A Course Material on

Analysis and Design of Algorithm

By

Mr. S.ANBARASU

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SASURIE COLLEGE OF ENGINEERING

VIJAYAMANGALAM – 638 056

QUALITY CERTIFICATE

This is to certify that the e-course material

Subject Code : **CS6402**

Subject : Design and Analysis of Algorithm

Class : II Year CSE

being prepared by me and it meets the knowledge requirement of the university curriculum.

Signature of the Author

Name: S.Anbarasu

Designation: AP

This is to certify that the course material being prepared by Mr. S. Anbarasu is of adequate quality. He has referred more than five books among them minimum one is from abroad author.

Signature of HD

Name: P.Muruga Priya

SEAL

S.No.	Contents	Page No.
	UNIT – I - Introduction	
1.1	Introduction	5
1.2	Fundamentals of Analysis of Algorithm	6
1.2.1	Analysis of Framework	6
1.2.2	Measuring an input size	6
1.2.3	Units for measuring runtime	7
1.2.4	Worst case, Best case and Average case	7
1.2.5	Asymptotic Notations	10
1.3	An Example	15
1.4	Analysis of Linear Search	9
	Questions	22
	Total Hours 9	
	UNIT – II – Divide and Conquer, Greedy Method	
2.1	Divide and Conquer	23
2.2	General Method – Divide and Conquer	24
2.2.1	Master theorem	24
2.2.2	Efficiency of Merge sort	27
2.3	Binary Search	28
2.3.1	Efficiency of Binary Search	30
2.4	Greedy Techniques	32
2.5	General Method – Greedy Method	32
2.5.1	Prim‘s Algorithm	32
2.6	Container Loading	34
2.6.1	Algorithm for Container Loading	37
2.7	Knapsack Problem	37
	Questions	39
	Total Hours 9	
	UNIT – III – Dynamic Programming	
3.1	Introduction	41
3.2	General Method – Dynamic Programming	41
3.2.1	Pseudo code for Binomial Coefficient	41
3.3	Multistage Graphs	42
3.3.1	Pseudo code for Floyd‘s Algorithm	43
3.4	Optimal Binary Search trees	45
3.5	0/1 Knapsack Problem	49
3.6	Traveling Salesman Problem	50
	Questions	53
	Total Hours 9	
	UNIT – IV - Backtracking	
4.1	Backtracking	55
4.1.1	General Method – Backtracking	55

4.1.2	State-Space tree	55
4.2	N-queens Problem	55
4.3	Subset- Sum problem	57
4.3.1	Pseudo code for Backtracking Algorithms	58
4.4	Graph Coloring	59
4.5	Hamiltonian circuit problem	61
4.6	Knapsack Problem	63
	Questions	65
	Total Hours 9	
	UNIT – V – Traversals, Branch & Bound	
5.1	Graph Traversals, BFS	66
5.1.1	BFS Forest	66
5.1.2	Efficiency	67
5.1.3	Ordering of Vertices	67
5.1.4	Application of BFS	67
5.2	DFS	67
5.2.1	DFS Forest	68
5.2.2	DFS Algorithm	69
5.1.3	Application of DFS	70
5.3	Connected Component	70
5.3.1	Biconnected Components	71
5.4	Spanning Trees	73
5.4.1	Prim‘s Algorithm	73
5.4.2	Kruskal‘s Algorithm	76
5.5	Branch & Bound	78
5.6	General Methods – FIFO & LC	79
5.7	Knapsack Problem	85
5.8	NP Hard & NP Completeness	86
	Questions	88
	Total Hours 9	

UNIT - I ALGORITHM ANALYSIS

Algorithm analysis – Time space tradeoff – Asymptotic notations – Conditional asymptotic notation – Removing condition from the conditional asymptotic notation – Properties of Big-oh notation – Recurrence equations – Solving recurrence equations – Analysis of linear search.

1.1 Introduction

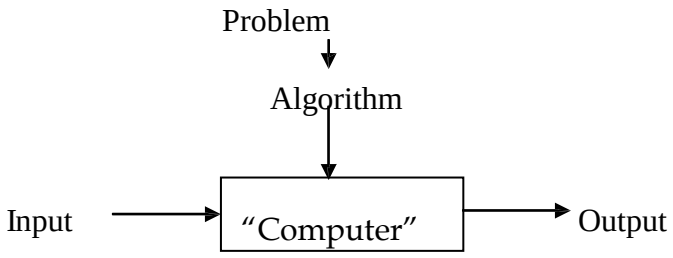
An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.

