge of weight $w\ (u*, u)$ such that $d\ u* + w(u*, u) < d\ u$, update the labels of u by $u*$ and $du* + w(u*, u)$, respectively.
    -

Illustration: An example of Dijkstra's algorithm is shown below. The next closest vertex is shown in bold.

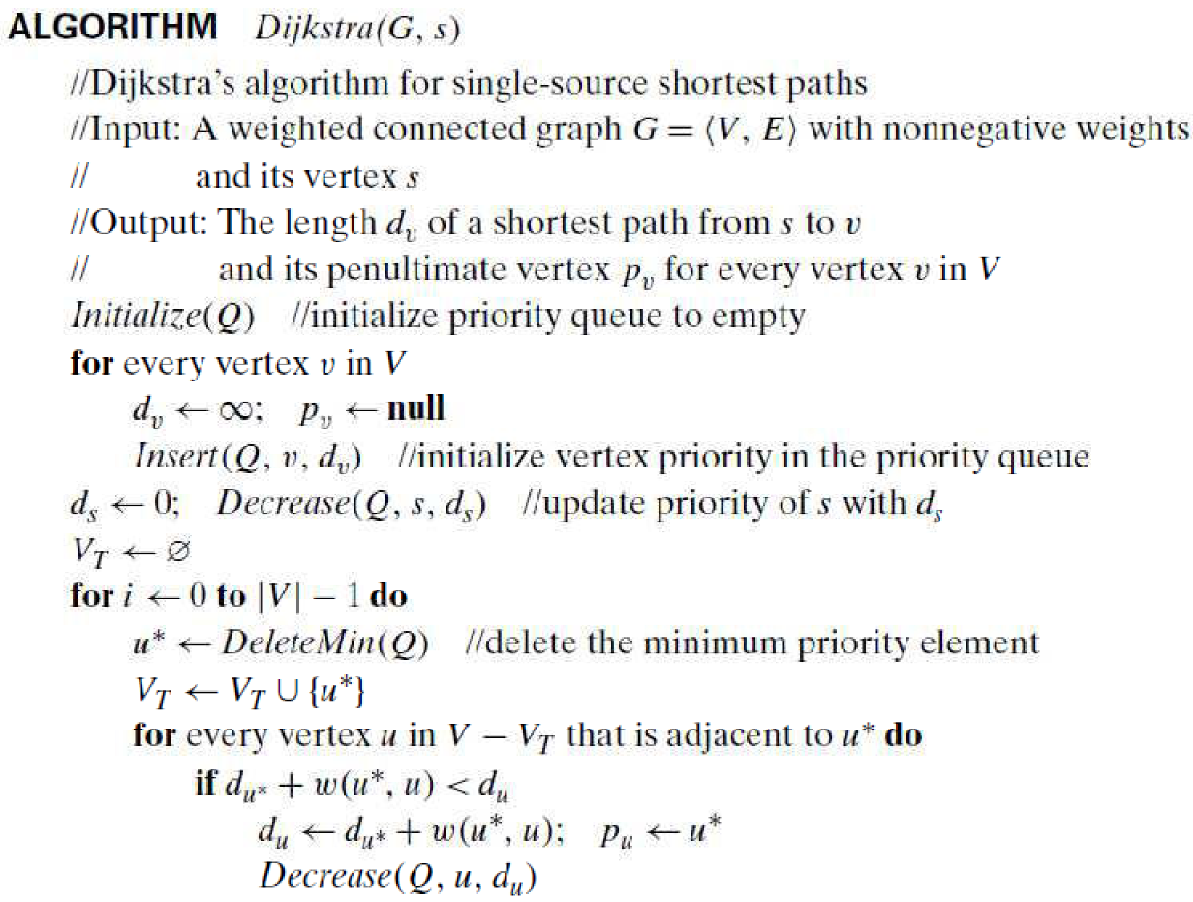


Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9

The pseudocode of Dijkstra's algorithm is given below. Note that in the following pseudocode, V_T contains a given source vertex and the fringe contains the vertices adjacent to it *after* iteration 0 is completed.



```
ALGORITHM  Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
    //          and its vertex $s$
    //Output: The length $d_v$ of a shortest path from $s$ to $v$
    //            and its penultimate vertex $p_v$ for every vertex $v$ in $V$
    Initialize(Q)   //initialize priority queue to empty
    for every vertex $v$ in $V$
        $d_v \leftarrow \infty$;   $p_v \leftarrow$ null
        Insert(Q, v, $d_v$)   //initialize vertex priority in the priority queue
    $d_s \leftarrow 0$;   Decrease(Q, s, $d_s$)   //update priority of $s$ with $d_s$
    $V_T \leftarrow \varnothing$
    for $i \leftarrow 0$ to $|V| - 1$ do
        $u^* \leftarrow$ DeleteMin(Q)   //delete the minimum priority element
        $V_T \leftarrow V_T \cup \{u^*\}$
        for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
            if $d_{u^*} + w(u^*, u) < d_u$
                $d_u \leftarrow d_{u^*} + w(u^*, u)$;   $p_u \leftarrow u^*$
                Decrease(Q, u, $d_u$)
```

Analysis:

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in **O (|E| log |V|)**

Applications

- Transportation planning and packet routing in communication networks, including the Internet
- Finding shortest paths in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling.

4. Optimal Tree problem

Background

Suppose we have to encode a text that comprises characters from some n-character alphabet by assigning to each of the text's characters some sequence of bits called the ***codeword.***There are two types of encoding: Fixed-length encoding, Variable-length encoding

Fixed-length encoding: This method assigns to each character a bit string of the same length *m* (m >= $\log_2 n$). This is exactly what the standard ASCII code does. One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter code-words to more frequent characters and longer code-words to less frequent characters.

Variable-length encoding: This method assigns code-words of different lengths to different characters, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the i^{th}) character? To avoid this complication, we can limit ourselves to *prefix-free* (or simply *prefix) codes*. In a prefix code, no codeword is a prefix of a codeword of another character. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some character, replace these bits by this character, and repeat this operation until the bit string's end is reached.

If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's characters with leaves of a binary tree in which all the left edges are labelled by 0 and all the right edges are labelled by 1 (or vice versa). The codeword of a character can then be obtained by recording the labels on the simple path from the root to the character's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; hence, any such tree yields a prefix code.

Among the many trees that can be constructed in this manner for a given **alphabet with known frequencies of the character occurrences, con**struction of such a tree that would assign shorter bit strings to high-frequency characters and longer ones to low-frequency characters can be done by the following greedy algorithm, invented by **David Huffman.**

4.1 Huffman Trees and Codes

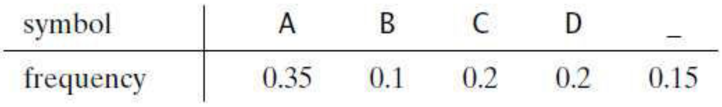
Huffman's Algorithm

Step 1*:* Initialize *n* one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's *weight.* (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2*:* Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

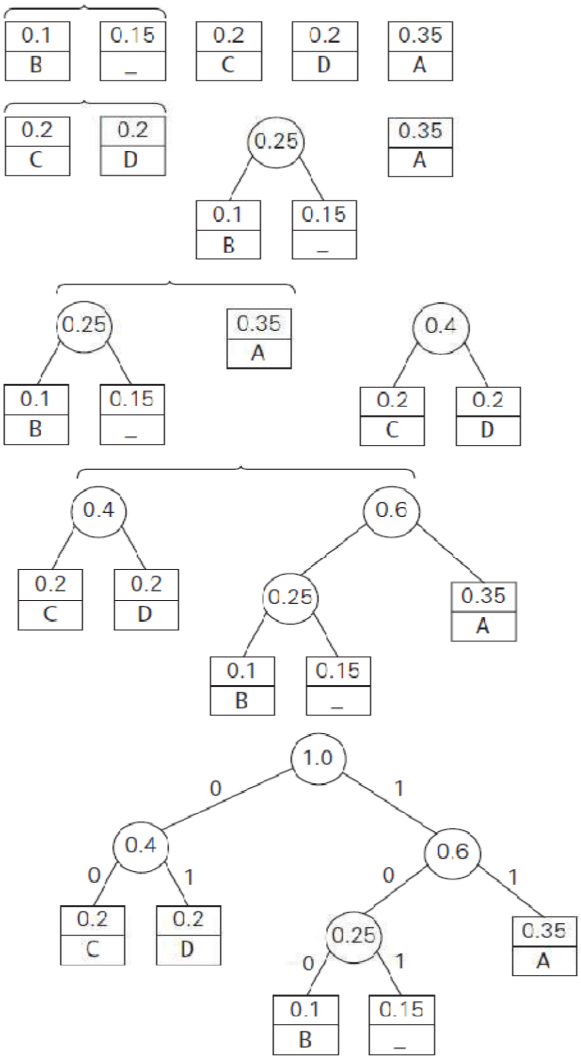
A tree constructed by the above algorithm is called **a *Huffman tree***. It defines-in the manner described-a ***Huffman code***.

Example: Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

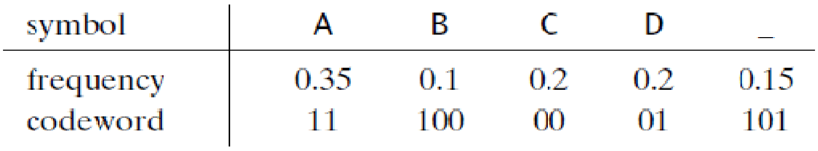


symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for the above problem is shown below:



The resulting codewords are as follows:



symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.

With the occurrence frequencies given and the codeword lengths obtained, the **average number of bits per symbol in** this code is

$$2 * 0.35 + 3 * 0.1 + 2 * 0.2 + 2 * 0.2 + 3 * 0.15 = 2.25.$$

Had we used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this example, Huffman's code achieves the ***compression ratio*** (a standard measure of a compression algorithm's effectiveness) of (3−2.25)/3*100%= 25%. In other words, Huffman's encoding of the above text will use 25% less memory than its fixed-length encoding.

5. Transform and Conquer Approach

5.1. Heaps

Heap is a partially ordered data structure that is especially suitable for implementing priority queues. **Priority queue** is a multiset of items with an orderable characteristic called an item's **priority**, with the following operations:

- finding an item with the highest (i.e., largest) priority
- deleting an item with the highest priority
- adding a new item to the multiset

Notion of the Heap

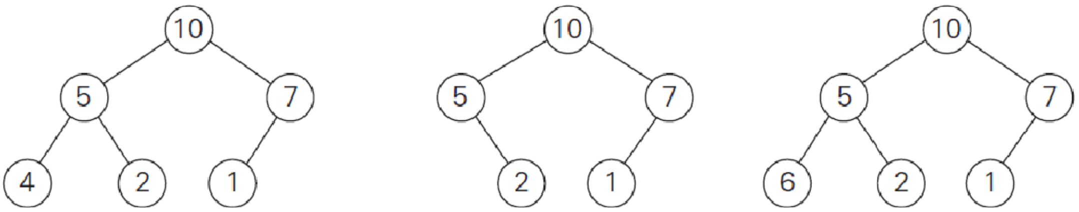
Definition:

A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

1. The **shape property**—the binary tree is **essentially complete** (or simply **complete**), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The **parental dominance** or **heap property**—the key in each node is greater than or equal to the keys in its children.

Illustration:

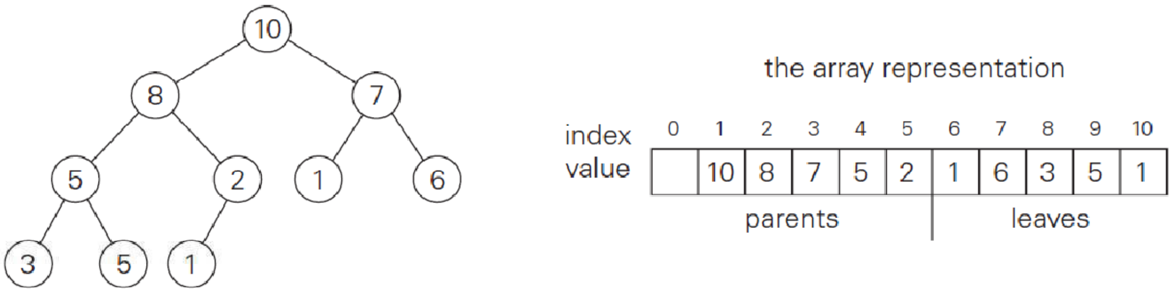
The illustration of the definition of heap is shown bellow: only the left most tree is heap. The second one is not a heap, because the tree's shape property is violated. The left child of last subtree cannot be empty. And the third one is not a heap, because the parental dominance fails for the node with key 5.



Properties of Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor log_2 n \rfloor$
2. The root of a heap always contains its largest element.

3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an **array** by recording its elements in the top down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
    a. the parental node keys will be in the first $\lfloor n/2 \rfloor$. positions of the array, while the leaf keys will occupy the last $\lfloor n/2 \rfloor$ positions;
    b. the children of a key in the array's parental position i ($1 \le i \le \lfloor n/2 \rfloor$) will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor n/2 \rfloor$.



index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

Heap and its array representation

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \ge \max \{H[2i], H[2i + 1]\} \text{ for } i = 1 . . . \lfloor n/2 \rfloor$$

Constructions of Heap - There are two principal alternatives for constructing Heap.
1) Bottom-up heap construction 2) Top-down heap construction

Bottom-up heap construction:

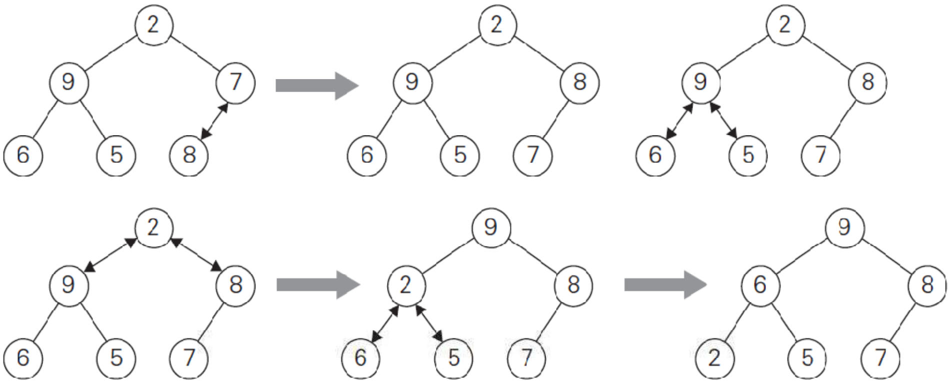
The bottom-up heap construction algorithm is illustrated bellow. It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows.

- Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key in this node. If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance for K is satisfied. (Eventually, it has to because it holds automatically for any key in a leaf.)
- After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor.
- The algorithm stops after this is done for the root of the tree.

```
ALGORITHM  HeapBottomUp(H[1..n])
    //Constructs a heap from elements of a given array
    // by the bottom-up algorithm
    //Input: An array H[1..n] of orderable items
    //Output: A heap H[1..n]
    for i ← ⌊n/2⌋ downto 1 do
        k ← i;   v ← H[k]
        heap ← false
        while not heap and 2 ∗ k ≤ n do
            j ← 2 ∗ k
            if j < n   //there are two children
                if H[j] < H[j + 1]  j ← j + 1
            if v ≥ H[j]
                heap ← true
            else H[k] ← H[j];   k ← j
        H[k] ← v
```

Illustration

Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double headed arrows show key comparisons verifying the parental dominance.



Analysis of efficiency - bottom up heap construction algorithm:

Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of nodes occurs on each level. Let h be the height of the tree.

According to the first property of heaps in the list at the beginning of the section, $h = \lfloor log_2 n \rfloor$ or just $\lfloor log_2(n+1) \rfloor = k - 1$ for the specific values of n we are considering.

Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key on level i will be $2(h - i)$.

Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^{h} i2^i$ or by mathematical induction on *h*.

Thus, with this bottom-up algorithm, a heap of size *n* can be constructed with fewer than **2*n*** comparisons.

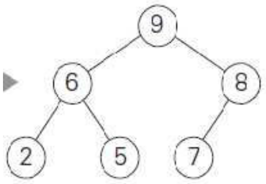
Top-down heap construction algorithm:

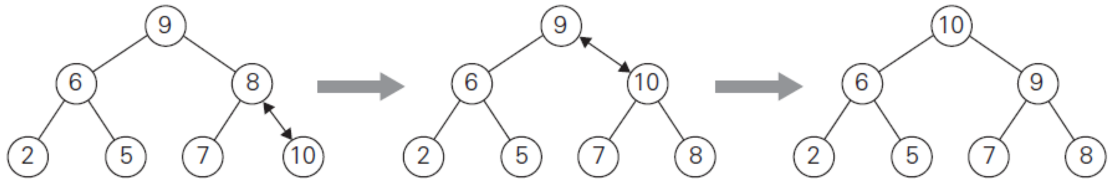
It constructs a heap by successive insertions of a new key into a previously constructed heap.

1. First, attach a new node with key *K* in it after the last leaf of the existing heap.
2. Then shift *K* up to its appropriate place in the new heap as follows.
   a. Compare *K* with its parent's key: if the latter is greater than or equal to *K*, stop (the structure is a heap); otherwise, swap these two keys and compare *K* with its new parent.
   b. This swapping continues until *K* is not greater than its last parent or it reaches root.

Obviously, this insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with *n* nodes is about $\log_2 n$, the time efficiency of insertion is in *O* (log *n*).

Illustration of inserting a new key: Inserting a new key (10) into the heap is constructed bellow. The new key is shifted up via a swap with its parents until it is not larger than its parents (or is in the root).



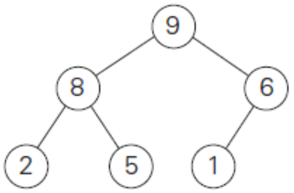


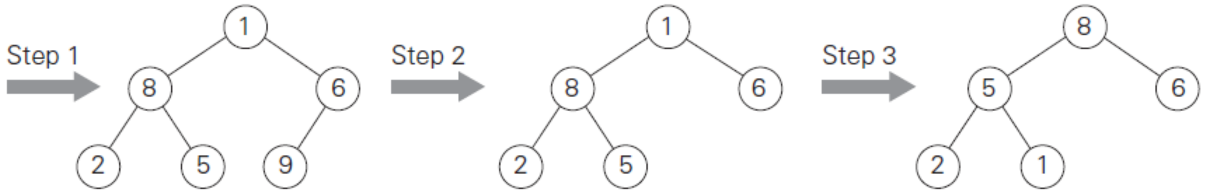
Delete an item from a heap: Deleting the root's key from a heap can be done with the following algorithm:

Maximum Key Deletion from a heap

1. Exchange the root's key with the last key *K* of the heap.
2. Decrease the heap's size by 1.
3. "Heapify" the smaller tree by sifting *K* down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for *K*: if it holds, we are done; if not, swap *K* with the larger of its children and repeat this operation until the parental dominance condition holds for *K* in its new position.

Illustration





The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1. Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in O (log n) as well.

5.2. Heap Sort

Heapsort - an interesting sorting algorithm is discovered by J. W. J. Williams. This is a two-stage algorithm that works as follows.

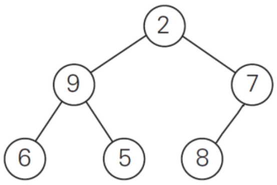
Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation n−1 times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.

Heap sort is traced on a specific input is shown below:

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2



Analysis of efficiency:

Since we already know that the heap construction stage of the algorithm is in *O(n)*, we have to investigate just the time efficiency of the second stage. For the number of key comparisons, *C(n),* needed for eliminating the root keys from the heaps of diminishing sizes from *n* to 2*,* we get the following inequality:

$$C(n) \le 2\lfloor \log_2(n-1)\rfloor + 2\lfloor \log_2(n-2)\rfloor + \cdots + 2\lfloor \log_2 1\rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$
$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n.$$

This means that C(n) ∈ O(n log n) for the second stage of heapsort.