Definition

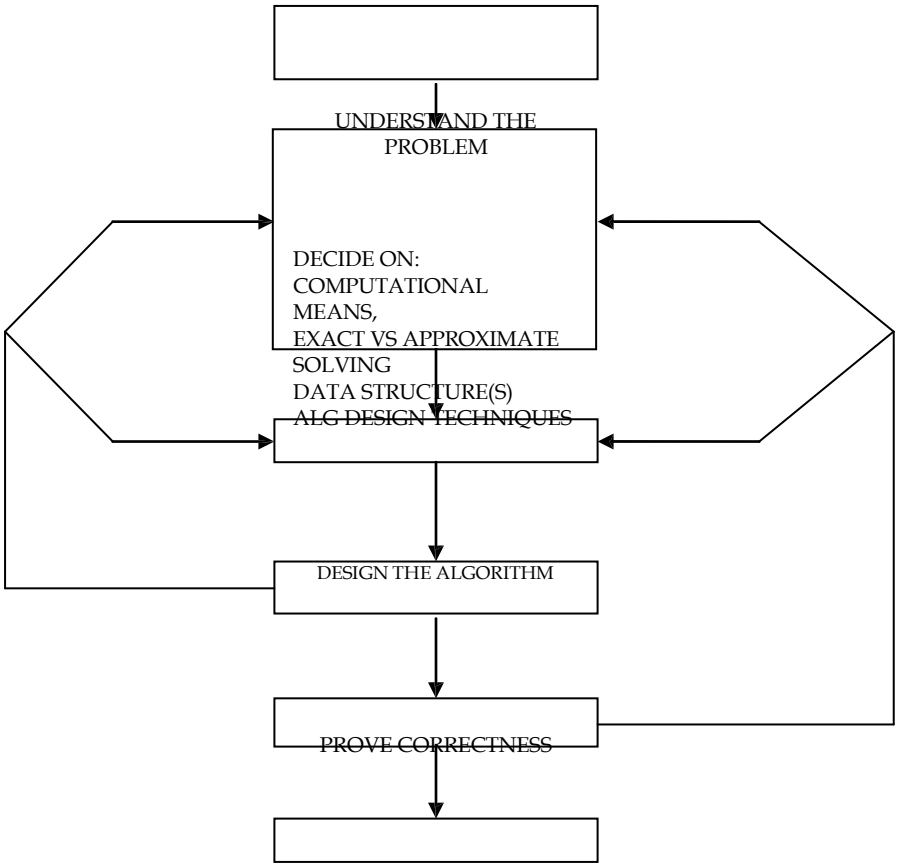
"Algorithmic is more than the branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant it most of science, business and technology"

Understanding of Algorithm

An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.



ALGORITHM DESIGN AND ANALYSIS PROCESS



1.2 FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

1.2.1 ANALYSIS FRAME WORK

♦ there are two kinds of efficiency

- ♦ **Time efficiency** - indicates how fast an algorithm in question runs.
- ♦ **Space efficiency** - deals with the extra space the algorithm requires.

1.2.2 MEASURING AN INPUT SIZE

♦ An algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

- In most cases, selecting such a parameter is quite straightforward.
- For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.
- For the problem of evaluating a polynomial $\mathbf{p(x) = a_n x^n + \ldots + a_0}$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

- There are situations, of course, where the choice of a parameter indicating an input size does matter.
  - **Example** - computing the product of two n-by-n matrices.
  - There are two natural measures of size for this problem.
    - The matrix order n.
    - The total number of elements N in the matrices being multiplied.
- Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.
- 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 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 efficiency of algorithms in question.

1.2.3 UNITS FOR MEASURING RUN TIME:

- We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.
- There are obvious drawbacks to such an approach. They are
- Dependence on the speed of a particular computer
- Dependence on the quality of a program implementing the algorithm
- The compiler used in generating the machine code
- The difficulty of clocking the actual running time of the program.
- Since we are in need to measure algorithm efficiency, we should 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 difficult and unnecessary.
- The main objective 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.

WORST CASE, BEST CASE AND AVERAGE CASE EFFICIENCES

- It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.
- But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.
- **Example, sequential search**. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.
- Here is the algorithm's pseudo code, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition A[i] i= K will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

ALGORITHM Sequential Search(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: Returns the index of the first element of A that matches K
//          or -1 ifthere 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
```

- Clearly, the running time of this algorithm can be quite different for the same list size n.

Worst case efficiency

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

$$\mathbf{C_{worst}(n) = n.}$$

- The way to determine is quite straightforward
- To analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst-case value $\mathbf{C_{worst}(n)}$
- The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $\mathbf{C_{worst}(n)}$ its running time on the worst-case inputs.

Best case Efficiency

- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.
- We can analyze the best case efficiency as follows.
- First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)
- Then ascertain the value of C (n) on these most convenient inputs.
- Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, $C_{best}(n) = 1$.
- The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency.
- But it is not completely useless. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast.
- Thus, such an algorithm might well be the method of choice for applications dealing with almost sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

Average case efficiency

- It yields the information about an algorithm about an algorithm‘s behaviour on a ―typical‖ and ―random‖ input.
- To analyze the algorithm's average-case efficiency, we must make some assumptions about possible i*n*puts of size n.
- The investigation of the average case efficiency is considerably more difficult than investigation of the worst case and best case efficiency.
- It involves dividing all instances of size n .into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same.
- Then a probability distribution of inputs needs to be obtained or assumed so that the expected value of the basic operation's count can then be derived.

The average number of key comparisions $C_{avg}(n)$ can be computed as follows,

- let us consider again sequential search. The standard assumptions are,
- In the case of a successful search, the probability of the first match occurring in the ith position of the list is pin 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,

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \ldots + i \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n} \right] + n \cdot (1 - p)$$

$$= \frac{p}{n} [1 + 2 + 3 + \ldots + i + \ldots + n] + n (1 - p)$$

$$= \frac{p}{n} \frac{n(n+1)}{2} + n (1 - p)$$

$$= \frac{p (n + 1)}{2} + n (1 - p)$$

- Example, if p = 1 (i.e., the search must be successful), the average number of key comparisons made by sequential search is (n + 1) /2.
- If p = 0 (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

1.2.5 Asymptotic Notations

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Because the values are not exact quantities. We need only comparative statements like $c_1n^2 \leq t_p(n) \leq c_2n^2$.

For example, consider two programs with complexities $c_1n^2 + c_2n$ and c_3n respectively. For small values of n, complexity depend upon values of c_1, c_2 and c_3. But there will also be an n beyond which complexity of c_3n is better than that of $c_1n^2 + c_2n$.This value of n is called break-even point. If this point is zero, c_3n is always faster (or at least as fast). Common asymptotic functions are given below.

Function	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	n log n
n^2	Quadratic
n^3	Cubic

2^n	Exponential
n!	Factorial

Big'Oh'Notation(O)