For both stages, we get **O(n) + O(n log n) = O(n log n).**

A more detailed analysis shows that the time efficiency of heapsort is, in fact, in **Θ(n log n)** in both the **worst** and **average** cases. Thus, heapsort's time efficiency falls in the same class as that of mergesort.

Unlike the latter, heapsort is **in-place**, i.e., it does not require any extra storage. Timing experiments on random files show that heapsort runs more slowly than quicksort but can be competitive with mergesort.

*****

Vivekananda
College of Engineering & Technology
Nehru Nagar Post, Puttur, D.K. 574203

Lecture Notes on

15CS43

Design and Analysis of Algorithms

(CBCS Scheme)

Prepared by

Harivinod N
Dept. of Computer Science and Engineering,
VCET Puttur

April 2017

Module-4 : Dynamic Programming

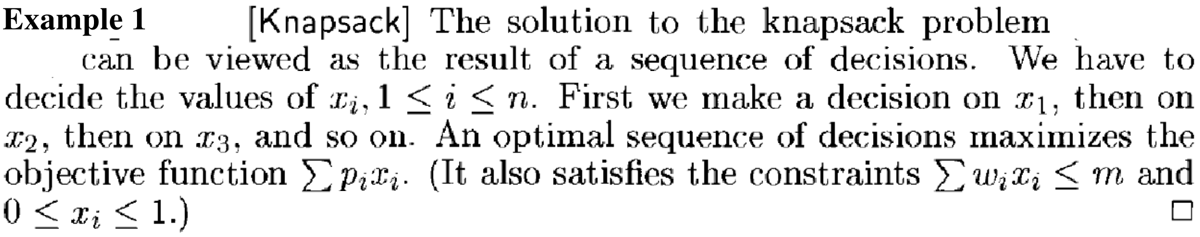
Contents

Course Website
www.TechJourney.in

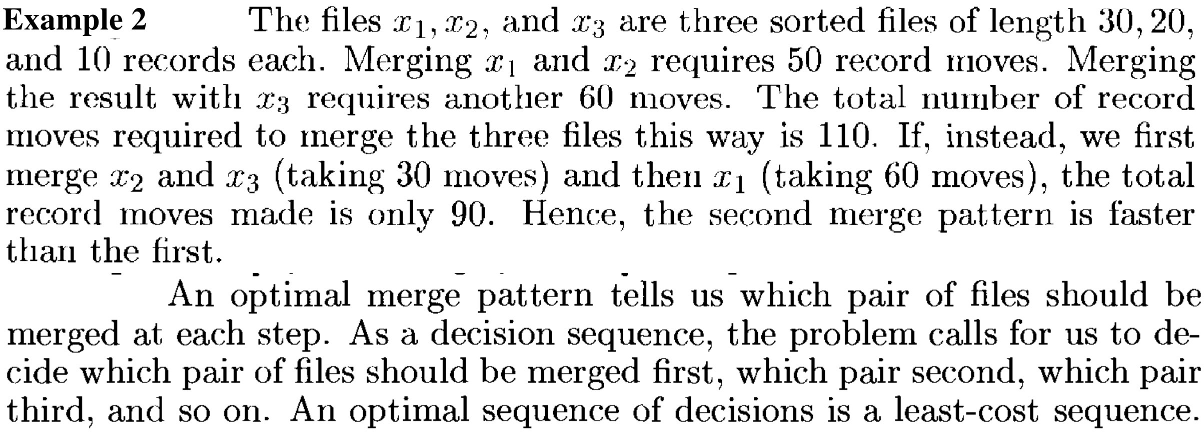
1. Introduction to Dynamic Programming

Dynamic programming is a technique for solving problems with **overlapping subproblems**. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained. *[From T1]*

The Dynamic programming can also be used when the solution to a problem can be viewed as the result of **sequence of decisions**. *[From T2]*. Here are some examples.

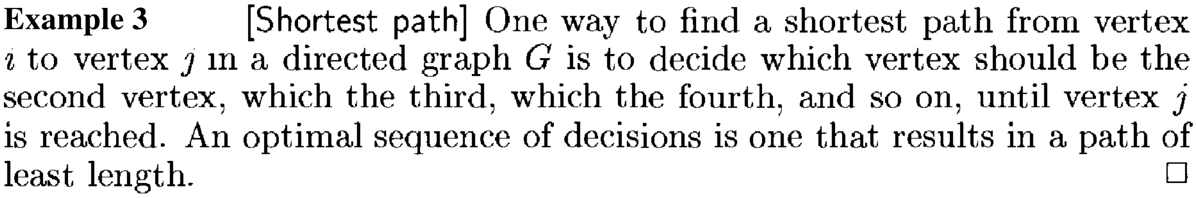


Example 1 [Knapsack] The solution to the knapsack problem can be viewed as the result of a sequence of decisions. We have to decide the values of $x_i, 1 \leq i \leq n$. First we make a decision on x_1, then on x_2, then on x_3, and so on. An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$. (It also satisfies the constraints $\sum w_i x_i \leq m$ and $0 \leq x_i \leq 1$.) □

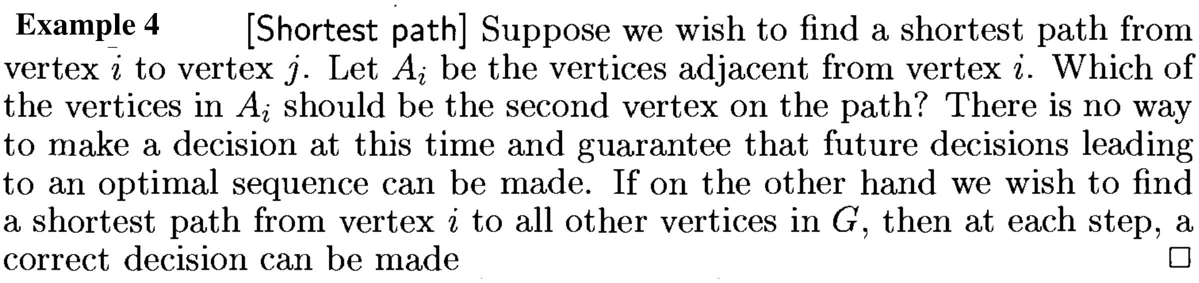


Example 2 The files x_1, x_2, and x_3 are three sorted files of length $30, 20$, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first.

An optimal merge pattern tells us which pair of files should be merged at each step. As a decision sequence, the problem calls for us to decide which pair of files should be merged first, which pair second, which pair third, and so on. An optimal sequence of decisions is a least-cost sequence.



Example 3 [Shortest path] One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth, and so on, until vertex j is reached. An optimal sequence of decisions is one that results in a path of least length. □



Example 4 [Shortest path] Suppose we wish to find a shortest path from vertex i to vertex j. Let A_i be the vertices adjacent from vertex i. Which of the vertices in A_i should be the second vertex on the path? There is no way to make a decision at this time and guarantee that future decisions leading to an optimal sequence can be made. If on the other hand we wish to find a shortest path from vertex i to all other vertices in G, then at each step, a correct decision can be made □

One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences. We could enumerate all decision sequences and then pick out the best. But the time and space requirements may be prohibitive. Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the *principle of optimality*.

Definition 5.1 [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. □

Thus, the essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal (if the principle of optimality holds) and so will not (as far as possible) be generated.

Example 5.5 [Shortest path] Consider the shortest-path problem of Example 5.3. Assume that $i, i_1, i_2, \ldots, i_k, j$ is a shortest path from i to j. Starting with the initial vertex i, a decision has been made to go to vertex i_1. Following this decision, the problem state is defined by vertex i_1 and we need to find a path from i_1 to j. It is clear that the sequence $i_1, i_2, \ldots, i_k, j$ must constitute a shortest i_1 to j path. If not, let $i_1, r_1, r_2, \ldots, r_q, j$ be a shortest i_1 to j path. Then $i, i_1, r_1, \cdots, r_q, j$ is an i to j path that is shorter than the path $i, i_1, i_2, \ldots, i_k, j$. Therefore the principle of optimality applies for this problem. □

Example 5.6 [0/1 knapsack] The 0/1 knapsack problem is similar to the knapsack problem of Section 4.2 except that the x_i's are restricted to have a value of either 0 or 1. Using KNAP(l, j, y) to represent the problem

$$\begin{array}{c}\text{maximize } \sum_{l \le i \le j} p_i x_i \\ \text{subject to } \sum_{l \le i \le j} w_i x_i \le y \\ x_i = 0 \text{ or } 1,\ l \le i \le j\end{array} \tag{5.1}$$

the knapsack problem is KNAP$(1, n, m)$. Let $y_1, y_2, \ldots, y_n$ be an optimal sequence of 0/1 values for $x_1, x_2, \ldots, x_n$, respectively. If $y_1 = 0$, then $y_2, y_3, \ldots, y_n$ must constitute an optimal sequence for the problem KNAP(2, n, m). If it does not, then $y_1, y_2, \ldots, y_n$ is not an optimal sequence for KNAP$(1, n, m)$. If $y_1 = 1$, then $y_2, \ldots, y_n$ must be an optimal sequence for the problem KNAP$(2, n, m - w_1)$. If it isn't, then there is another 0/1 sequence $z_2, z_3, \ldots, z_n$ such that $\sum_{2 \le i \le n} w_i z_i \le m - w_1$ and $\sum_{2 \le i \le n} p_i z_i > \sum_{2 \le i \le n} p_i y_i$. Hence, the sequence $y_1, z_2, z_3, \ldots, z_n$ is a sequence for (5.1) with greater value. Again the principle of optimality applies. □

Example 5.7 [Shortest path] Let A_i be the set of vertices adjacent to vertex i. For each vertex $k \in A_i$, let Γ_k be a shortest path from k to j. Then, a shortest i to j path is the shortest of the paths $\{i, \Gamma_k | k \in A_i\}$. □

Example 5.8 [0/1 knapsack] Let $g_j(y)$ be the value of an optimal solution to KNAP$(j+1, n, y)$. Clearly, $g_0(m)$ is the value of an optimal solution to KNAP$(1, n, m)$. The possible decisions for x_1 are 0 and 1 ($D_1 = \{0, 1\}$). From the principle of optimality it follows that

$$g_0(m) = \max\ \{g_1(m),\ g_1(m - w_1) + p_1\} \tag{5.2}$$

□

While the principle of optimality has been stated only with respect to the initial state and decision, it can be applied equally well to intermediate states and decisions. The next two examples show how this can be done.

Example 5.9 [Shortest path] Let k be an intermediate vertex on a shortest i to j path $i, i_1, i_2, \ldots, k, p_1, p_2, \ldots, j$. The paths $i, i_1, \ldots, k$ and $k, p_1, \ldots, j$ must, respectively, be shortest i to k and k to j paths. □

Example 5.10 [0/1 knapsack] Let $y_1, y_2, \ldots, y_n$ be an optimal solution to KNAP$(1, n, m)$. Then, for each j, $1 \le j \le n$, $y_1, \ldots, y_j$, and $y_{j+1}, \ldots, y_n$ must be optimal solutions to the problems KNAP$(1, j, \sum_{1 \le i \le j} w_i y_i)$ and KNAP$(j+1, n, m - \sum_{1 \le i \le j} w_i y_i)$ respectively. This observation allows us to generalize (5.2) to

$$g_i(y) = \max\ \{g_{i+1}(y),\ g_{i+1}(y - w_{i+1}) + p_{i+1}\} \tag{5.3}$$

The recursive application of the optimality principle results in a recurrence equation of type (5.3). Dynamic programming algorithms solve this recurrence to obtain a solution to the given problem instance. The recurrence (5.3) can be solved using the knowledge $g_n(y) = 0$ for all $y \ge 0$ and $g_n(y) = -\infty$ for $y < 0$. From $g_n(y)$, one can obtain $g_{n-1}(y)$ using (5.3) with $i = n-1$. Then, using $g_{n-1}(y)$, one can obtain $g_{n-2}(y)$. Repeating in this way, one can determine $g_1(y)$ and finally $g_0(m)$ using (5.3) with $i = 0$.

1.2 Multistage Graphs

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \ge 2$ disjoint sets V_i, $1 \le i \le k$. In addition, if $\langle u, v\rangle$ is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \le i < k$. The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$. Let s and t, respectively, be the vertices in V_1 and V_k. The vertex s is the *source*, and t the *sink*. Let $c(i, j)$ be the cost of edge $\langle i, j\rangle$. The cost of a path from s to t is the sum of the costs of the edges on the path. The *multistage graph problem* is to find a minimum-cost path from s to t. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k. Figure 5.2 shows a five-stage graph. A minimum-cost s to t path is indicated by the broken edges.

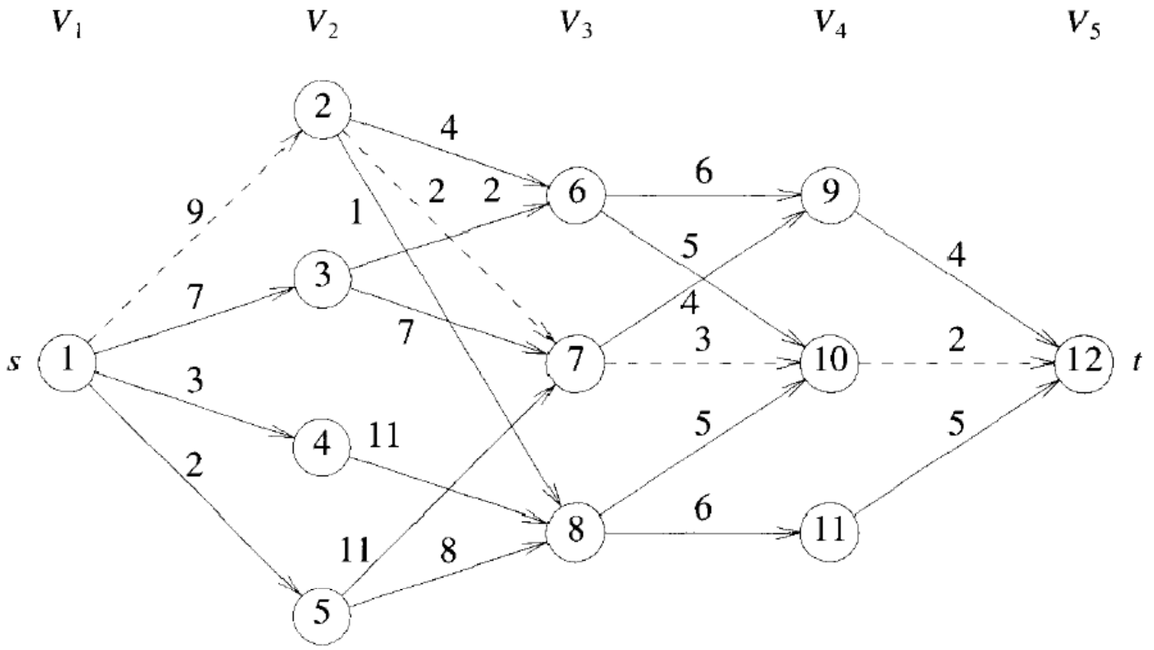


Figure: Five stage graph

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of $k-2$ decisions. The ith decision involves determining which vertex in V_{i+1}, $1 \le i \le k-2$, is to be on the path. It is easy to see that the principle of optimality holds. Let $p(i,j)$ be a minimum-cost path from vertex j in V_i to vertex t. Let $cost(i,j)$ be the cost of this path. Then, using the forward approach, we obtain

$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l\rangle \in E}} \{c(j,l) + cost(i+1,l)\} \tag{5.5}$$

Since, $cost(k-1,j) = c(j,t)$ if $\langle j,t\rangle \in E$ and $cost(k-1,j) = \infty$ if $\langle j,t\rangle \notin E$, (5.5) may be solved for $cost(1,s)$ by first computing $cost(k-2,j)$ for all $j \in V_{k-2}$, then $cost(k-3,j)$ for all $j \in V_{k-3}$, and so on, and finally $cost(1,s)$. Trying this out on the graph of Figure 5.2, we obtain

$$\begin{aligned} cost(3,6) &= \min\ \{6 + cost(4,9), 5 + cost(4,10)\} \\ &= 7 \\ cost(3,7) &= \min\ \{4 + cost(4,9), 3 + cost(4,10)\} \\ &= 5 \end{aligned}$$

$$\begin{aligned}
cost(3,8) &= 7\\
cost(2,2) &= \min\ \{4+cost(3,6), 2+cost(3,7), 1+cost(3,8)\}\\
&= 7\\
cost(2,3) &= 9\\
cost(2,4) &= 18\\
cost(2,5) &= 15\\
cost(1,1) &= \min\ \{9+cost(2,2), 7+cost(2,3), 3+cost(2,4),\\
&\qquad\qquad 2+cost(2,5)\}\\
&= 16
\end{aligned}$$

Note that in the calculation of $cost(2,2)$, we have reused the values of $cost(3,6), cost(3,7)$, and $cost(3,8)$ and so avoided their recomputation. A minimum cost s to t path has a cost of 16. This path can be determined easily if we record the decision made at each state (vertex). Let $d(i,j)$ be the value of l (where l is a node) that minimizes $c(j,l) + cost(i+1,l)$ (see Equation 5.5). For Figure 5.2 we obtain

$$\begin{array}{llllllllllll}
d(3,6) & = & 10; & d(3,7) & = & 10; & d(3,8) & = & 10; & & & \\
d(2,2) & = & 7; & d(2,3) & = & 6; & d(2,4) & = & 8; & d(2,5) & - & 8;\\
d(1,1) & = & 2 & & & & & & & & &
\end{array}$$

Let the minimum-cost path be $s = 1, v_2, v_3, \ldots, v_{k-1}, t$. It is easy to see that $v_2 = d(1,1) = 2, v_3 = d(2,d(1,1)) = 7$, and $v_4 = d(3,d(2,\ d(1,1))) = d(3,7) = 10$.

Algorithm 5.1 Multistage graph pseudocode corresponding to the forward approach

```
Algorithm FGraph($G, k, n, p$)
// The input is a $k$-stage graph $G = (V, E)$ with $n$ vertices
// indexed in order of stages. $E$ is a set of edges and $c[i, j]$
// is the cost of $\langle i, j \rangle$. $p[1 : k]$ is a minimum-cost path.
{
    $cost[n] := 0.0$;
    for $j := n - 1$ to 1 step -1 do
    { // Compute $cost[j]$.
        Let $r$ be a vertex such that $\langle j, r \rangle$ is an edge
        of $G$ and $c[j, r] + cost[r]$ is minimum;
        $cost[j] := c[j, r] + cost[r]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := 2$ to $k - 1$ do $p[j] := d[p[j - 1]]$;
}
```

The complexity analysis of the function FGraph is fairly straightforward. If G is represented by its adjacency lists, then r in line 9 of Algorithm 5.1 can be found in time proportional to the degree of vertex j. Hence, if G has $|E|$ edges, then the time for the **for** loop of line 7 is $\Theta(|V| + |E|)$. The time for the **for** loop of line 16 is $\Theta(k)$. Hence, the total time is $\Theta(|V| + |E|)$. In addition to the space needed for the input, space is needed for $cost[\ ]$, $d[\ ]$, and $p[\ ]$.