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as

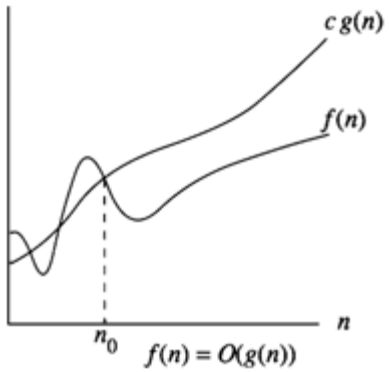


Fig 1.1

Find the Big _Oh' for the following functions:

Linear Functions

Example 1.6
f(n) = 3n + 2

General form is f(n) ≤ cg(n)

When n ≥ 2, 3n + 2 ≤ 3n + n = 4n
Hence f(n) = O(n), here c = 4 and n_0 = 2

When n ≥ 1, 3n + 2 ≤ 3n + 2n = 5n
Hence f(n) = O(n), here c = 5 and n_0 = 1

Hence we can have different c,n_0 pairs satisfying for a given function.

Example
f(n) = 3n + 3
When n ≥ 3, 3n + 3 ≤ 3n + n = 4n
Hence f(n) = O(n), here c = 4 and n_0 = 3

Example
f(n) = 100n + 6
When n ≥ 6, 100n + 6 ≤ 100n + n = 101n
Hence f(n) = O(n), here c = 101 and n_0 = 6

Quadratic Functions

Example 1.9

$f(n) = 10n^2 + 4n + 2$

When $n \geq 2$, $10n^2 + 4n + 2 \leq 10n^2 + 5n$

When $n \geq 5$, $5n \leq n^2$, $10n^2 + 4n + 2 \leq 10n^2 + n^2 = 11n^2$

Hence $f(n) = O(n^2)$, here $c = 11$ and $n_0 = 5$

Example 1.10

$f(n) = 1000n^2 + 100n - 6$

$f(n) \leq 1000n^2 + 100n$ for all values of n.

When $n \geq 100$, $5n \leq n^2$, $f(n) \leq 1000n^2 + n^2 = 1001n^2$

Hence $f(n) = O(n^2)$, here $c = 1001$ and $n_0 = 100$

Exponential Functions

Example 1.11

$f(n) = 6*2^n + n^2$

When $n \geq 4$, $n^2 \leq 2^n$

So $f(n) \leq 6*2^n + 2^n = 7*2^n$

Hence $f(n) = O(2^n)$, here $c = 7$ and $n_0 = 4$

Constant Functions

Example 1.12

$f(n) = 10$

$f(n) = O(1)$, because $f(n) \leq 10*1$

OmegaNotation(Ω)

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as

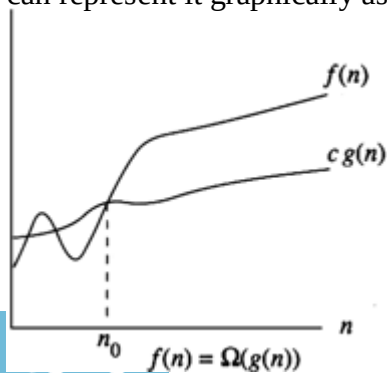


Fig 1.2

Example 1.13
f(n) = 3n + 2
3n + 2 > 3n for all n.
Hence f(n) = Ω(n)

Similarly we can solve all the examples specified under Big _Oh'.

ThetaNotation(Θ)

Θ(g(n)) = {f(n) : there exist positive constants c_1,c_2 and n_0 such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$ }
If f(n) = Θ(g(n)), all values of n right to n_0 f(n) lies on or above $c_1g(n)$ and on or below $c_2g(n)$. Hence it is asymptotic tight bound for f(n).

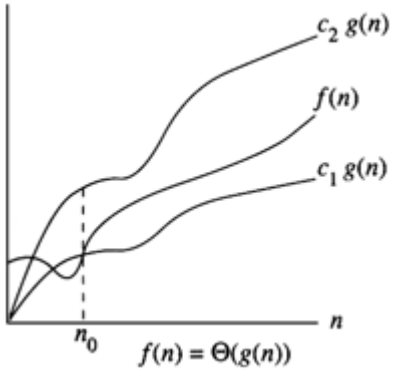


Fig 1.3

.

Little-O Notation

For non-negative functions, f(n) and g(n), f(n) is little o of g(n) if and only if f(n) = O(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = o(g(n))".

This represents a loose bounding version of Big O. g(n) bounds from the top, but it does not bound the bottom.

Little Omega Notation

For non-negative functions, f(n) and g(n), f(n) is little omega of g(n) if and only if f(n) = Ω(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = ω(g(n))".

Much like Little Oh, this is the equivalent for Big Omega. g(n) is a loose lower boundary of the function f(n); it bounds from the bottom, but not from the top.

Conditional asymptotic notation

Many algorithms are easier to analyse if initially we restrict our attention to instances whose size satisfies a certain condition, such as being a power of 2. Consider, for example, the divide and conquer algorithm for multiplying large integers that we saw in the Introduction. Let *n* be the size of the integers to be multiplied.

The algorithm proceeds directly if *n = 1*, which requires *a* microseconds for an appropriate constant *a*. If *n>1*, the algorithm proceeds by multiplying four pairs of integers of size □n/2□ (or three if we use the better algorithm). Moreover, it takes a linear amount of time to carry out additional tasks. For simplicity, let us say that the additional work takes at most *bn* microseconds for an appropriate constant *b*.

Properties of Big-Oh Notation

Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. For example, if you want to write a function that searches through an array of numbers and returns the smallest one:

```
function find-min(array a[1..n])
  let j :=
  for i := 1 to n:
    j := min(j, a[i])
  repeat
return j
end
```

Regardless of how big or small the array is, every time we run find-min, we have to initialize the i and j integer variables and return j at the end. Therefore, we can just think of those parts of the function as constant and ignore them.

So, how can we use asymptotic notation to discuss the find-min function? If we search through an array with 87 elements, then the for loop iterates 87 times, even if the very first element we hit turns out to be the minimum. Likewise, for n elements, the for loop iterates n times. Therefore we say the function runs in time O(n).

```
function find-min-plus-max(array a[1..n])
  // First, find the smallest element in the array
  let j := ;
  for i := 1 to n:
    j := min(j, a[i])
  repeat
  let minim := j

  // Now, find the biggest element, add it to the smallest and
  j := ;
  for i := 1 to n:
    j := max(j, a[i])
  repeat

  let maxim := j

  // return the sum of the two
  return minim + maxim;
end
```

What's the running time for find-min-plus-max? There are two for loops, that each iterate n times, so the running time is clearly O(2n). Because 2 is a constant, we throw it away and write the running time as O(n). Why can you do this? If you recall the definition of Big-O notation, the function whose bound you're testing can be multiplied by some constant. If f(x)=2x, we can see that if g(x) = x, then the Big-O condition holds. Thus O(2n) = O(n). This rule is general for the various asymptotic notations.

Recurrence

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence

Recurrence Equation

A recurrence relation is an equation that recursively defines a sequence. Each term of the sequence is defined as a function of the preceding terms. A difference equation is a specific type of recurrence relation.

An example of a recurrence relation is the logistic map:

$$x_{n+1} = r x_n (1 - x_n)$$

1.3 Another Example: Fibonacci numbers

The Fibonacci numbers are defined using the linear recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values:

$$F_0 = 0$$
$$F_1 = 1$$

Explicitly, recurrence yields the equations:

$$F_2 = F_1 + F_0$$
$$F_3 = F_2 + F_1$$
$$F_4 = F_3 + F_2$$

We obtain the sequence of Fibonacci numbers which begins:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

It can be solved by methods described below yielding the closed form expression which involve powers of the two roots of the characteristic polynomial $t^2 = t + 1$; the generating function of the sequence is the rational function $t / (1 - t - t^2)$.

Solving Recurrence Equation

i. substitution method

The substitution method for solving recurrences entails two steps:

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution works. The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful, but it obviously

can be applied only in cases when it is easy to guess the form of the answer. The substitution method can be used to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n ,$$

which is similar to recurrences (4.2) and (4.3). We guess that the solution is T (n) = O(n lg n).Our method is to prove that T (n) ≤ cn lg n for an appropriate choice of the constant c > 0. We start by assuming that this bound holds for ⌊n/2⌋, that is, that T (⌊n/2⌋) ≤ c ⌊n/2⌋ lg(⌊n/2⌋).

Substituting into the recurrence yields

$$
\begin{aligned}
T(n) &\le 2(c\ \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\
&\le cn \lg(n/2) + n \\
&= cn \lg n - cn \lg 2 + n \\
&= cn \lg n - cn + n \\
&\le cn \lg n,
\end{aligned}
$$

where the last step holds as long as $c \ge 1$.

Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. For the recurrence (4.4), we must show that we can choose the constant c large enough so that the bound T(n) = cn lg n works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that T (1) = 1 is the sole boundary condition of the recurrence. Then for n = 1, the bound T (n) = cn lg n yields T (1) = c1 lg 1 = 0, which is at odds with T (1) = 1. Consequently, the base case of our inductive proof fails to hold.

This difficulty in proving an inductive hypothesis for a specific boundary condition can be easily overcome. For example, in the recurrence (4.4), we take advantage of asymptotic notation only requiring us to prove T (n) = cn lg n for $n \ge n0$, where n0 is a constant of our choosing. The idea is to remove the difficult boundary condition T (1) = 1 from consideration

1. In the inductive proof.

Observe that for n > 3, the recurrence does not depend directly on T (1). Thus, we can replace T (1) by T (2) and T (3) as the base cases in the inductive proof, letting n0 = 2. Note that we make a distinction between the base case of the recurrence (n = 1) and the base cases of the inductive proof (n = 2 and n = 3). We derive from the recurrence that T (2) = 4 and T (3) = 5. The inductive proof that T (n) $\le$ cn lg n for some constant $c \ge 1$ can now be completed by choosing c large enough so that T (2) $\le$ c2 lg 2 and T (3) $\le$ c3 lg 3. As it turns out, any choice of $c \ge 2$ suffices for the base cases of n = 2 and n = 3 to hold. For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n.

2. The iteration method

The method of iterating a recurrence doesn't require us to guess the answer, but it may require more algebra than the substitution method. The idea is to expand (iterate) the

recurrence and express it as a summation of terms dependent only on n and the initial conditions. Techniques for evaluating summations can then be used to provide bounds on the solution.

As an example, consider the recurrence

T(n) = 3T(n/4) + n.
We iterate it as follows:

T(n) = n + 3T(n/4)

= n + 3 (n/4 + 3T(n/16))

= n + 3(n/4 + 3(n/16 + 3T(n/64)))

= n + 3 n/4 + 9 n/16 + 27T(n/64),
where n/4/4 = n/16 and n/16/4 = n/64 follow from the identity (2.4).

How far must we iterate the recurrence before we reach a boundary condition? The *i*th term in the series is $3^i\ n/4^i$. The iteration hits $n = 1$ when $n/4^i = 1$ or, equivalently, when i exceeds $\log_4 n$. By continuing the iteration until this point and using the bound $n/4^i \leq n/4^i$, we discover that the summation contains a decreasing geometric series:

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^{\log_4 n}\Theta(1) \\ &\leq n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n)\ . \end{aligned}$$

3. The master method

The master method provides a "cookbook" method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The master method requires memorization of three cases, but then the solution of many recurrences can be determined quite easily, often without pencil and paper.

The recurrence (4.5) describes the running time of an algorithm that divides a problem of size
n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$. (That is, using the notation from Section 2.3.2, $f(n) = D(n)+C(n)$.) For example, the recurrence arising from the MERGE-SORT procedure has $a = 2$, $b = 2$, and $f(n)$ =
$\Theta(n)$.

As a matter of technical correctness, the recurrence isn't actually well defined because n/b might not be an integer. Replacing each of the a terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ doesn't affect the asymptotic behavior of the recurrence, however. We normally find it convenient, therefore, to omit the floor and ceiling functions when writing divide-and- conquer recurrences of this form.

1.4 Analysis of Linear Search

Linear Search, as the name implies is a searching algorithm which obtains its result by traversing a list of data items in a linear fashion. It will start at the beginning of a list, and mosey on through until the desired element is found, or in some cases is not found. The aspect of Linear Search which makes it inefficient in this respect is that if the element is not in the list it will have to go through the entire list. As you can imagine this can be quite cumbersome for lists of very large magnitude, keep this in mind as you contemplate how and where to implement this algorithm. Of course conversely the best case for this would be that the element one is searching for is the first element of the list, this will be elaborated more so in the –Analysis & Conclusion‖ section of this tutorial.

Linear Search Steps:

Step 1 - Does the item match the value I'm looking for?

Step 2 - If it does match return, you've found your item!
Step 3 - If it does not match advance and repeat the process.
Step 4 - Reached the end of the list and still no value found? Well obviously the item is not in the list! Return -1 to signify you have not found your value.

As always, visual representations are a bit more clear and concise so let me present one for you now. Imagine you have a random assortment of integers for this list:

Legend:
-The key is blue
-The current item is green.
-Checked items are red

Ok so here is our number set, my lucky number happens to be 7 so let‘s put this value as the key, or the value in which we hope Linear Search can find. Notice the indexes of the

---

array above each of the elements, meaning this has a size or length of 5. I digress let us look at the first term at position 0. The value held here 3, which is not equal to 7. We move on.
--0 1 2 3 4 5
[3 2 5 1 7 0]

So we hit position 0, on to position 1. The value 2 is held here. Hmm still not equal to 7. We march on.
--0 1 2 3 4 5
[3 2 5 1 7 0]

Position 2 is next on the list, and sadly holds a 5, still not the number we‘re looking for. Again we move up one.
--0 1 2 3 4 5
[3 2 5 1 7 0]

Now at index 3 we have value 1. Nice try but no cigar let‘s move forward yet again.
--0 1 2 3 4 5
[3 2 5 1 7 0]

Ah Ha! Position 4 is the one that has been harboring 7, we return the position in the array which holds 7 and exit.
--0 1 2 3 4 5
[3 2 5 1 7 0]

As you can tell, the algorithm may work find for sets of small data but for incredibly large data sets I don‘t think I have to convince you any further that this would just be down right inefficient to use for exceeding large sets. Again keep in mind that Linear Search has its place and it is not meant to be perfect but to mold to your situation that requires a search.

Also note that if we were looking for lets say 4 in our list above (4 is not in the set) we would traverse through the entire list and exit empty handed. I intend to do a tutorial on Binary Search which will give a much better solution to what we have here however it requires a special case.