Backward Approach

The multistage graph problem can also be solved using the backward approach. Let $bp(i, j)$ be a minimum-cost path from vertex s to a vertex j in V_i. Let $bcost(i, j)$ be the cost of $bp(i, j)$. From the backward approach we obtain

$$bcost(i,j) = \min_{\substack{l \in V_{i-1} \\ \langle l,j \rangle \in E}} \{bcost(i-1,l) + c(l,j)\} \tag{5.6}$$

Since $bcost(2, j) = c(1, j)$ if $\langle 1, j \rangle \in E$ and $bcost(2, j) = \infty$ if $\langle 1, j \rangle \notin E$, $bcost(i, j)$ can be computed using (5.6) by first computing $bcost$ for $i = 3$, then for $i = 4$, and so on. For the graph of Figure 5.2, we obtain

$$\begin{aligned} bcost(3,6) &= \min\ \{bcost(2,2) + c(2,6), bcost(2,3) + c(3,6)\} \\ &= \min\ \{9 + 4, 7 + 2\} \\ &= 9 \end{aligned}$$

$$\begin{aligned} bcost(3,7) &= 11 & \qquad bcost(4,10) &= 14 \\ bcost(3,8) &= 10 & \qquad bcost(4,11) &= 16 \\ bcost(4,9) &= 15 & \qquad bcost(5,12) &= 16 \end{aligned}$$

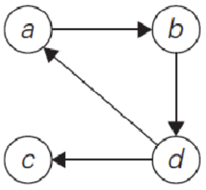
Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

```
Algorithm BGraph(G, k, n, p)
// Same function as FGraph
{
    bcost[1] := 0.0;
    for j := 2 to n do
    { // Compute bcost[j].
        Let r be such that ⟨r, j⟩ is an edge of
        G and bcost[r] + c[r, j] is minimum;
        bcost[j] := bcost[r] + c[r, j];
        d[j] := r;
    }
    // Find a minimum-cost path.
    p[1] := 1; p[k] := n;
    for j := k - 1 to 2 do p[j] := d[p[j + 1]];
}
```

2. Transitive Closure using Warshall's Algorithm,

Definition: The **transitive closure** of a directed graph with n vertices can be defined as the n × n boolean matrix T = {t_{ij} }, in which the element in the i^{th} row and the j^{th} column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t^{ij} is 0.

Example: An example of a digraph, its adjacency matrix, and its transitive closure is given below.



(a) Digraph.

$$A = \begin{array}{c|cccc|} & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

(b) Its adjacency matrix.

$$T = \begin{array}{c|cccc|} & a & b & c & d \\ a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

(c) Its transitive closure.

We can generate the transitive closure of a digraph with the help of depthfirst search or breadth-first search. Performing either traversal starting at the i^{th} vertex gives the information about the vertices reachable from it and hence the columns that contain 1's in the i^{th} row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

Since this method traverses the same digraph several times, we can use a better algorithm called **Warshall's algorithm**. Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}.$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix $R^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to 1 if and only if there exists a directed path of a positive length from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k.

Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph. $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate. The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

This means that there exists a path from the ith vertex vi to the jth vertex vj with each intermediate vertex numbered not higher than k:

v_i, a list of intermediate vertices each numbered not higher than k, v_j . --- (*)

Two situations regarding this path are possible.

1. In the first, the list of its intermediate vertices **does not** contain the kth vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than k−1. i.e. $r_{ij}^{(k-1)} = 1$
2. The second possibility is that path (*) **does contain** the kth vertex v_k among the intermediate vertices. Then path (*) can be rewritten as;

$$v_i, \text{ vertices numbered} \le k-1, v_k, \text{ vertices numbered} \le k-1, v_j .$$

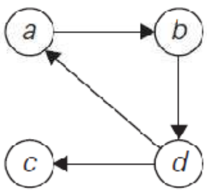
$$\text{i.e } r_{ik}^{(k-1)} = 1 \text{ and } r_{kj}^{(k-1)} = 1$$

Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right)$$

The Warshall's algorithm works based on the above formula.

As an example, the application of Warshall's algorithm to the digraph is shown below. New 1's are in bold.



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

```
ALGORITHM  Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

Analysis

Its time efficiency is $\Theta(n^3)$. We can make the algorithm to run faster by treating matrix rows as bit strings and employ the bitwise or operation available in most modern computer languages.

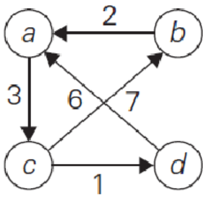
Space efficiency: Although separate matrices for recording intermediate results of the algorithm are used, that can be avoided.

3. All Pairs Shortest Paths using Floyd's Algorithm,

Problem definition: Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances—i.e., the lengths of the shortest paths - from each vertex to all other vertices.

Applications: Solution to this problem finds applications in communications, transportation networks, and operations research. Among recent applications of the all-pairs shortest-path problem is pre-computing distances for motion planning in computer games.

We store the lengths of shortest paths in an n x n matrix D called the distance matrix: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex.



(a) Digraph.

$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

(b) Its weight matrix.

$$D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

(c) Its distance matrix

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called **Floyd's algorithm.**

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n × n matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}.$$

The element $d_{ij}^{(k)}$ in the ith row and the jth column of matrix D$^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k.

As in Warshall's algorithm, we can compute all the elements of each matrix D$^{(k)}$ from its immediate predecessor D$^{(k-1)}$

If $d_{ij}^{(k)} = 1$, then it means that there is a path;

vi, a list of intermediate vertices each numbered not higher than k, vj .

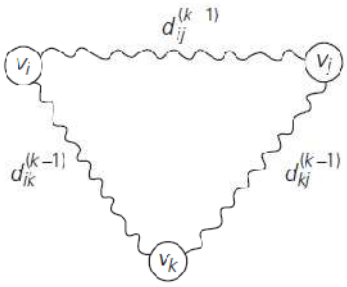
We can partition all such paths into two disjoint subsets: those that do not use the kth vertex v_k as intermediate and those that do.

i. Since the paths of the first subset have their intermediate vertices numbered not higher than k − 1, the shortest of them is, by definition of our matrices, of length $d_{ij}^{(k-1)}$

ii. In the second subset the paths are of the form

v_i, vertices numbered ≤ k − 1, v_k, vertices numbered ≤ k − 1, v_j .

The situation is depicted symbolically in Figure, which shows the underlying idea of Floyd's algorithm.



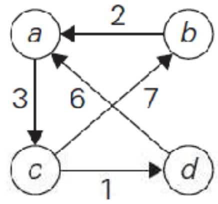
Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}.$$

```
ALGORITHM  Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix W of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths' lengths
    $D \leftarrow W$  //is not necessary if W can be overwritten
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
    return D
```

Analysis: Its time efficiency is Θ(n^3), similar to the warshall's algorithm.

Application of Floyd's algorithm to the digraph is shown below. Updated elements are shown in bold.



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

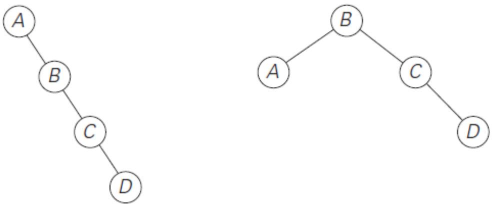
Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

4. Optimal Binary Search Trees

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

If probabilities of searching for elements of a set are known e.g., from accumulated data about past searches it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.

As an example, consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. The figure depicts two out of 14 possible binary search trees containing these keys.



The average number of comparisons in a successful search in the first of these trees is 0.1 * 1+ 0.2 * 2 + 0.4 * 3+ 0.3 * 4 = 2.9, and for the second one it is 0.1 * 2 + 0.2 * 1+ 0.4 * 2 + 0.3 * 3= 2.1. Neither of these two trees is, in fact, optimal.

For our tiny example, we could find the optimal tree by generating all 14 binary search trees with these keys. As a general algorithm, this exhaustive-search approach is unrealistic: the total number of binary search trees with n keys is equal to the nth **Catalan** number,

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$

which grows to infinity as fast as $4^n / n^{1.5}$

So let $a_1, \ldots, a_n$ be distinct keys ordered from the smallest to the largest and let $p_1, \ldots, p_n$ be the probabilities of searching for them. Let C(i, j) be the smallest average number of comparisons made in a successful search in a binary search tree T_i^j made up of keys $a_i, \ldots, a_j$, where i, j are some integer indices, $1 \le i \le j \le n$.

Following the classic dynamic programming approach, we will find values of C(i, j) for all smaller instances of the problem, although we are interested just in C(1, n). To derive a recurrence underlying a dynamic programming algorithm, we will consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree (Figure 8.8), the root contains key ak, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree T^j_{k+1}contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged. (Note how we are taking advantage of the principle of optimality here.)

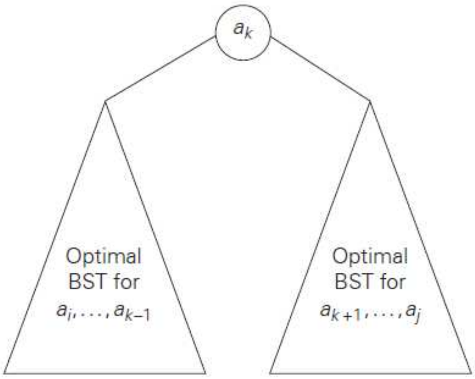

a_k
Optimal BST for $a_i, \ldots, a_{k-1}$
Optimal BST for $a_{k+1}, \ldots, a_j$

FIGURE 8.8 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j.

If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

$$C(i,j) = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1)$$
$$+ \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\}$$

$$= \min_{i\le k\le j} \{\sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s\}$$

$$= \min_{i\le k\le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

Thus, we have the recurrence

$$C(i, j) = \min_{i\le k\le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n. \qquad \textbf{(8.8)}$$

We assume in formula (8.8) that $C(i, i-1) = 0$ for $1 \le i \le n+1$, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that

$$C(i, i) = p_i \quad \text{for } 1 \le i \le n,$$

as it should be for a one-node binary search tree containing a_i.

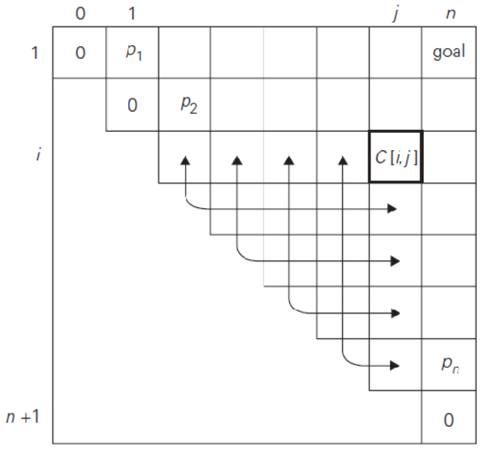


FIGURE 8.9 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

The two-dimensional table in Figure 8.9 shows the values needed for computing C(i, j) by formula (8.8): they are in row i and the columns to the left of column j and in column j and the rows below row i. The arrows point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of C(i, j). This suggests filling the table along its diagonals, starting with all zeros on the main diagonal and given probabilities pi, 1≤ i ≤ n, right above it and moving toward the upper right corner.

The algorithm we just sketched computes C(1, n)—the average number of comparisons for successful searches in the optimal binary tree. If we also want to get the optimal tree itself, we need to maintain another two-dimensional table to record the value of k for which the minimum in (8.8) is achieved. The table has the same shape as the table in Figure 8.9 and is filled in the same manner, starting with entries R(i, i) = i for 1≤ i ≤ n. When the table is filled, its entries indicate indices of the roots of the optimal subtrees, which makes it possible to reconstruct an optimal tree for the entire set given.

Example: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute C(1, 2):

$$C(1,2) = \min \left\{ \begin{array}{ll} k=1: & C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: & C(1,1) + C(3,2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\}$$

$$= 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4. On finishing the computations we get the following final tables:

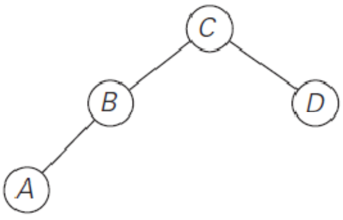
main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

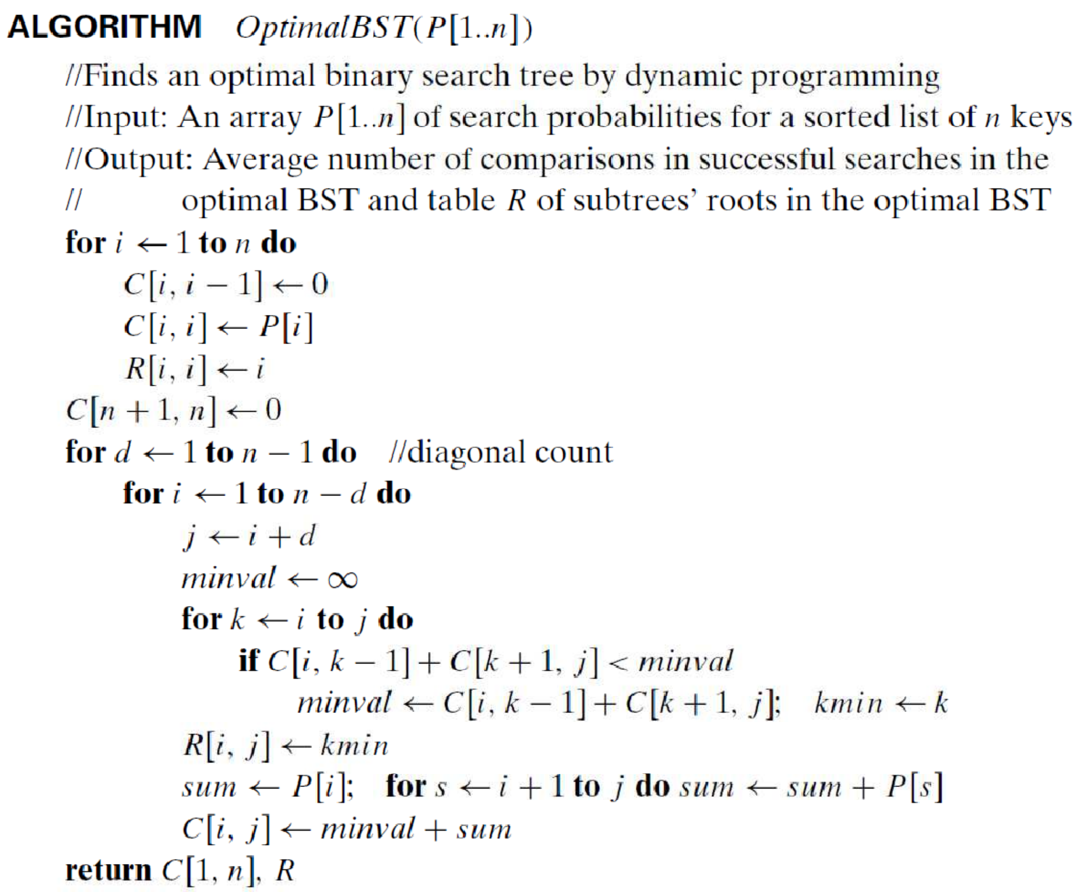
root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since R(1, 4) = 3, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D. To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since R(1, 2) = 2, the root of the optimal tree containing A and B is B, with A being its left child (and the root of the one-node tree: R(1, 1) = 1). Since R(4, 4) = 4, the root of this one-node optimal tree is its only key D. Figure given below presents the optimal tree in its entirety.



Here is Pseudocode of the dynamic programming algorithm.



```
ALGORITHM  $OptimalBST(P[1..n])$
    //Finds an optimal binary search tree by dynamic programming
    //Input: An array $P[1..n]$ of search probabilities for a sorted list of $n$ keys
    //Output: Average number of comparisons in successful searches in the
    //        optimal BST and table $R$ of subtrees' roots in the optimal BST
    for $i \leftarrow 1$ to $n$ do
        $C[i, i-1] \leftarrow 0$
        $C[i, i] \leftarrow P[i]$
        $R[i, i] \leftarrow i$
    $C[n+1, n] \leftarrow 0$
    for $d \leftarrow 1$ to $n-1$ do    //diagonal count
        for $i \leftarrow 1$ to $n-d$ do
            $j \leftarrow i+d$
            $minval \leftarrow \infty$
            for $k \leftarrow i$ to $j$ do
                if $C[i, k-1] + C[k+1, j] < minval$
                    $minval \leftarrow C[i, k-1] + C[k+1, j]$;   $kmin \leftarrow k$
            $R[i, j] \leftarrow kmin$
            $sum \leftarrow P[i]$;   for $s \leftarrow i+1$ to $j$ do $sum \leftarrow sum + P[s]$
            $C[i, j] \leftarrow minval + sum$
    return $C[1, n]$, $R$
```

5. Knapsack problem

We start this section with designing a dynamic programming algorithm for the knapsack problem: given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances.

Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \le j \le W$. Let F(i, j) be the value of an optimal solution to this instance. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the i^{th} item and those that do. Note the following:

i. Among the subsets that do not include the i^{th} item, the value of an optimal subset is, by definition, F(i − 1, j).

ii. Among the subsets that do include the i^{th} item (hence, $j - w_i \ge 0$), an optimal subset is made up of this item and an optimal subset of the first i−1 items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

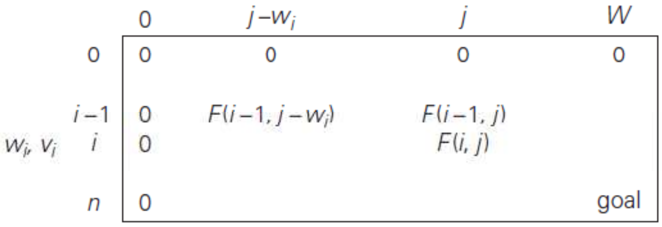
Thus, the value of an optimal solution among all feasible subsets of the first I items is the maximum of these two values.

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \text{ and } F(i, 0) = 0 \text{ for } i \geq 0.$$

Our goal is to find **F(n, W),** the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.



		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Table for solving the knapsack problem by dynamic programming.

Example-1: Let us consider the instance given by the following data:

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

The dynamic programming table, filled by applying formulas is given below

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	**37**

Thus, the maximal value is F(4, 5) = \$37.

We can find the composition of an optimal subset by backtracing the computations of this entry in the table. Since F(4, 5) > F(3, 5), item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity. The value of the latter is F(3, 3). Since F(3, 3) = F(2, 3), item 3 need not be in an optimal subset. Since F(2, 3) > F(1, 3), item 2 is a part of an optimal selection, which leaves element F(1, 3 − 1) to specify its remaining composition. Similarly, since F(1, 2) > F(0, 2), item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

Analysis

The time efficiency and space efficiency of this algorithm are both in Θ(nW). The time needed to find the composition of an optimal solution is in O(n).

Memory Functions

The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient.

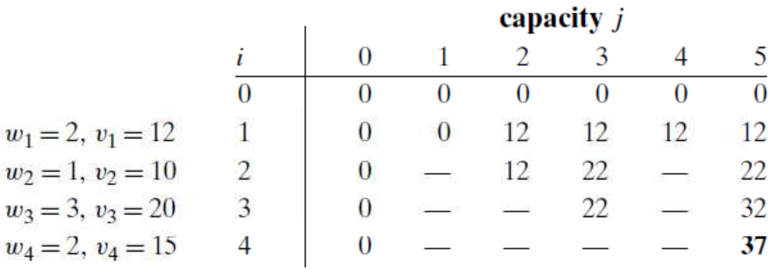
The classic dynamic programming approach, on the other hand, works bottom up: it fills a table with solutions to all smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given. Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using **memory functions**.

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm. Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with i = n (the number of items) and j = W (the knapsack capacity).