```
//linearSearch Function
int linearSearch(int data[], int length, int val) {

  for (int i = 0; i <= length; i++) {
    if (val == data[i]) {
      return i;
    }//end if
  }//end for
  return -1;    //Value was not in the list
}//end linearSearch Function
```

Analysis & Conclusion

As we have seen throughout this tutorial that Linear Search is certainly not the absolute best method for searching but do not let this taint your view on the algorithm itself. People are always attempting to better versions of current algorithms in an effort to make existing ones more efficient. Not to mention that Linear Search as shown has its place and at the very least is a great beginner‘s introduction into the world of searching algorithms. With this is mind we progress to the asymptotic analysis of the Linear Search:

Worst Case:
The worse case for Linear Search is achieved if the element to be found is not in the list at all. This would entail the algorithm to traverse the entire list and return nothing. Thus the worst case running time is:

O(N).

Average Case:
The average case is in short revealed by insinuating that the average element would be somewhere in the middle of the list or N/2. This does not change since we are dividing by a constant factor here, so again the average case would be:

O(N).

Best Case:
The best case can be a reached if the element to be found is the first one in the list. This would not have to do any traversing spare the first one giving this a constant time complexity or:

O(1).

IMPORTANT QUESTIONS

PART-A

1. Define Algorithm & Notion of algorithm.
2. What is analysis framework?
3. What are the algorithm design techniques?
4. How is an algorithm‘s time efficiency measured?
5. Mention any four classes of algorithm efficiency.
6. Define Order of Growth.
7. State the following Terms.
(i) Time Complexity
(ii) Space Complexity
8. What are the various asymptotic Notations?
9. What are the important problem types?
10. Define algorithmic Strategy (or) Algorithmic Technique.
11. What are the various algorithm strategies (or) algorithm Techniques?
12. What are the ways to specify an algorithm?
13. Define Best case Time Complexity .
14. Define Worst case Time Complexity.
15. Define Average case time complexity.
16. What are the Basic Efficiency Classes.
17. Define Asymptotic Notation.
18. How to calculate the GCD value?

PART-B

1. (a) Describe the steps in analyzing & coding an algorithm. (10)
(b) Explain some of the problem types used in the design of algorithm. (6)
2. (a) Discuss the fundamentals of analysis framework . (10)
(b) Explain the various asymptotic notations used in algorithm design. (6)
3. (a) Explain the general framework for analyzing the efficiency of algorithm. (8)
(b) Explain the various asymptotic efficiencies of an algorithm. (8)
4. (a) Explain the basic efficiency classes. (10)
(b) Explain briefly the concept of algorithmic strategies. (6)
5. Describe briefly the notions of complexity of an algorithm. (16)
6. (a) What is Pseudo-code? Explain with an example. (8)
(b) Find the complexity C(n) of the algorithm for the worst case, best case and average case.(Evaluate average case complexity for n=3,Where n is the number of inputs) (8)

UNIT II: DIVIDE AND CONQUER, GREEDY METHOD

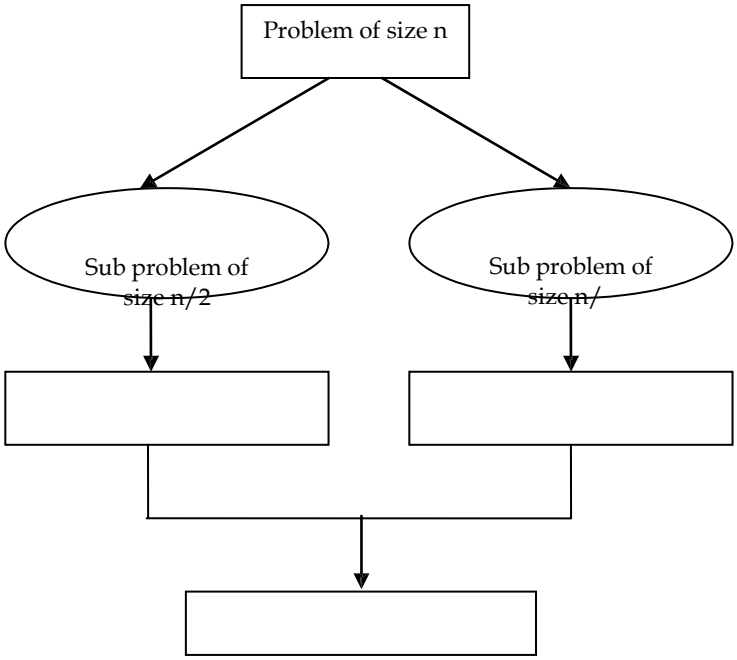
Divide and conquer – General method – Binary search – Finding maximum and minimum – Merge sort – Greedy algorithms – General method – Container loading – Knapsack problem

2.1. DIVIDE AND CONQUER

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

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original problem.

The divide-and-conquer technique is diagrammed, which depicts the case of dividing a problem into two smaller sub problems,



Examples of divide and conquer method are **Binary search, Quick sort,**

2.2. GENERAL METHOD

e.g. Merge sort.

♦ As an example, let us consider the problem of computing the sum of *n* numbers $a_0, \ldots a_{n-1}$.

♦ If $n > 1$, we can divide the problem into two instances of the same problem. They are To compute the sum of the first $\lfloor n/2 \rfloor$ numbers