```
Algorithm MFKnapsack(i, j )
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first items being
        considered and a nonnegative integer j indicating the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], V alues[1..n], and
        table F[0..n, 0..W ] whose entries are initialized with −1's except for
        row 0 and column 0 initialized with 0's
if $F[i, j] < 0$
    if $j < Weights[i]$
        $value \leftarrow MFKnapsack(i-1, j)$
    else
        $value \leftarrow \max(MFKnapsack(i-1, j),$
                          $Values[i] + MFKnapsack(i-1, j - Weights[i]))$
    $F[i, j] \leftarrow value$
return $F[i, j]$
```

Example-2 Let us apply the memory function method to the instance considered in Example 1. The table in Figure given below gives the results. Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, V (1, 2), is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger.



		capacity *j*					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

Figure: Example of solving an instance of the knapsack problem by the memory function algorithm

In general, we cannot expect more than a constant-factor gain in using the memory function method for the knapsack problem, because its time efficiency class is the same as that of the bottom-up algorithm

6. Bellman-Ford Algorithm (Single source shortest path with –ve weights)

Problem definition

Single source shortest path - Given a graph and a source vertex ***s*** in graph, find shortest paths from ***s*** to all vertices in the given graph. The graph may contain negative weight edges.

Note that we have discussed Dijkstra’s algorithm for single source shortest path problem. Dijksra’s algorithm is a Greedy algorithm and time complexity is O(VlogV). But Dijkstra doesn’t work for graphs with negative weight edges.

Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

How it works?

Like other Dynamic Programming Problems, the algorithm calculates shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i^{th} iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

Let $dist^{\ell}[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most ℓ edges. Then, $dist^{1}[u] = cost[v, u]$, $1 \leq u \leq n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n - 1$ edges. Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u.

Our goal then is to compute $dist^{n-1}[u]$ for all u. This can be done using the dynamic programming methodology. First, we make the following observations:

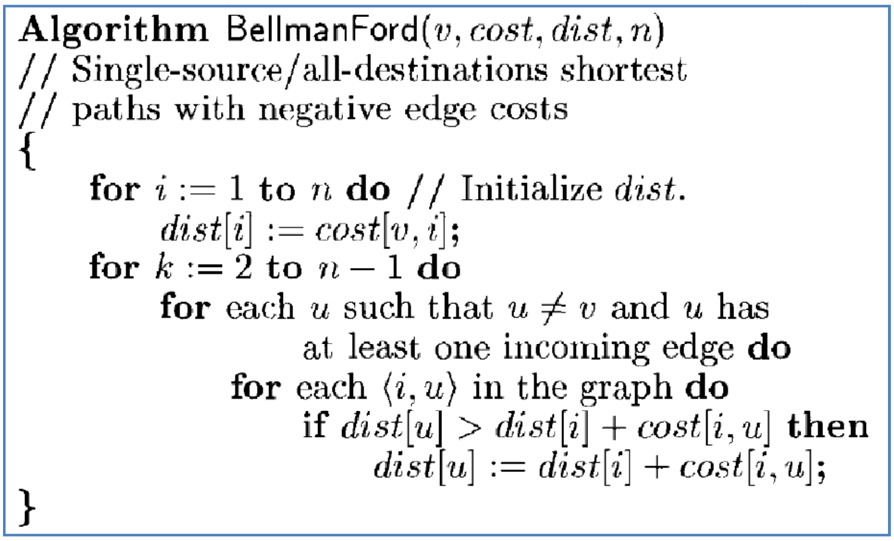
1. If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k - 1$ edges, then $dist^{k}[u] = dist^{k-1}[u]$.
2. If the shortest path from v to u with at most k, $k > 1$, edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the edge $\langle j, u \rangle$. The path from v to j has $k - 1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $\langle i, u \rangle$ is in the graph are candidates for j. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i, u]$ is the correct value for j.

These observations result in the following recurrence for $dist$:

$$dist^{k}[u] = \min \{dist^{k-1}[u], \min_{i} \{dist^{k-1}[i] + cost[i, u]\}\}$$

This recurrence can be used to compute $dist^{k}$ from $dist^{k-1}$, for $k = 2, 3, \ldots, n - 1$.

Bellman-Ford algorithm to compute shortest path

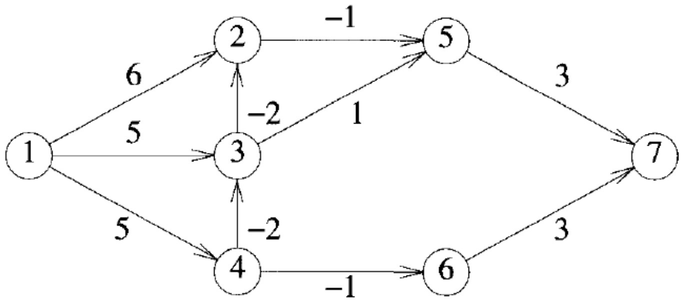


```
Algorithm BellmanFord($v, cost, dist, n$)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
    for $i := 1$ to $n$ do // Initialize $dist$.
        $dist[i] := cost[v, i]$;
    for $k := 2$ to $n - 1$ do
        for each $u$ such that $u \neq v$ and $u$ has
                at least one incoming edge do
            for each $\langle i, u \rangle$ in the graph do
                if $dist[u] > dist[i] + cost[i, u]$ then
                    $dist[u] := dist[i] + cost[i, u]$;
}
```

Example 5.16 Figure 5.10 gives a seven-vertex graph, together with the arrays $dist^k$, $k = 1, \ldots, 6$. These arrays were computed using the equation just given. For instance, $dist^k[1] = 0$ for all k since 1 is the source node. Also, $dist^1[2] = 6, dist^1[3] = 5$, and $dist^1[4] = 5$, since there are edges from 1 to these nodes. The distance $dist^1[]$ is ∞ for the nodes 5, 6, and 7 since there are no edges to these from 1.

$$\begin{aligned} dist^2[2] &= \min\,\{dist^1[2], \min_i dist^1[i] + cost[i,2]\} \\ &= \min\,\{6, 0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty, \infty+\infty\} = 3 \end{aligned}$$

Here the terms $0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty$, and $\infty+\infty$ correspond to a choice of $i = 1, 3, 4, 5, 6$, and 7, respectively. The rest of the entries are computed in an analogous manner. □



(a) A directed graph

k	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 5.10 Shortest paths with negative edge lengths

7. Travelling Sales Person problem (T2:5.9),

We have seen how to apply dynamic programming to a subset selection problem (0/1 knapsack). Now we turn our attention to a permutation problem. Note that permutation problems usually are much harder to solve than subset problems as there are $n!$ different permutations of n objects whereas there are only 2^n different subsets of n objects ($n! > 2^n$). Let $G = (V, E)$ be a directed graph with edge costs c_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$ if $\langle i, j\rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A *tour* of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The *traveling salesperson problem* is to find a tour of minimum cost.

The traveling salesperson problem finds application in a variety of situations. Suppose we have to route a postal van to pick up mail from mail boxes located at n different sites. An $n+1$ vertex graph can be used to represent the situation. One vertex represents the post office from which the postal van starts and to which it must return. Edge $\langle i, j\rangle$ is assigned a cost equal to the distance from site i to site j. The route taken by the postal van is a tour, and we are interested in finding a tour of minimum length.

As a second example, suppose we wish to use a robot arm to tighten the nuts on some piece of machinery on an assembly line. The arm will start from its initial position (which is over the first nut to be tightened), successively move to each of the remaining nuts, and return to the initial position. The path of the arm is clearly a tour on a graph in which vertices represent the nuts. A minimum-cost tour will minimize the time needed for the arm to complete its task (note that only the total arm movement time is variable; the nut tightening time is independent of the tour).

In the following discussion we shall, without loss of generality, regard a tour to be a simple path that starts and ends at vertex 1. Every tour consists of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once. It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V - \{1, k\}$. Hence, the principle of optimality holds. Let $g(i, S)$ be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function $g(1, V - \{1\})$ is the length of an optimal salesperson tour. From the principal of optimality it follows that

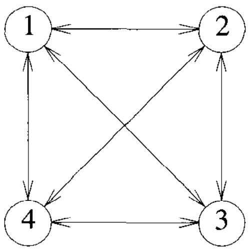
$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \tag{5.20}$$

Generalizing (5.20), we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \tag{5.21}$$

Equation 5.20 can be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all choices of k. The g values can be obtained by using (5.21). Clearly, $g(i, \phi) = c_{i1}$, $1 \le i \le n$. Hence, we can use (5.21) to obtain $g(i, S)$ for all S of size 1. Then we can obtain $g(i, S)$ for S with $|S| = 2$, and so on. When $|S| < n - 1$, the values of i and S for which $g(i, S)$ is needed are such that $i \neq 1$, $1 \notin S$, and $i \notin S$.

Example 5.26 Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix c of Figure 5.21(b).



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Figure 5.21 Directed graph and edge length matrix c

Thus $g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6$, and $g(4,\phi) = c_{41} = 8$. Using (5.21), we obtain

$$\begin{array}{lclclclcl} g(2,\{3\}) & = & c_{23}+g(3,\phi) & = & 15 & & g(2,\{4\}) & = & 18 \\ g(3,\{2\}) & = & 18 & & & & g(3,\{4\}) & = & 20 \\ g(4,\{2\}) & = & 13 & & & & g(4,\{3\}) & = & 15 \end{array}$$

Next, we compute $g(i,S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{lclcl} g(2,\{3,4\}) & = & \min\ \{c_{23}+g(3,\{4\}), c_{24}+g(4,\{3\})\} & = & 25 \\ g(3,\{2,4\}) & = & \min\ \{c_{32}+g(2,\{4\}), c_{34}+g(4,\{2\})\} & = & 25 \\ g(4,\{2,3\}) & = & \min\ \{c_{42}+g(2,\{3\}), c_{43}+g(3,\{2\})\} & = & 23 \end{array}$$

Finally, from (5.20) we obtain

$$\begin{array}{lcl} g(1,\{2,3,4\}) & = & \min\{c_{12}+g(2,\{3,4\}), c_{13}+g(3,\{2,4\}), c_{14}+g(4,\{2,3\})\} \\ & = & \min\{35,40,43\} \\ & = & 35 \end{array}$$

An optimal tour of the graph of Figure 5.21(a) has length 35. A tour of this length can be constructed if we retain with each $g(i,S)$ the value of j that minimizes the right-hand side of (5.21). Let $J(i,S)$ be this value. Then, $J(1,\{2,3,4\}) = 2$. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from $g(2, \{3, 4\})$. So $J(2, \{3, 4\}) = 4$. Thus the next edge is $\langle 2, 4\rangle$. The remaining tour is for $g(4, \{3\})$. So $J(4, \{3\}) = 3$. The optimal tour is 1, 2, 4, 3, 1. □

Let N be the number of $g(i,S)$'s that have to be computed before (5.20) can be used to compute $g(1, V - \{1\})$. For each value of $|S|$ there are $n-1$ choices for i. The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$. Hence

$$N = \sum_{k=0}^{n-2}(n-1)\binom{n-2}{k} = (n-1)2^{n-2}$$

An algorithm that proceeds to find an optimal tour by using (5.20) and (5.21) will require $\Theta(n^2 2^n)$ time as the computation of $g(i, S)$ with $|S| = k$ requires $k-1$ comparisons when solving (5.21). This is better than enumerating all $n!$ different tours to find the best one. The most serious drawback of this dynamic programming solution is the space needed, $O(n2^n)$. This is too large even for modest values of n.

8. Reliability design

In this section we look at an example of how to use dynamic programming to solve a problem with a multiplicative optimization function. The problem is to design a system that is composed of several devices connected in series (Figure 5.19). Let r_i be the reliability of device D_i (that is, r_i is the probability that device i will function properly). Then, the reliability of the entire system is Πr_i. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. For example, if $n = 10$ and $r_i = .99$, $1 \leq i \leq 10$, then $\Pi r_i = .904$. Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel (Figure 5.20) through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

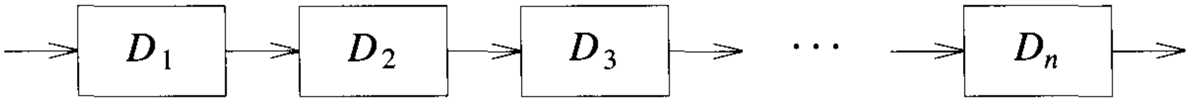


Figure 5.19 n devices D_i, $1 \leq i \leq n$, connected in series

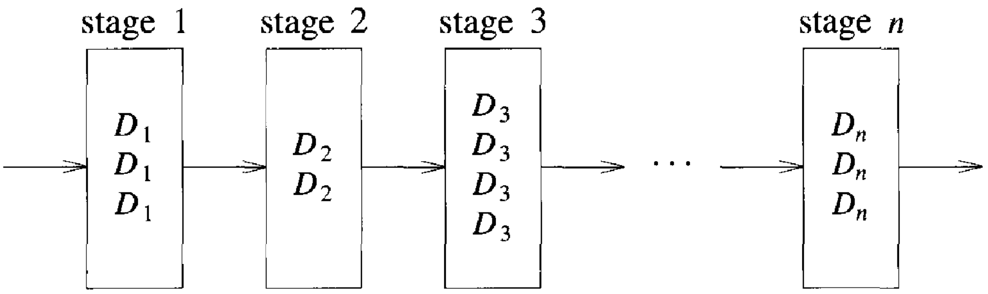


Figure 5.20 Multiple devices connected in parallel in each stage

If stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$. Thus, if $r_i = .99$ and $m_i = 2$, the stage reliability becomes .9999. In any practical situation, the stage reliability is a little less than $1 - (1 - r_i)^{m_i}$ because the switching circuits themselves are not fully reliable. Also, failures of copies of the same device may not be fully independent (e.g., if failure is due to design defect). Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$, $1 \leq n$. (It is quite conceivable that $\phi_i(m_i)$ may decrease after a certain value of m_i.) The reliability of the system of stages is $\Pi_{1 \leq i \leq n} \phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. We wish to solve the following maximization problem:

$$\text{maximize } \Pi_{1 \leq i \leq n} \; \phi_i(m_i)$$

$$\text{subject to } \sum_{1 \leq i \leq n} c_i m_i \leq c$$

$$m_i \geq 1 \text{ and integer}, \; 1 \leq i \leq n$$

A dynamic programming solution can be obtained in a manner similar to that used for the knapsack problem. Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor (c + c_i - \sum_{1}^{n} c_j)/c_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \geq 1$. An optimal solution $m_1, m_2, \ldots, m_n$ is the result of a sequence of decisions, one decision for each m_i. Let $f_i(x)$ represent the maximum value of $\Pi_{1 \leq j \leq i} \; \phi(m_j)$ subject to the constraints $\sum_{1 \leq j \leq i} c_j m_j \leq x$ and $1 \leq m_j \leq u_j$, $1 \leq j \leq i$. Then, the value of an optimal solution is $f_n(c)$. The last decision made requires one to choose m_n from $\{1, 2, 3, \ldots, u_n\}$. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $c - c_n m_n$ in an optimal way. The principal of optimality holds and

$$f_n(c) = \max_{1 \leq m_n \leq u_n} \{\phi_n(m_n) f_{n-1}(c - c_n m_n)\} \qquad (5.18)$$

For any $f_i(x)$, $i \geq 1$, this equation generalizes to

$$f_i(x) = \max_{1 \leq m_i \leq u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\} \qquad (5.19)$$

Clearly, $f_0(x) = 1$ for all x, $0 \leq x \leq c$. Hence, (5.19) can be solved using an approach similar to that used for the knapsack problem. Let S^i consist of tuples of the form (f, x), where $f = f_i(x)$. There is at most one tuple for each different x that results from a sequence of decisions on $m_1, m_2, \ldots, m_n$. The dominance rule (f_1, x_1) dominates (f_2, x_2) iff $f_1 \geq f_2$ and $x_1 \leq x_2$ holds for this problem too. Hence, dominated tuples can be discarded from S^i.

Example 5.25 We are to design a three stage system with device types D_1, D_2, and D_3. The costs are \$30, \$15, and \$20 respectively. The cost of the system is to be no more than \$105. The reliability of each device type is .9, .8 and .5 respectively. We assume that if stage i has m_i devices of type i in parallel, then $\phi_i(m_i) = 1-(1-r_i)^{m_i}$. In terms of the notation used earlier, $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, $c = 105$, $r_1 = .9$, $r_2 = .8$, $r_3 = .5, u_1 = 2, u_2 = 3$, and $u_3 = 3$.

We use S^i to represent the set of all undominated tuples (f, x) that may result from the various decision sequences for $m_1, m_2, \ldots, m_i$. Hence, $f(x) = f_i(x)$. Beginning with $S^0 = \{(1, 0)\}$, we can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together. Using S^i_j to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$, we obtain $S^1_1 = \{(.9,\ 30)\}$ and $S^1_2 = \{(.9,\ 30), (.99, 60)\}$. The set $S^2_1 = \{(.72, 45), (.792, 75)\}$; S^2_2= $\{(.864,\ 60)\}$. Note that the tuple (.9504, 90) which comes from (.99, 60) has been eliminated from S^2_2 as this leaves only \$10. This is not enough to allow $m_3 = 1$. The set $S^2_3 = \{(.8928, 75)\}$. Combining, we get $S^2 = \{(.72, 45), (.864, 60), (.8928, 75)\}$ as the tuple (.792, 75) is dominated by (.864, 60). The set $S^3_1 = \{(.36, 65), (.432, 80), (.4464, 95)\}, S^3_2 = \{(.54, 85), (.648, 100)\}$, and $S^3_3 = \{(.63, 105)\}$. Combining, we get $S^3 = \{(.36, 65),\ (.432,\ 80),\ (.54,\ 85),\ (.648, 100)\}$.

The best design has a reliability of .648 and a cost of 100. Tracing back through the S^i's, we determine that $m_1 = 1, m_2 = 2$, and $m_3 = 2$. □

As in the case of the knapsack problem, a complete dynamic programming algorithm for the reliability problem will use heuristics to reduce the size of the S^i's. There is no need to retain any tuple (f, x) in S^i with x value greater that $c - \sum_{i \le j \le n} c_j$ as such a tuple will not leave adequate funds to complete the system. In addition, we can devise a simple heuristic to determine the best reliability obtainable by completing a tuple (f, x) in S^i. If this is less than a heuristically determined lower bound on the optimal system reliability, then (f, x) can be eliminated from S^i.

Vivekananda
College of Engineering & Technology
Nehru Nagar Post, Puttur, D.K. 574203

Lecture Notes on

DAA

15CS43

Design and Analysis of Algorithms

(CBCS Scheme)

Prepared by

Harivinod N
Dept. of Computer Science and Engineering,
VCET Puttur

May 2017

Module-5 : Backtracking

Contents

Course Website
www.TechJourney.in

1. Backtracking

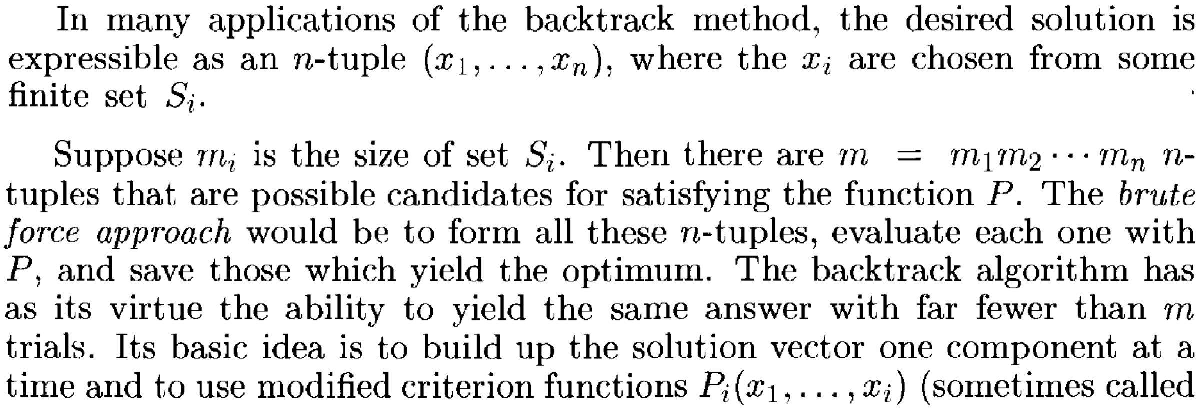
Some problems can be solved, by exhaustive search. The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.

Backtracking is a more intelligent variation of this approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm **backtracks** to replace the last component of the partially constructed solution with its next option.

It is convenient to implement this kind of processing by constructing a tree of choices being made, called the **state-space tree**. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second component, and so on. A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called **non-promising**. Leaves represent either non-promising dead ends or complete solutions found by the algorithm.

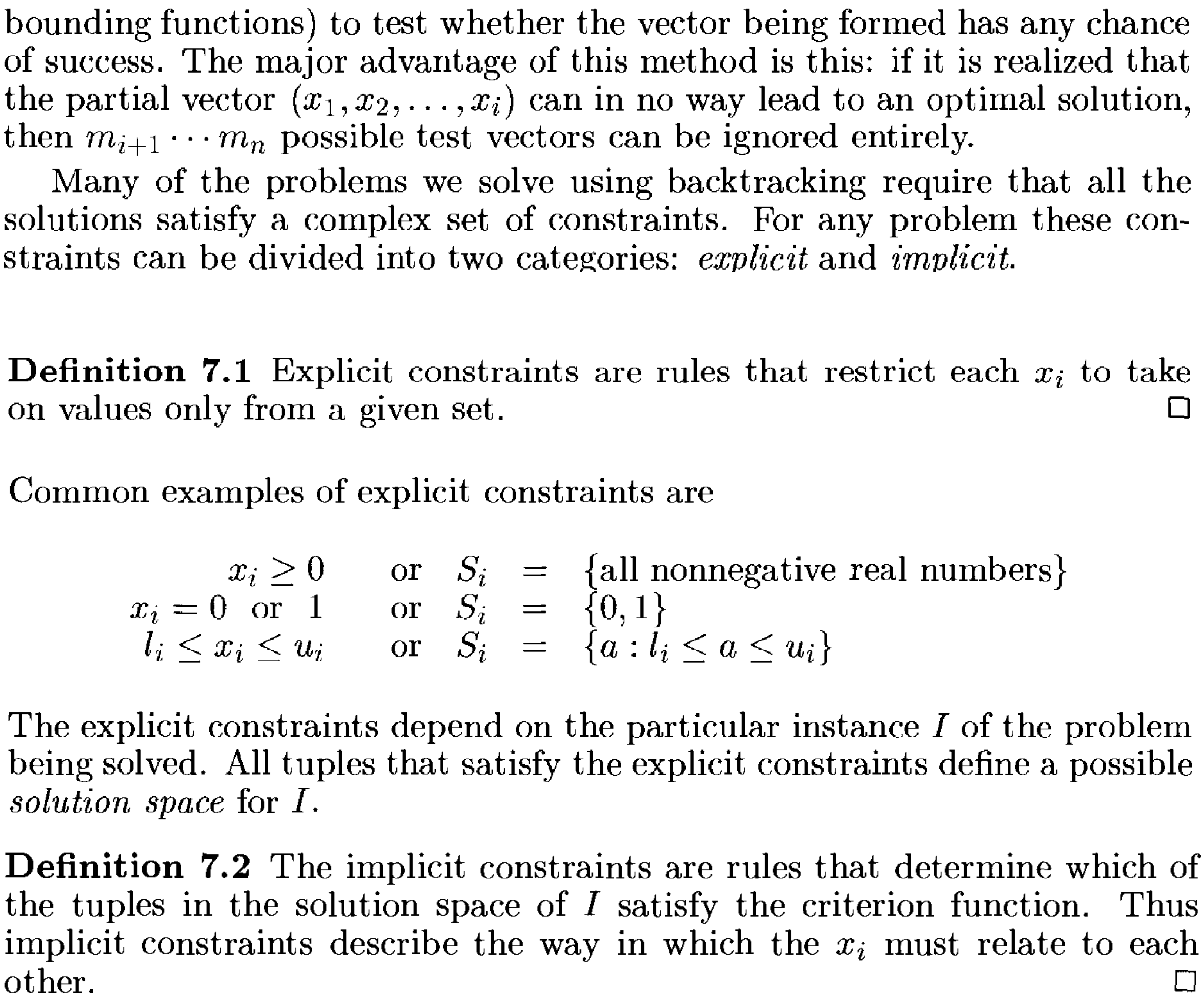
In the majority of cases, a statespace tree for a backtracking algorithm is constructed in the manner of depth-first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

1.1 General method (Textbook T2:7.1)



In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, \ldots, x_n)$, where the x_i are chosen from some finite set S_i.

Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for satisfying the function P. The *brute force approach* would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials. Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \ldots, x_i)$ (sometimes called



bounding functions) to test whether the vector being formed has any chance of success. The major advantage of this method is this: if it is realized that the partial vector $(x_1, x_2, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories: *explicit* and *implicit*.

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. □

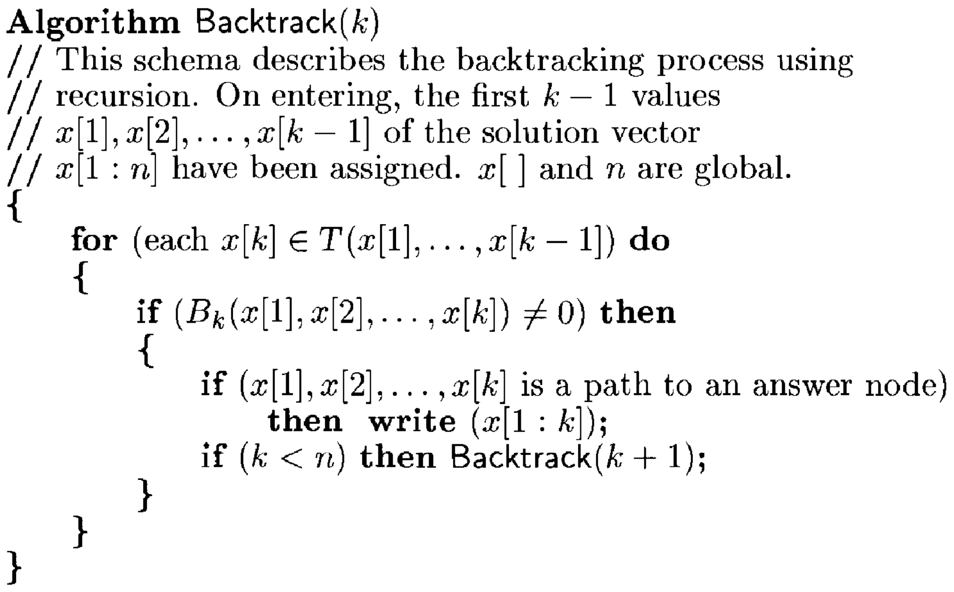
Common examples of explicit constraints are

$$\begin{array}{rcrcl} x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\ x_i = 0 \ \text{ or } \ 1 & \text{or} & S_i & = & \{0, 1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I.

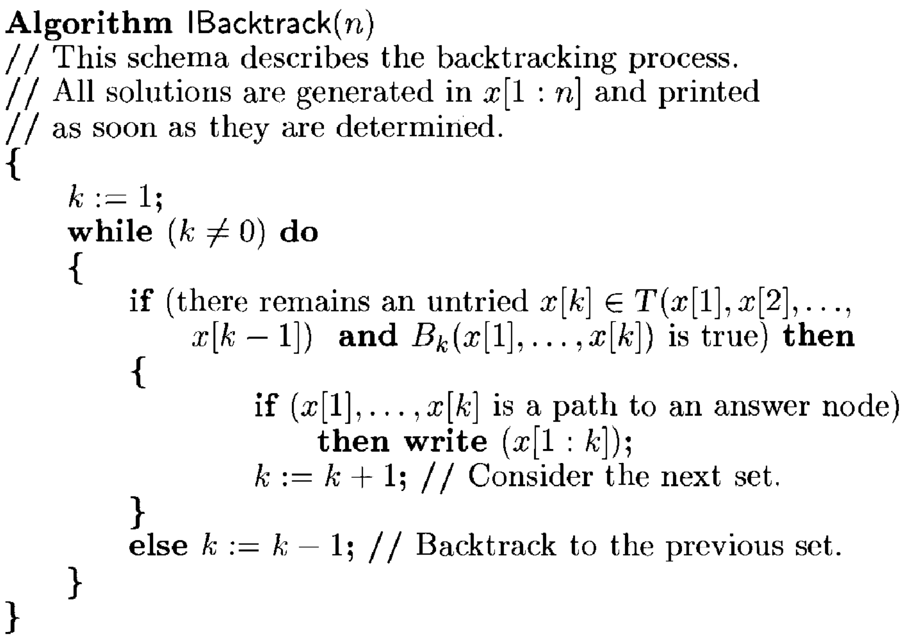
Definition 7.2 The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other. □

General Algorithm (Recursive)



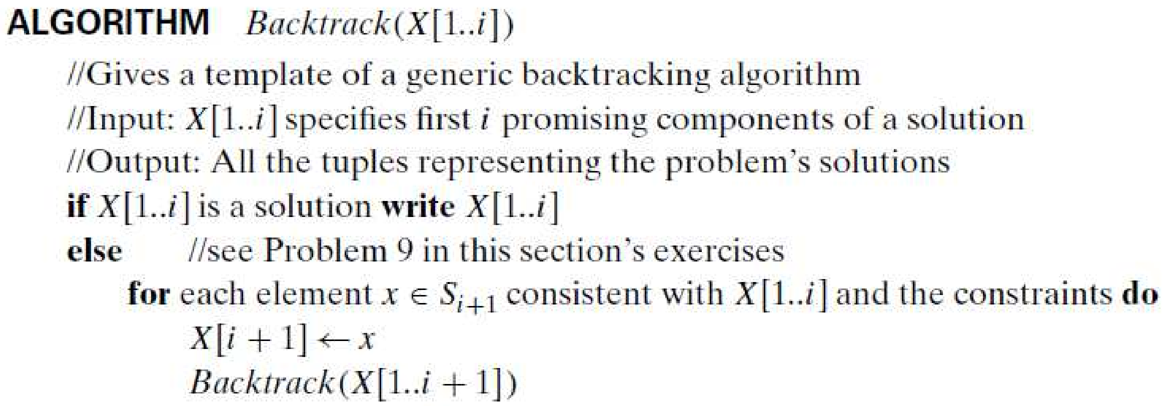
```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first $k-1$ values
// $x[1], x[2], \ldots, x[k-1]$ of the solution vector
// $x[1:n]$ have been assigned. $x[\,]$ and $n$ are global.
{
    for (each $x[k] \in T(x[1], \ldots, x[k-1])$ do
    {
        if ($B_k(x[1], x[2], \ldots, x[k]) \neq 0$) then
        {
            if ($x[1], x[2], \ldots, x[k]$ is a path to an answer node)
                then write ($x[1:k]$);
            if ($k < n$) then Backtrack($k+1$);
        }
    }
}
```

General Algorithm (Iterative)



```
Algorithm IBacktrack($n$)
// This schema describes the backtracking process.
// All solutions are generated in $x[1:n]$ and printed
// as soon as they are determined.
{
    $k := 1$;
    while ($k \neq 0$) do
    {
        if (there remains an untried $x[k] \in T(x[1], x[2], \ldots,$
            $x[k-1])$ and $B_k(x[1], \ldots, x[k])$ is true) then
        {
                if ($x[1], \ldots, x[k]$ is a path to an answer node)
                    then write ($x[1:k]$);
                $k := k + 1$; // Consider the next set.
        }
        else $k := k - 1$; // Backtrack to the previous set.
    }
}
```

General Algorithm for backtracking (From textbook T1)

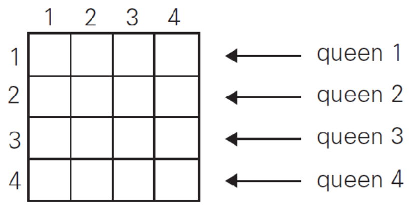


```
ALGORITHM  Backtrack($X[1..i]$)
    //Gives a template of a generic backtracking algorithm
    //Input: $X[1..i]$ specifies first $i$ promising components of a solution
    //Output: All the tuples representing the problem's solutions
    if $X[1..i]$ is a solution write $X[1..i]$
    else    //see Problem 9 in this section's exercises
        for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints do
            $X[i+1] \leftarrow x$
            Backtrack($X[1..i+1]$)
```

1.2 N-Queens problem (Textbook T1:12.1),

The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

So let us consider the **four-queens problem** and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in figure.



We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in figure.

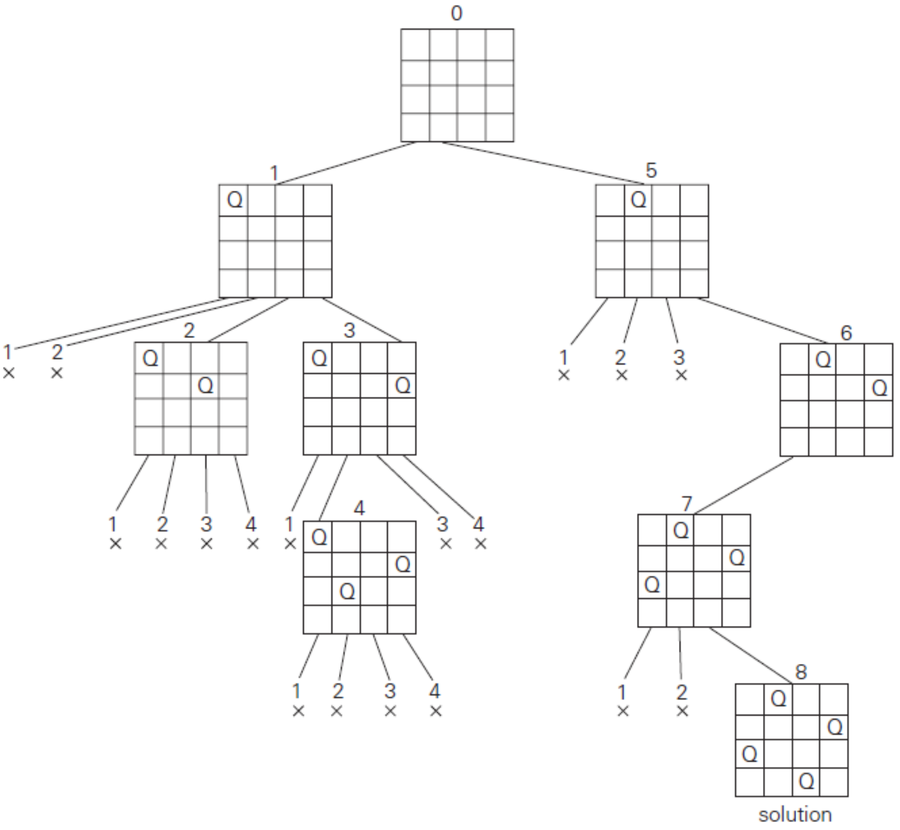


Figure: State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

If other solutions need to be found, the algorithm can simply resume its operations at the leaf at which it stopped. Alternatively, we can use the board's symmetry for this purpose.

Finally, it should be pointed out that a single solution to the n-queens problem for any n ≥ 4 can be found in **linear time**.

Note: The algorithm NQueens() is not in the syllabus. It is given here for interested learners. The algorithm is referred from textbook T2.

```
Algorithm NQueens($k, n$)
// Using backtracking, this procedure prints all
// possible placements of $n$ queens on an $n \times n$
// chessboard so that they are nonattacking.
{
    for $i := 1$ to $n$ do
    {
        if Place($k, i$) then
        {
            $x[k] := i$;
            if ($k = n$) then write ($x[1 : n]$);
            else NQueens($k + 1, n$);
        }
    }
}
```

```
Algorithm Place($k, i$)
// Returns true if a queen can be placed in $k$th row and
// $i$th column. Otherwise it returns false. $x[\,]$ is a
// global array whose first $(k - 1)$ values have been set.
// Abs($r$) returns the absolute value of $r$.
{
    for $j := 1$ to $k - 1$ do
        if (($x[j] = i$) // Two in the same column
             or (Abs($x[j] - i$) = Abs($j - k$)))
                // or in the same diagonal
            then return false;
    return true;
}
```

1.3 Sum of subsets problem

Problem definition: Find a subset of a given set A = {a_1, . . . , a_n } of n positive integers whose sum is equal to a given positive integer d.

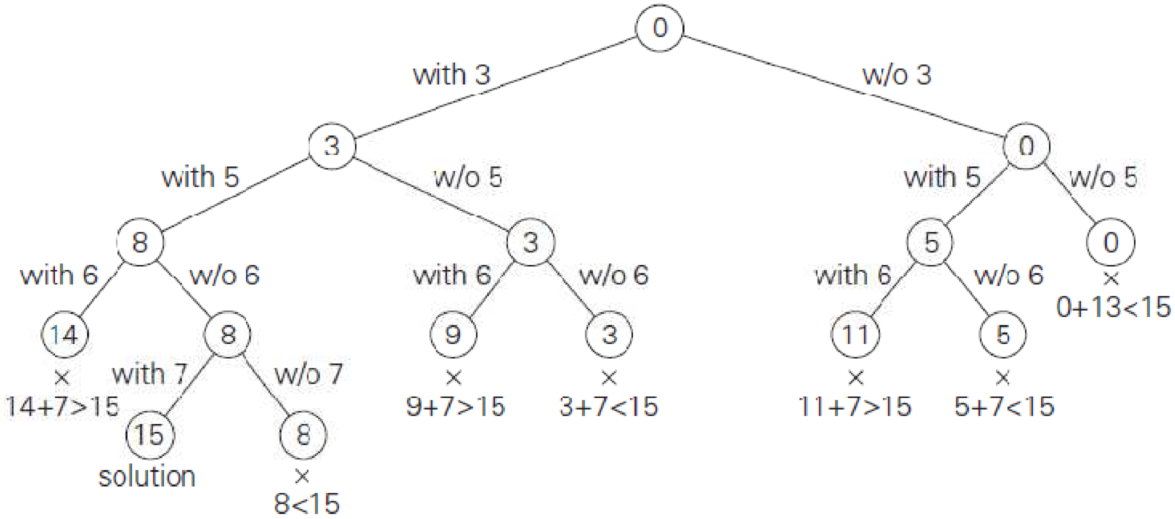
For example, for A = {1, 2, 5, 6, 8} and d = 9, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that

$$a_1 < a_2 < \ldots < a_n.$$

The state-space tree can be constructed as a binary tree like that in Figure shown below for the instance A = {3, 5, 6, 7} and d = 15.

The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.



The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought.

Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.

We record the value of s, the sum of these numbers, in the node. If s is equal to ***d***, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If s is not equal to ***d***, we can terminate the node as non-promising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

Example: Apply backtracking to solve the following instance of the subset sum problem: A = {1, 3, 4, 5} and d = 11.

1.4 Graph coloring

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the *m-colorability decision* problem Note that if d is the degree of the given graph, then it can be colored with $d + 1$

colors. The *m-colorability optimization* problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the *chromatic number* of the graph. For example, the graph of Figure 7.11 can be colored with three colors 1, 2, and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

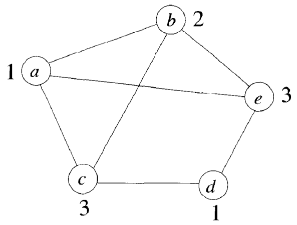


Figure 7.11 An example graph and its coloring

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other. A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs. This problem asks the following question: given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed? This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge. Figure 7.12 shows a map with five regions and its corresponding graph. This map requires four colors. For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient. In this section we consider not only graphs that are produced from maps but all graphs. We are interested in determining all the different ways in which a given graph can be colored using at most m colors.

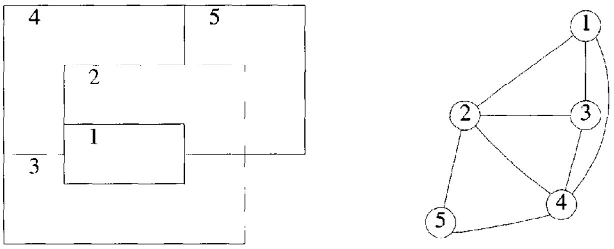
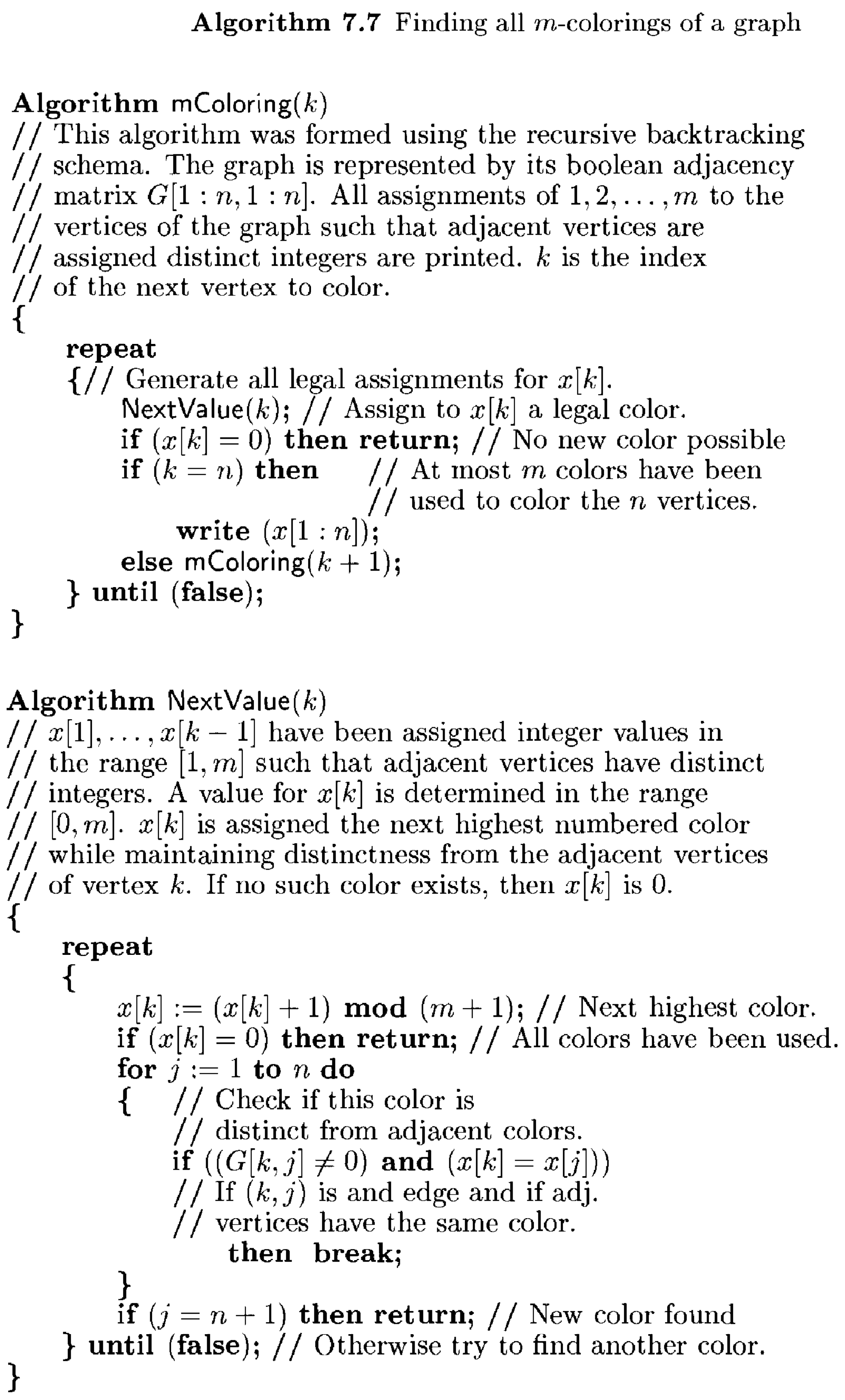


Figure 7.12 A map and its planar graph representation

Suppose we represent a graph by its adjacency matrix $G[1:n, 1:n]$, where $G[i,j] = 1$ if (i,j) is an edge of G, and $G[i,j] = 0$ otherwise. The colors are represented by the integers $1, 2, \ldots, m$ and the solutions are given by the n-tuple $(x_1, \ldots, x_n)$, where x_i is the color of node i. Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is mColoring (Algorithm 7.7). The underlying state space tree used is a level $n+1$ are leaf nodes. Figure 7.13 shows the state space tree when $n = 3$ and $m = 3$.



Algorithm 7.7 Finding all m-colorings of a graph

```
Algorithm mColoring($k$)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix $G[1:n, 1:n]$. All assignments of $1, 2, \ldots, m$ to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. $k$ is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for $x[k]$.
        NextValue($k$); // Assign to $x[k]$ a legal color.
        if ($x[k] = 0$) then return; // No new color possible
        if ($k = n$) then      // At most $m$ colors have been
                               // used to color the $n$ vertices.
            write ($x[1:n]$);
        else mColoring($k+1$);
    } until (false);
}

Algorithm NextValue($k$)
// $x[1], \ldots, x[k-1]$ have been assigned integer values in
// the range $[1, m]$ such that adjacent vertices have distinct
// integers. A value for $x[k]$ is determined in the range
// $[0, m]$. $x[k]$ is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex $k$. If no such color exists, then $x[k]$ is 0.
{
    repeat
    {
        $x[k] := (x[k] + 1)$ mod $(m+1)$; // Next highest color.
        if ($x[k] = 0$) then return; // All colors have been used.
        for $j := 1$ to $n$ do
        {   // Check if this color is
            // distinct from adjacent colors.
            if (($G[k, j] \neq 0$) and ($x[k] = x[j]$))
            // If $(k, j)$ is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if ($j = n+1$) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

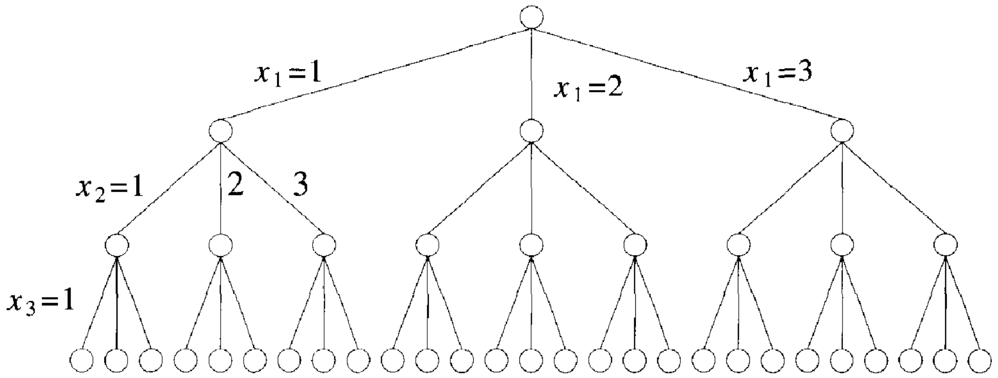


Figure 7.13 State space tree for mColoring when n =3 and m = 3

Function mColoring is begun by first assigning the graph to its adjacency matrix, *setting the array* $x[\ ]$ *to zero*, and then invoking the statement mColoring(1);.

Function NextValue (Algorithm 7.8) produces the possible colors for x_k after x_1 through x_{k-1} have been defined. The main loop of mColoring repeatedly picks an element from the set of possibilities, assigns it to x_k, and then calls mColoring recursively. For instance, Figure 7.14 shows a simple graph containing four nodes. Below that is the tree that is generated by mColoring. Each path to a leaf represents a coloring using at most three colors. Note that only 12 solutions exist with *exactly* three colors. In this tree, after choosing $x_1 = 2$ and $x_2 = 1$, the possible choices for x_3 are 2 and 3. After choosing $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, possible values for x_4 are 1 and 3. And so on.

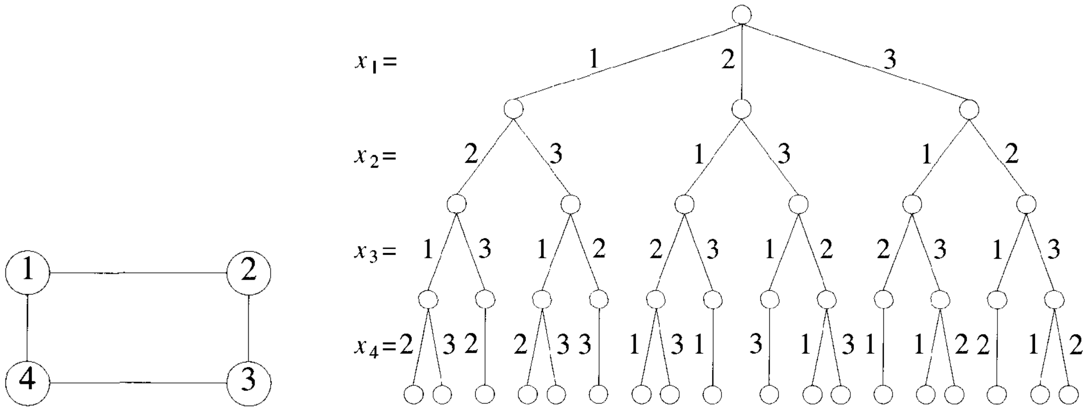


Figure 7.14 A 4-node graph and all possible 3-colorings

Analysis

An upper bound on the computing time of mColoring can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by NextValue to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m - 1) = O(nm^n)$.

1.5 Hamiltonian cycles

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

The graph $G1$ of Figure 7.15 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph $G2$ of Figure 7.15 contains no Hamiltonian cycle. There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.

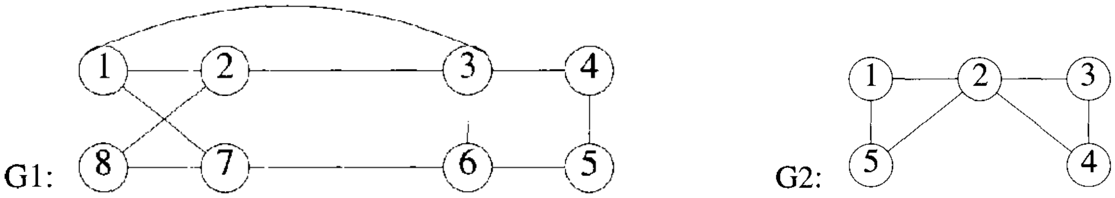


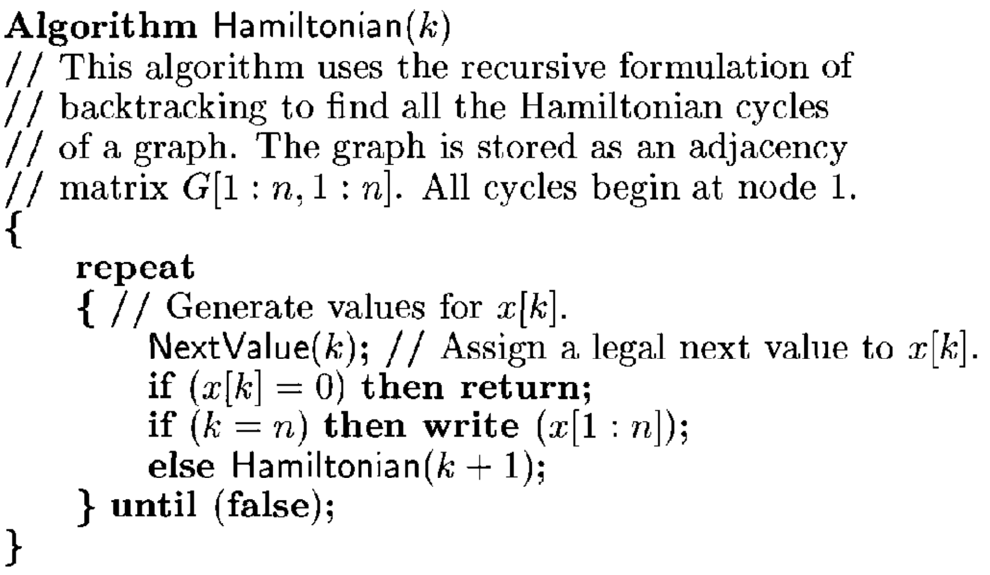
Figure 7.15 Two graphs, one containing a Hamiltonian cycle

We now look at a backtracking algorithm that finds all the Hamiltonian cycles in a graph. The graph may be directed or undirected. Only distinct cycles are output.

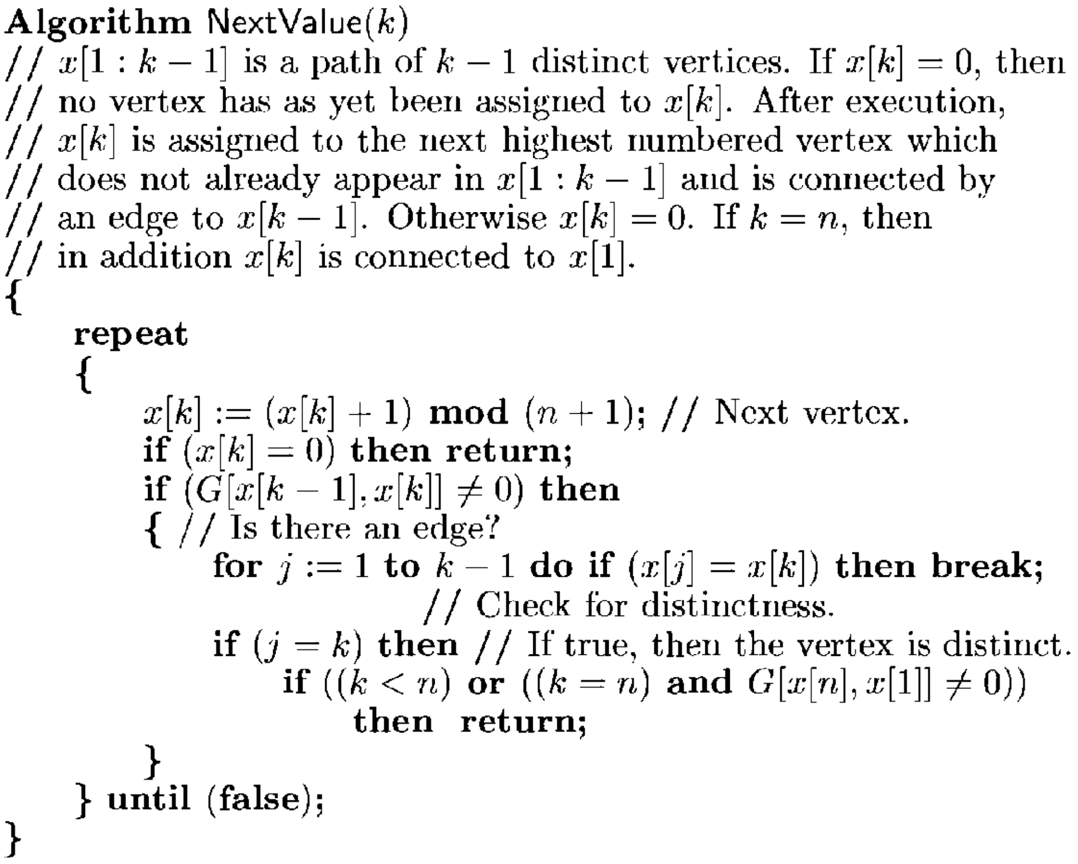
The backtracking solution vector $(x_1, \ldots, x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle. Now all we need do is determine how to compute the set of possible vertices for x_k if $x_1, \ldots, x_{k-1}$ have already been chosen. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to x_{k-1}. The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1. We begin by presenting function NextValue(k) which determines a possible next vertex for the proposed cycle.

Using NextValue we can particularize the recursive backtracking schema to find all Hamiltonian cycles . This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ to zero and $x[1]$ to 1, and then executing Hamiltonian(2).

Recall from the traveling salesperson problem which asked for a tour that has minimum cost. This tour is a Hamiltonian cycle. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists. If the common edge cost is c, the cost of a tour is cn since there are n edges in a Hamiltonian cycle.



```
Algorithm Hamiltonian($k$)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix $G[1:n,1:n]$. All cycles begin at node 1.
{
    repeat
    { // Generate values for $x[k]$.
        NextValue($k$); // Assign a legal next value to $x[k]$.
        if ($x[k] = 0$) then return;
        if ($k = n$) then write ($x[1:n]$);
        else Hamiltonian($k + 1$);
    } until (false);
}
```



```
Algorithm NextValue($k$)
// $x[1:k-1]$ is a path of $k-1$ distinct vertices. If $x[k] = 0$, then
// no vertex has as yet been assigned to $x[k]$. After execution,
// $x[k]$ is assigned to the next highest numbered vertex which
// does not already appear in $x[1:k-1]$ and is connected by
// an edge to $x[k-1]$. Otherwise $x[k] = 0$. If $k = n$, then
// in addition $x[k]$ is connected to $x[1]$.
{
    repeat
    {
        $x[k] := (x[k] + 1)$ mod $(n + 1)$; // Next vertex.
        if ($x[k] = 0$) then return;
        if ($G[x[k-1], x[k]] \neq 0$) then
        { // Is there an edge?
            for $j := 1$ to $k - 1$ do if ($x[j] = x[k]$) then break;
                                // Check for distinctness.
            if ($j = k$) then // If true, then the vertex is distinct.
                if (($k < n$) or (($k = n$) and $G[x[n], x[1]] \neq 0$))
                    then  return;
        }
    } until (false);
}
```

2. Branch and Bound

Recall that the central idea of backtracking, discussed in the previous section, is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution. This idea can be strengthened further if we deal with an optimization problem.

An optimization problem seeks to minimize or maximize some objective function (a tour length, the value of items selected, the cost of an assignment, and the like), usually subject to some constraints. An optimal solution is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).

Compared to backtracking, branch-and-bound requires two additional items:

1. a way to provide, for every node of a state-space tree, **a bound on the best value of the objective function** on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
2. the **value of the best solution** seen so far

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

1. The value of the node's bound is not better than the value of the best solution seen so far.
2. The node represents no feasible solutions because the constraints of the problem are already violated.
3. The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

2.1 Assignment Problem

Let us illustrate the branch-and-bound approach by applying it to the problem of **assigning n people to n jobs so that the total cost of the assignment is as small as possible.**

An instance of the assignment problem is specified by an n × n cost matrix C so that we can state the problem as follows: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible. We will demonstrate how this problem can be solved using the branch-and-bound technique by considering the small instance of the problem. Consider the data given below.

$$
\begin{array}{rccccl}
 & \text{job } 1 & \text{job } 2 & \text{job } 3 & \text{job } 4 & \\
 & 9 & 2 & 7 & 8 & \text{person } a \\
C = & 6 & 4 & 3 & 7 & \text{person } b \\
 & 5 & 8 & 1 & 8 & \text{person } c \\
 & 7 & 6 & 9 & 4 & \text{person } d
\end{array}
$$

How can we find a lower bound on the cost of an optimal selection without actually solving the problem?

We can do this by several methods. For example, it is clear that the **cost of any solution**, including an optimal one, **cannot be smaller than the sum of the smallest elements in each of the matrix's rows**. For the instance here, this sum is 2 + 3+ 1+ 4 = 10. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17.

Rather than generating a single child of the last promising node as we did in backtracking, we will generate all the children of the most promising node among non-terminated leaves in the current tree. (Nonterminated, i.e., still promising, leaves are also called live.) How can we tell which of the nodes is most promising? We can do this by comparing the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the **best-first branch-and-bound**.

We start with the root that corresponds to no elements selected from the cost matrix. The lower-bound value for the root, denoted lb, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person a. See the figure given below.

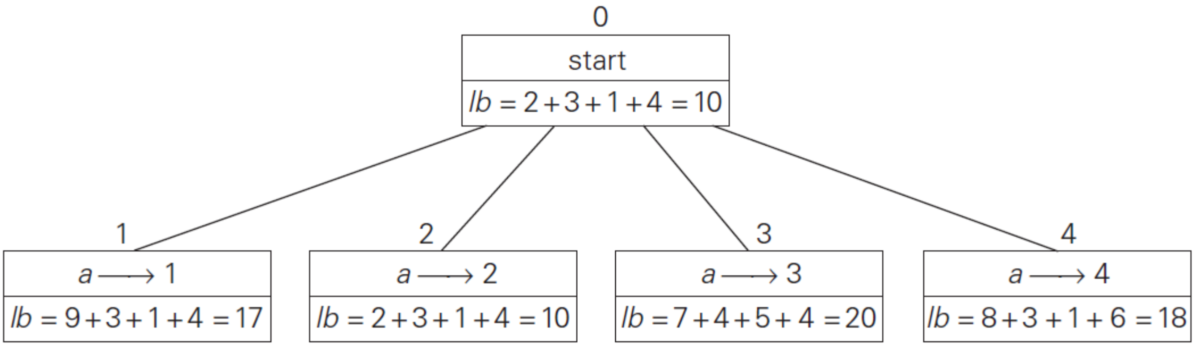


Figure: Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value. Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column - the three different jobs that can be assigned to person b. See the figure given below (Fig 12.7).

Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from c's row (i.e., assigning person c to job 3); this leaves us with no choice but to select the element from the fourth column of d's row (assigning person d to job 4). This yields leaf 8 (Figure 12.7), which corresponds to the feasible solution {a→2, b→1, c→3, d →4} with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution {a→2, b→1, c→4, d →3} with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)

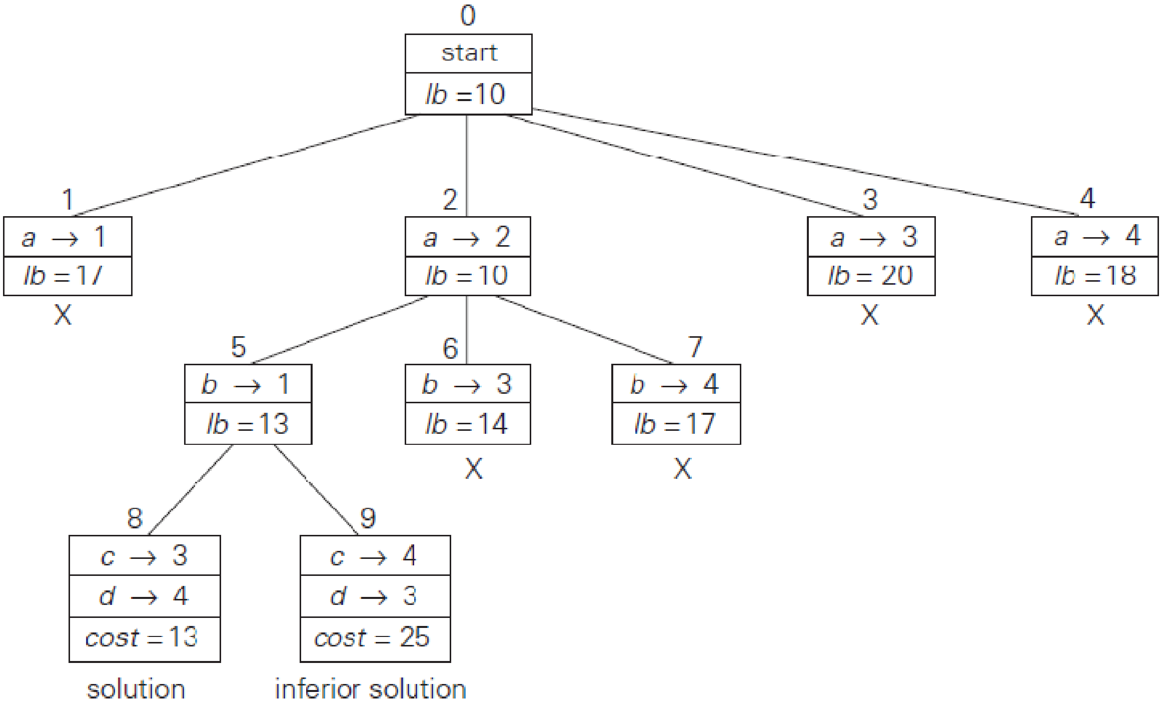


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in Figure 12.7—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

2.2 Travelling Sales Person problem

We will be able to apply the branch-and-bound technique to instances of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths. One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n.

But there is a less obvious and more informative lower bound for instances with symmetric matrix D, which does not require a lot of work to compute. We can compute a lower bound on the length l of any tour as follows. For each city i, $1 \leq i \leq n$, find the sum s_i of the distances from city i to the two nearest cities; compute the sum s of these n numbers, divide the result by 2, and, if all the distances are integers, round up the result to the nearest integer:

$$lb = \lceil s/2 \rceil \qquad \text{... (1)}$$

For example, for the instance in Figure 2.2a, formula (1) yields

$$lb = \lceil[(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2\rceil = 14.$$

Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound (formula 1) accordingly. For example, for all the Hamiltonian circuits of the graph in Figure 2.2a that must include edge (a, d), we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges (a, d) and (d, a):

$$\lceil[(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2\rceil = 16.$$

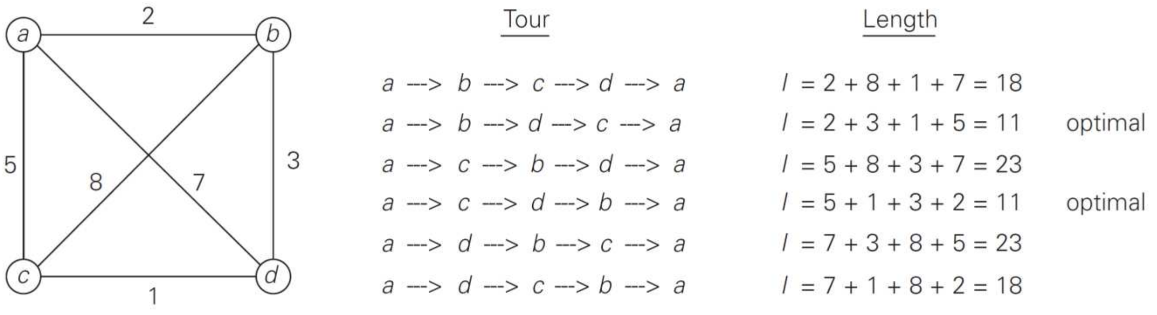
We now apply the branch-and-bound algorithm, with the bounding function given by formula-1, to find the shortest Hamiltonian circuit for the graph in Figure 2.2a.

To reduce the amount of potential work, we take advantage of two observations.

1. First, without loss of generality, we can consider only tours that start at a.
2. Second, because our graph is undirected, we can generate only tours in which b is visited before c. (Refer ***Note*** at the end of section 2.2 for more details)

In addition, after visiting n−1= 4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one. The state-space tree tracing the algorithm's application is given in Figure 2.2b.

Note: An inspection of graph with 4 nodes (figure given below) reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. We could, for example, choose any two intermediate vertices, say, b and c, and then consider only permutations in which b precedes c. (This trick implicitly defines a tour's direction.)



Tour	Length	
a ---> b ---> c ---> d ---> a	l = 2 + 8 + 1 + 7 = 18	
a ---> b ---> d ---> c ---> a	l = 2 + 3 + 1 + 5 = 11	optimal
a ---> c ---> b ---> d ---> a	l = 5 + 8 + 3 + 7 = 23	
a ---> c ---> d ---> b ---> a	l = 5 + 1 + 3 + 2 = 11	optimal
a ---> d ---> b ---> c ---> a	l = 7 + 3 + 8 + 5 = 23	
a ---> d ---> c ---> b ---> a	l = 7 + 1 + 8 + 2 = 18	

Figure: Solution to a small instance of the traveling salesman problem by exhaustive search.

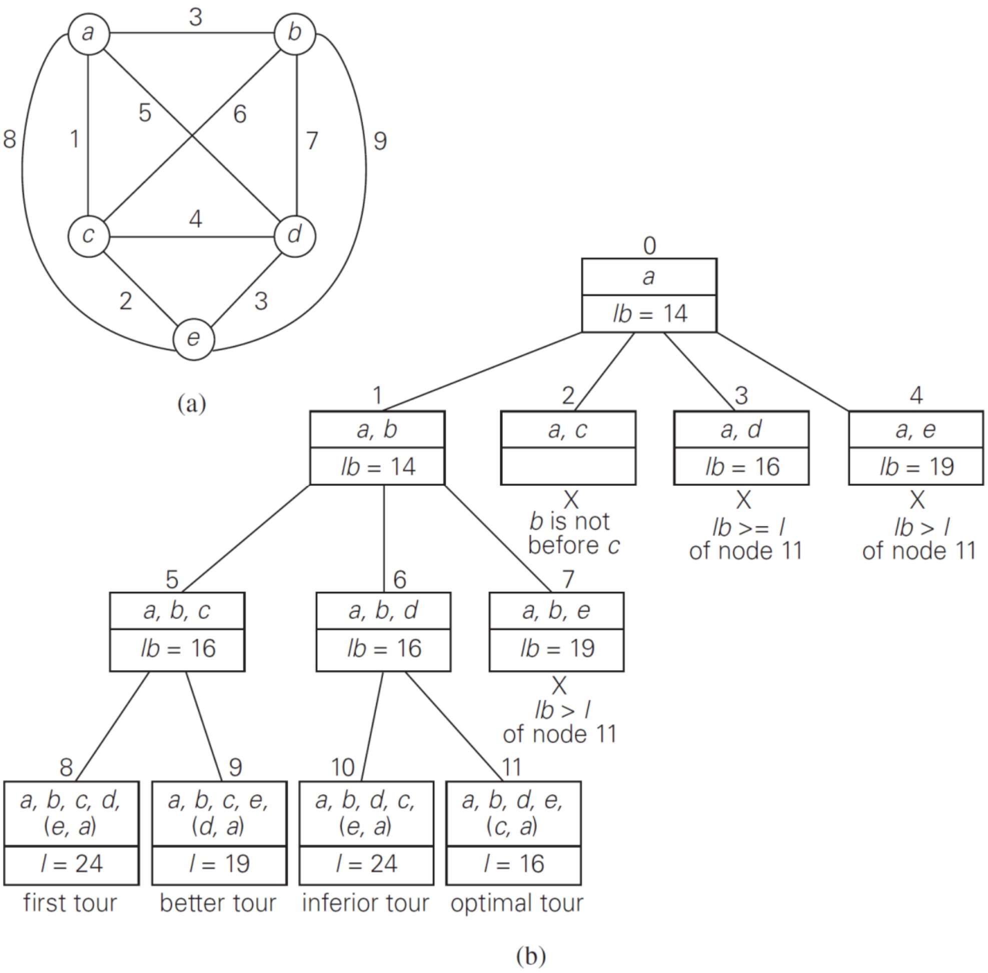


Figure 2.2 (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

Discussion

The strengths and weaknesses of backtracking are applicable to branch-and-bound as well. The state-space tree technique enables us to solve many large instances of difficult combinatorial problems. As a rule, however, it is virtually impossible to predict which instances will be solvable in a realistic amount of time and which will not.

In contrast to backtracking, solving a problem by branch-and-bound has both the challenge and opportunity of choosing the order of node generation and finding a good bounding function. Though the best-first rule we used above is a sensible approach, it may or may not lead to a solution faster than other strategies. (Artificial intelligence researchers are particularly interested in different strategies for developing state-space trees.)

Finding a good bounding function is usually not a simple task. On the one hand, we want this function to be easy to compute. On the other hand, it cannot be too simplistic - otherwise, it would fail in its principal task to prune as many branches of a state-space tree as soon as possible. Striking a proper balance between these two competing requirements may require intensive experimentation with a wide variety of instances of the problem in question.

3. 0/1 Knapsack problem

Note: For this topic as per the syllabus both textbooks T1 & T2 are suggested. Here we discuss the concepts from T1 first and then that of from T2.

Topic form T1 (Levitin)

Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. Given n items of known weights w_i and values v_i , i = 1, 2, . . . , n, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack.

$$\sum_{1\le i\le n} w_i x_i \le W \ and \ \sum_{1\le i\le n} p_i x_i \ \ is\ maximized,\ \ where\ x_i = 0\ or\ 1$$

It is convenient to order the items of a given instance in descending order by their value-to-weight ratios.

$$v_1/w_1 \ge v_2/w_2 \ge \cdots \ge v_n/w_n$$

Each node on the ith level of state space tree, 0 ≤ i ≤ n, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion.

We record the total ***weight w*** and the total ***value v*** of this selection in the node, along with some upper bound ***ub*** on the value of any subset that can be obtained by adding zero or more items to this selection. A simple way to compute the upper bound ***ub*** is to add to ***v***, the total value of the items already selected, the product of the remaining capacity of the knapsack ***W* − *w*** and the best per unit payoff among the remaining items, which is $\mathbf{v_{i+1}/w_{i+1}}$:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$

Example: Consider the following problem. The items are already ordered in descending order of their value-to-weight ratios.

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

Let us apply the branch-and-bound algorithm. At the root of the state-space tree (see Figure 12.8), no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal to 0. The value of the upper bound is 100.

Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 and 40, respectively; the value of the upper bound is 40 + (10 − 4) * 6 = 76.

Node 2 represents the subsets that do not include item 1. Accordingly, w = 0, v = 0, and ub = 0 + (10 − 0) * 6 = 60. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children—nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively. Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately.

Node 4 has the same values of w and v as its parent; the upper bound ***ub*** is equal to 40 + (10 − 4) * 5 = 70. Selecting node 4 over node 2 for the next branching (Due to better ub), we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upper bounds for these nodes are computed in the same way as for the preceding nodes.

Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset {1, 3} of value 65. The remaining live nodes 2 and 6 have smaller upper-bound values than the value of the solution represented by node 8. Hence, both can be terminated making the subset {1, 3} of node 8 the optimal solution to the problem.

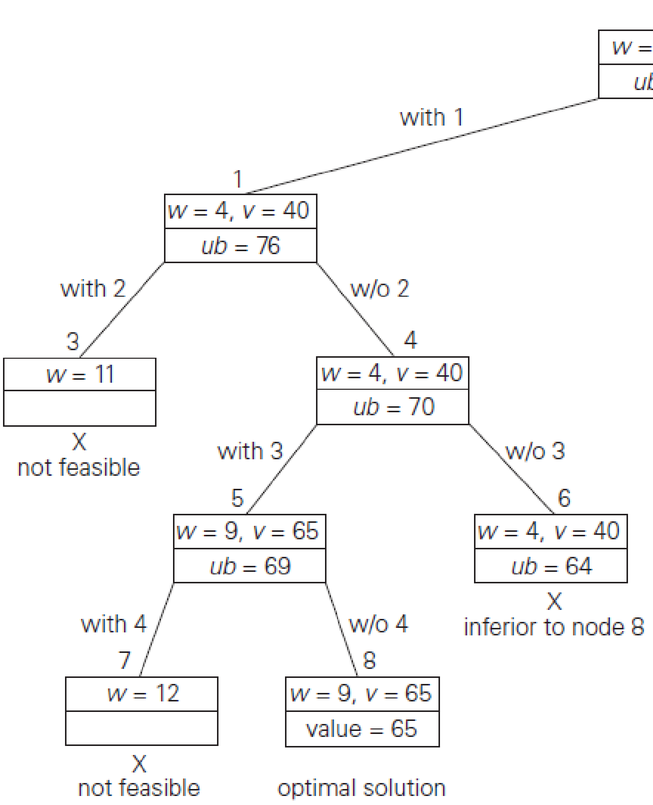


FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic. Typically, internal nodes of a state-space tree do not define a point of the problem's search space, because some of the solution's components remain undefined. (See, for example, the branch-and-bound tree for the assignment problem discussed in the preceding subsection.) For the knapsack problem, however, every node of the tree represents a subset of the items given. We can use this fact to update the information about the best subset seen so far after generating each new node in the tree. If we had done this for the instance investigated above, we could have terminated nodes 2 and 6 before node 8 was generated because they both are inferior to the subset of value 65 of node 5.

Concepts form textbook T2 (Horowitz)

Let us understand some of the **terminologies used in backtracking & branch and bound**.

→ ***Live node*** - a node which has been generated and all of whose children are not yet been generated.

→ ***E-node*** - is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

→ ***Dead node*** - a node that is either not to be expanded further, or for which all of its children have been generated

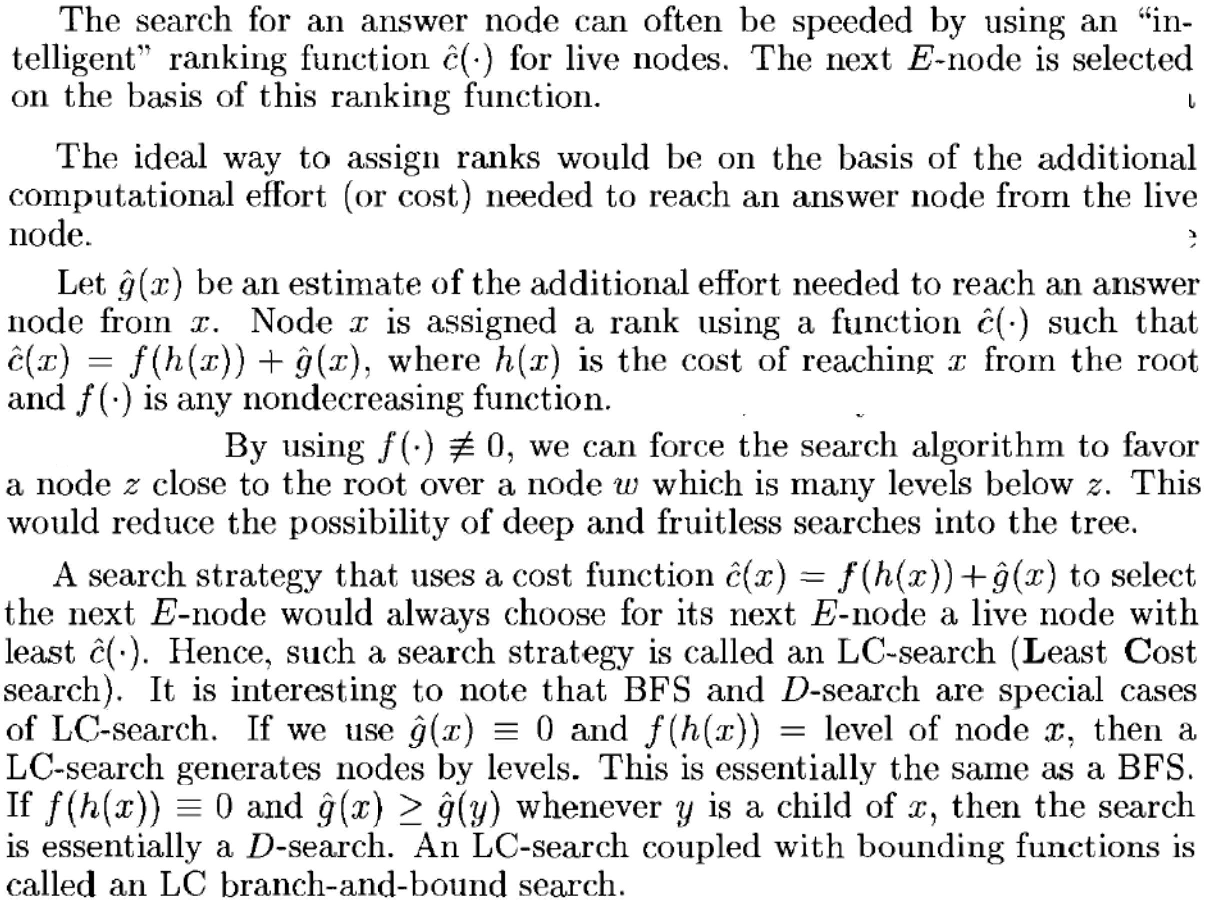
→ ***Bounding Function*** - will be used to kill live nodes without generating all their children.

→ ***Backtracking*** - is depth first node generation with bounding functions.

→ ***Branch-And-Bound*** is a method in which E-node remains E-node until it is dead.

→ ***Breadth-First-Search:*** Branch-and Bound with each new node placed in a queue. The front of the queen becomes the new E-node.

→ ***Depth-Search (D-Search):*** New nodes are placed in to a stack. The last node added is the first to be explored.



The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function.

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(\cdot)$ such that $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, where $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any nondecreasing function.

By using $f(\cdot) \neq 0$, we can force the search algorithm to favor a node z close to the root over a node w which is many levels below z. This would reduce the possibility of deep and fruitless searches into the tree.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search). It is interesting to note that BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) =$ level of node x, then a LC-search generates nodes by levels. This is essentially the same as a BFS. If $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search. An LC-search coupled with bounding functions is called an LC branch-and-bound search.

0/1 Knapsack problem - Branch and Bound based solution

As the technique discussed here is applicable for minimization problems, let us convert the knapsack problem (maximizing the profit) into minimization problem by negating the objective function

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i \quad \text{subject to } \sum_{i=1}^{n} w_i x_i \le m \qquad x_i = 0 \text{ or } 1, \quad 1 \le i \le n$$

Every leaf node in the state space tree representing an assignment for which $\sum_{1\le i\le n} w_i x_i \le m$ is an answer (or solution) node. All other leaf nodes are infeasible. For a minimum-cost answer node to correspond to any optimal solution, we need to define $c(x) = -\sum_{1<i<n} p_i x_i$ for every answer node x. The cost $c(x) = \infty$ for infeasible leaf nodes. For nonleaf nodes, $c(x)$ is recursively defined to be min $\{c(lchild(x)),\ c(rchild(x))\}$.

We now need two functions $\hat{c}(x)$ and $u(x)$ such that $\hat{c}(x) \le c(x) \le u(x)$ for every node x. The cost $\hat{c}(\cdot)$ and $u(\cdot)$ satisfying this requirement may be obtained as follows. Let x be a node at level j, $1 \le j \le n+1$. At node x assignments have already been made to x_i, $1 \le i < j$. The cost of these assignments is $-\sum_{1\le i<j} p_i x_i$. So, $c(x) \le -\sum_{1\le i<j} p_i x_i$ and we may use $u(x) = -\sum_{1\le i<j} p_i x_i$. If $q = -\sum_{1\le i<j} p_i x_i$, then an improved upper bound function $u(x)$ is $u(x) = \text{UBound}(q, \sum_{1\le i<j} w_i x_i, j-1, m)$, where UBound is defined in Algorithm 8.2.

Algorithm 8.2 Function $u(\cdot)$ for knapsack problem

```
Algorithm UBound(cp, cw, k, m)
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
//                  w[i] and p[i] are respectively
// the weight and profit of the ith object.
{
    b := cp; c := cw;
    for i := k + 1 to n do
    {
        if (c + w[i] ≤ m) then
        {
            c := c + w[i]; b := b − p[i];
        }
    }
    return b;
}
```

3.1 LC (Least Cost) Branch and Bound solution

To use LCBB to solve the knapsack problem, we need to specify (1) the structure of nodes in the state space tree being searched, (2) how to generate the children of a given node, (3) how to recognize a solution node, and (4) a representation of the list of live nodes and a mechanism for adding a node into the list as well as identifying the least-cost node. The node structure needed depends on which of the two formulations for the state space tree is being used. Let us continue with a fixed size tuple formulation. Each node x that is generated and put onto the list of live nodes must have a *parent* field. In addition, as noted in Example 8.2, each node should have a one bit *tag* field. This field is needed to output the x_i values corresponding to an optimal solution. To generate x's children, we need to know the level of node x in the state space tree. For this we shall use a field *level*. The left child of x is chosen by setting $x_{level(x)} = 1$ and the right child by setting $x_{level(x)} = 0$. To determine the feasibility of the left child, we need to know the amount of knapsack space available at node x. This can be determined either by following the path from node x to the root or by explicitly retaining this value in the node. Say we choose to retain this value in a field *cu* (capacity unused). The evaluation of $\hat{c}(x)$ and $u(x)$ requires knowledge of the profit $\sum_{1 \leq i < level(x)} p_i x_i$ earned by the filling corresponding to node x. This can be computed by following the path from x to the root. Alternatively, this value can be explicitly retained in a field *pe*. Finally, in order to determine the live node with least $\hat{c}$ value or to insert nodes properly into the list of live nodes, we need to know $\hat{c}(x)$. Again, we have a choice. The value $\hat{c}(x)$ may be stored explicitly in a field *ub* or may be computed when needed. Assuming all information is kept explicitly, we need nodes with six fields each: *parent*, *level*, *tag*, *cu*, *pe*, and *ub*.

Using this six-field node structure, the children of any live node x can be easily determined. The left child y is feasible iff $cu(x) \geq w_{level(x)}$. In this case, $parent(y) = x$, $level(y) = level(x) + 1$, $cu(y) = cu(x) - w_{level(x)}$, $pe(y) = pe(x) + p_{level(x)}$, $tag(y) = 1$, and $ub(y) = ub(x)$. The right child can be generated similarly. Solution nodes are easily recognized too. Node x is a solution node iff $level(x) = n + 1$.

We are now left with the task of specifying the representation of the list of live nodes. The functions we wish to perform on this list are (1) test if the list is empty, (2) add nodes, and (3) delete a node with least *ub*. We have seen a data structure that allows us to perform these three functions efficiently: a min-heap. If there are m live nodes, then function (1) can be carried out in $\Theta(1)$ time, whereas functions (2) and (3) require only $O(\log m)$ time.

Example 8.2 [LCBB] Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$. Let us trace the working of an LC branch-and-bound search using $\hat{c}(\cdot)$ and $u(\cdot)$ as defined previously. We continue to use the fixed tuple size formulation. The search begins with the root as the E-node. For this node, node 1 of Figure 8.8, we have $\hat{c}(1) = -38$ and $u(1) = -32$.

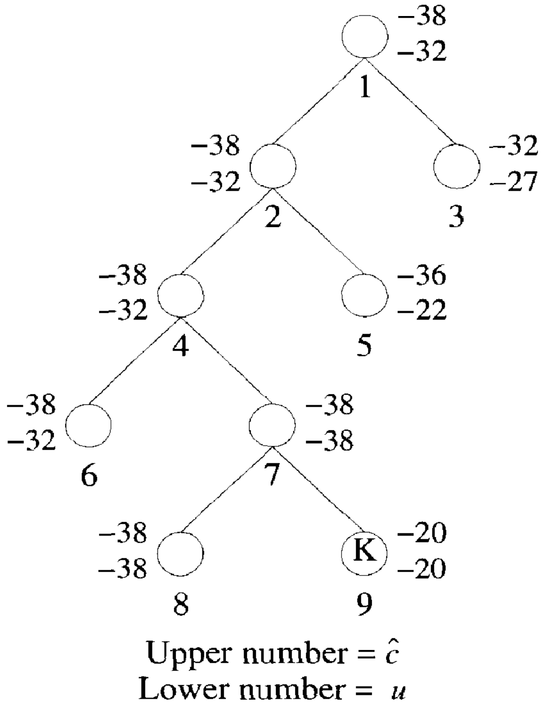


Figure 8.8 LC branch-and-bound tree for Example 8.2

The computation of $u(1)$ and $\hat{c}(1)$ is done as follows. The bound $u(1)$ has a value UBound$(0, 0, 0, 15)$. UBound scans through the objects from left to right starting from j; it adds these objects into the knapsack until the first object that doesn't fit is encountered. At this time, the negation of the total profit of all the objects in the knapsack plus cw is returned. In Function UBound, c and b start with a value of zero. For $i = 1, 2$, and 3, c gets incremented by $2, 4$, and 6, respectively. The variable b also gets decremented by $10, 10$, and 12, respectively. When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is -32. Function Bound is similar to UBound, except that it also considers a fraction of the first object that doesn't fit the knapsack. For example, in computing $\hat{c}(1)$, the first object that doesn't fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction $\frac{3}{9}$ of the object 4 and hence returns $-32 - \frac{3}{9} * 18 = -38$.

Since node 1 is not a solution node, LCBB sets $ans = 0$ and $upper = -32$ (ans being a variable to store intermediate answer nodes). The E-node is expanded and its two children, nodes 2 and 3, generated. The cost $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$, and $u(3) = -27$. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node

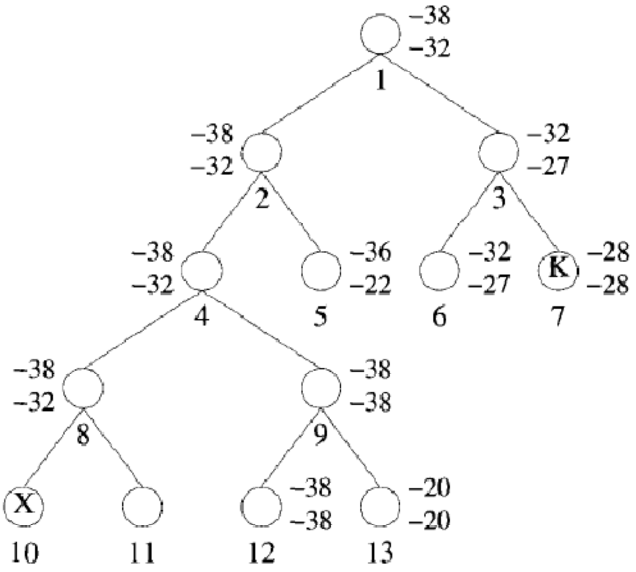
4 is the live node with least $\hat{c}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and *upper* is updated to -38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then *upper* is updated to -38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) > upper$ and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq upper$ and the search terminates with node 8 the answer node. At this time, the value -38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\sum p_i x_i = upper$. Hence, a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, *tag*. The sequence of *tag* bits from the answer node to the root give the x_i values. Thus, we have $tag(2) = tag(4) = tag(6) = tag(8) = 1$ and $tag(3) = tag(5) = tag(7) = tag(9) = 0$. The *tag* sequence for the path 8, 7, 4, 2, 1 is 1 0 1 1 and so $x_4 = 1, x_3 = 0, x_2 = 1$, and $x_1 = 1$. □

3.2 FIFO Branch and Bound solution

In branch-and-bound terminology, a BFS-like state space search will be called FIFO (**F**irst **I**n **F**irst **O**ut) search as the list of live nodes is a first-in-first-out list (or queue).

Example 8.3 Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in Example 8.2. Initially the root node, node 1 of Figure 8.9, is the E-node and the queue of live nodes is empty. Since this is not a solution node, *upper* is initialized to $u(1) = -32$.

We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of *upper* remains unchanged. Node 2 becomes the next E-node. Its children, nodes 4 and 5, are generated and added to the queue. Node 3, the next E-node, is expanded. Its children nodes are generated. Node 6 gets added to the queue. Node 7 is immediately killed as $\hat{c}(7) > upper$. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then *upper* is updated to $u(9) = -38$. Nodes 5 and 6 are the next two nodes to become E-nodes. Neither is expanded as for each, $\hat{c}() > upper$. Node 8 is the next E-node. Nodes 10 and 11 are generated. Node 10 is infeasible and so killed. Node 11 has $\hat{c}(11) > upper$ and so is also killed. Node 9 is expanded next. When node 12 is generated, *upper* and *ans* are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as $\hat{c}(13) > upper$. The only remaining live node is node 12. It has no children and the search terminates. The value of *upper* and the path from node 12 to the root is output. As in the case of Example 8.2, additional information is needed to determine the x_i values on this path. □



upper number = $\hat{c}$
lower number = u

Figure 8.9 FIFO branch-and-bound tree for Example 8.3

Conclusion

At first we may be tempted to discard FIFOBB in favor of LCBB in solving knapsack. Our intuition leads us to believe that LCBB will examine fewer nodes in its quest for an optimal solution. However, we should keep in mind that insertions into and deletions form a heap are far more expensive (proportional to the logarithm of the heap size) than the corresponding operations on a queue ($\Theta(1)$). Consequently, the work done for each E-node is more in LCBB than in FIFOBB. Unless LCBB uses far fewer E-nodes than FIFOBB, FIFOBB will outperform (in terms of real computation time) LCBB.

4. NP-Complete and NP-Hard problems

4.1 Basic concepts

For many of the problems we know and study, the best algorithms for their solution have computing times can be clustered into two groups;

1. Solutions are bounded by the **polynomial**- Examples include Binary search O(log n), Linear search O(n), sorting algorithms like merge sort O(n log n), Bubble sort $O(n^2)$ & matrix multiplication $O(n^3)$ or in general $O(n^k)$ where k is a constant.
2. Solutions are bounded by a non-polynomial - Examples include travelling salesman problem $O(n^2 2^n)$ & knapsack problem $O(2^{n/2})$. As the time increases exponentially, even moderate size problems cannot be solved.

So far, no one has been able to device an algorithm which is bounded by the polynomial for the problems belonging to the non-polynomial. However impossibility of such an algorithm is not proved.

4.2 Non deterministic algorithms

We also need the idea of two models of computer (Turing machine): deterministic and non-deterministic. A deterministic computer is the regular computer we always thinking of; a non-deterministic computer is one that is just like we're used to except that is has unlimited parallelism, so that any time you come to a branch, you spawn a new "process" and examine both sides.

When the result of every operation is uniquely defined then it is called **deterministic algorithm**.

When the outcome is not uniquely defined but is limited to a specific set of possibilities, we call it **non deterministic** algorithm.

We use new statements to specify such **non deterministic** h algorithms.

- **choice(S) -** arbitrarily choose one of the elements of set S
- **failure -** signals an unsuccessful completion
- **success -** signals a successful completion

The assignment ***X = choice(1:n)*** could result in X being assigned any value from the integer ***range[1..n]***. There is no rule specifying how this value is chosen.

"The nondeterministic algorithms terminates unsuccessfully iff there is no set of choices which leads to the successful signal".

Example-1: Searching an element **x** in a given set of elements A(1:n). We are required to determine an index j such that A(j) = x or j = 0 if x is not present.

```
j := choice(1:n)
if A(j) = x then print(j); success endif
```

```
print('0'); failure
```

Example-2: Checking whether n integers are sorted or not

```
procedure NSORT(A,n);
//sort n positive integers//
var integer A(n), B(n), n, i, j;
begin
        B := 0; //B is initialized to zero//
        for i := 1 to n do
        begin
                j := choice(1:n);
                if B(j) <> 0 then failure;
                B(j) := A(j);
        end;

        for i := 1 to n-1 do //verify order//
                if B(i) > B(i+1) then failure;
        print(B);
        success;
end.
```

"A nondeterministic machine does not make any copies of an algorithm every time a choice is to be made. Instead it has the ability to correctly choose an element from the given set".

A deterministic interpretation of the nondeterministic algorithm can be done by making unbounded parallelism in the computation. Each time a choice is to be made, the algorithm makes several copies of itself, one copy is made for each of the possible choices.

Decision vs Optimization algorithms

An optimization problem tries to find an optimal solution.

A decision problem tries to answer a yes/no question. Most of the problems can be specified in decision and optimization versions.

For example, Traveling salesman problem can be stated as two ways

- Optimization - find hamiltonian cycle of minimum weight,
- Decision - is there a hamiltonian cycle of weight ≤ k?

For graph coloring problem,

- Optimization – find the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color
- Decision - whether there exists such a coloring of the graph's vertices with no more than m colors?

Many optimization problems can be recast in to decision problems with the property that the decision algorithm can be solved in polynomial time if and only if the corresponding optimization problem.

4.3 P, NP, NP-Complete and NP-Hard classes

NP stands for Non-deterministic Polynomial time.

Definition: P is a set of all decision problems solvable by a deterministic algorithm in polynomial time.

Definition: NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time. This also implies P ⊆ NP

Problems known to be in P are trivially in NP — the nondeterministic machine just never troubles itself to fork another process, and acts just like a deterministic one. One example of a problem not in P but in NP is ***Integer Factorization***.

But there are some problems which are known to be in NP but don't know if they're in P. The traditional example is the decision-problem version of the Travelling Salesman Problem (***decision-TSP***). It's not known whether decision-TSP is in P: there's no known poly-time solution, but there's no proof such a solution doesn't exist.

There are problems that are known to be neither in P nor NP; a simple example is to enumerate all the bit vectors of length n. No matter what, that takes 2^n steps.

Now, one more concept: given decision problems P and Q, if an algorithm can transform a solution for P into a solution for Q in polynomial time, it's said that Q is ***poly-time reducible*** (or just reducible) to P.

The most famous unsolved problem in computer science is "whether P=NP or P≠NP? "

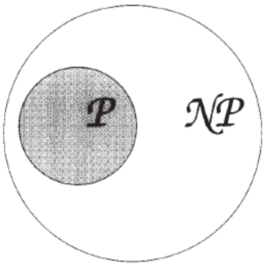


Figure: Commonly believed relationship between P and NP

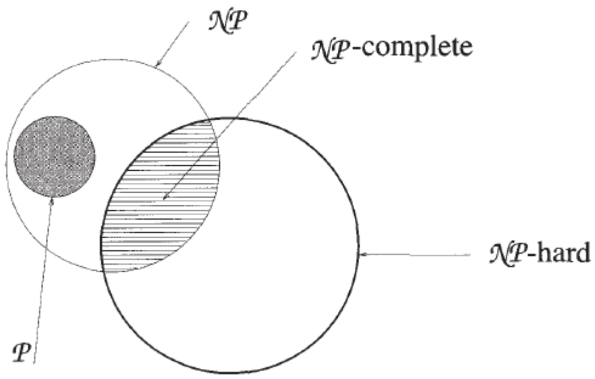


Figure: Commonly believed relationship between P, NP, NP-Complete and NP-hard problems

Definition: A decision problem D is said to be **NP-complete** if:

1. it belongs to class NP
2. every problem in NP is polynomially reducible to D

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

NP-Complete problems have the property that it can be solved in polynomial time if all other NP-Complete problems can be solved in polynomial time. i.e if anyone ever finds a poly-time solution to one NP-complete problem, they've automatically got one for ***all*** the NP-complete problems; that will also mean that P=NP.

Example for NP-complete is **CNF-satisfiability problem**. The CNF-satisfiability problem deals with boolean expressions. This is given by Cook in 1971. The CNF-satisfiability problem asks whether or not one can assign values true and false to variables of a given boolean expression in its CNF form to make the entire expression true.

Over the years many problems in NP have been proved to be in P (like Primality Testing). Still, there are many problems in NP not proved to be in P. i.e. the question still remains whether P=NP? NP Complete Problems helps in solving this question. They are a subset of NP problems with the property that all other NP problems can be reduced to any of them in polynomial time. So, they are the hardest problems in NP, in terms of running time. If it can be showed that any NP-Complete problem is in P, then all problems in NP will be in P (because of NP-Complete definition), and hence P=NP=NPC.

NP Hard Problems - These problems need not have any bound on their running time. If any NP-Complete Problem is polynomial time reducible to a problem X, that problem X belongs to NP-Hard class. Hence, all NP-Complete problems are also NP-Hard. In other words if a NP-Hard problem is non-deterministic polynomial time solvable, it is a NP-Complete problem. Example of a NP problem that is not NPC is Halting Problem.

If a NP-Hard problem can be solved in polynomial time then all NP-Complete can be solved in polynomial time.

"All NP-Complete problems are NP-Hard but not all NP-Hard problems are not NP-Complete." NP-Complete problems are subclass of NP-Hard

The more conventional optimization version of Traveling Salesman Problem for finding the shortest route is NP-hard, not strictly NP-complete.

*****