♦ To compute the sum of the remaining [n/2]numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.)

♦ Once each of these two sums is computed (by applying the same method, i.e., recursively),

we can add their values to get the sum in question. i.e.,

$$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$$

♦ More generally, an instance of size n can be divided into several instances of size nib, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.).

♦ Assuming that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time T(n).

$$T(n)=aT(n/b)+f(n)$$

♦ This is called the **general divide and-conquer recurrence.** Where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. (For the summation example, a = b = 2 and f (n) = 1.

♦ Obviously, the order of growth of its solution T(n) depends on the values of the constants a and b and the order of growth of the function f (n). The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem.

2.2.1. MASTER THEOREM

If f(n) □ □ (n^d) where d ≥ 0 in recurrence equation is given by ,

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

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

♦ For example, the recurrence equation for the number of additions A(n) made by the divide-and-conquer summation 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 $a > b^d$,

$$A(n) \square\ \square\ (n^{\log_b a}) = \square\ (n^{\log_z 2}) = \square\ (n).$$

ADVANTAGES:

♦ The time spent on executing the problem using divide and conquer is smaller then other methods.

♦ The divide and conquer approach provides an efficient algorithm in computer science.

♦ The divide and conquer technique is ideally suited for parallel computation in which each sub problem can be solved simultaneously by its own processor.

♦ Merge sort is a perfect example of a successful application of the divide-and conquer technique.

♦ It sorts a given array Al O .. n - 1) by dividing it into two halves A [O.. ⌊ n/2⌋ - 1] and *A[* ⌊ n/2⌋ ..*n* - 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM Mergesort (A[O .. n - 1])
//Sorts array *A[O .. n* - 1] by recursive mergesort
//Input: An array *A[O .. n* - 1] of orderable elements
//Output*:* Array *A[O...* n - 1] sorted in nondecreasing order
if n > 1
copy A [O.. ⌊ n/2⌋ - 1] to B [O.. ⌊ n/2⌋ - 1]
copy *A[* ⌊ n/2⌋ ..*n* - 1] to C[0... ⌊n/2⌋ - 1]
Mergesort (B [O.. ⌊ n/2⌋ - 1])
Mergesort (C[0... ⌊n/2⌋ - 1])
Mergesort(B,C,A)

STEPS TO BE FOLLOWED

♦ The first step of the merge sort is to chop the list into two.

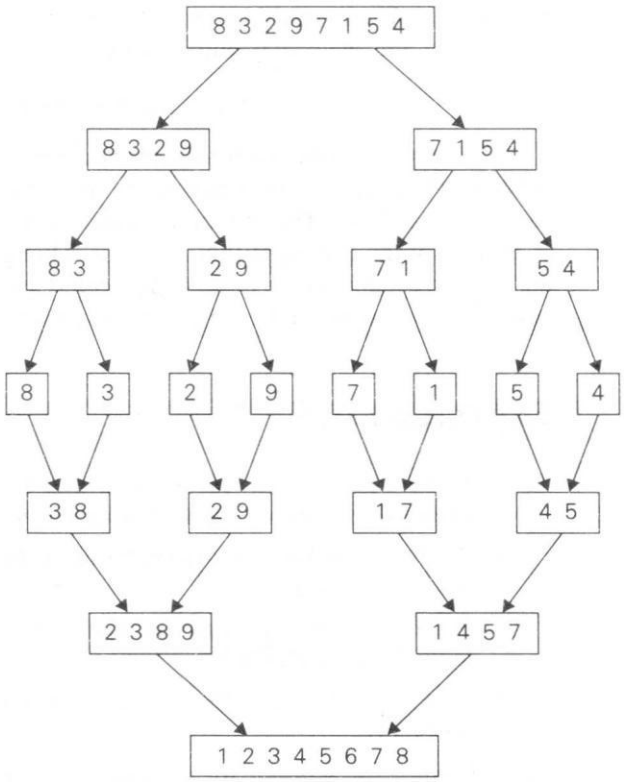
♦ If the list has even length, split the list into two equal sub lists.

♦ If the list has odd length, divide the list into two by making the first sub list one entry greater than the second sub list.

♦ then split both the sub list into two and go on until each of the sub lists are of size one.

♦ finally, start merging the individual sub lists to obtain a sorted list.

♦ The operation of the algorithm for the array of elements 8,3,2,9,7,1,5,4 is explained as follows,



AN EXAMPLE OF MERGE SORT OPEERATION

♦ 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.

♦ Then the elements pointed to are compared and the smaller of them is added to a new array or list being constructed.

♦ Then the index of the smaller element is incremented to point to its immediate successor in the array.

♦ This operation is continued until one of the two given arrays is exhausted ♦Then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM : *Merge Sort B(0....p-1), C(0....q-1), A(0....p + q -1)*
// Merges two sorted arrays into one sorted array .
// Input: Array 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 ← 0; j ←0 ;k← 0
while i<p and j<q do
if B[i] ≤ C[j]
A[k] ←B[i] ;i ←i+1
else A[k] ← C[j] j← j+1
k ←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]

2.2.2 EFFICIENCY OF MERGE SORT

♦ The recurrence relation for the number of key comparison C (n) is

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

♦ Assume, n is a power of 2. C_{merge} (n) is the number of key comparison performed during the merge sort.

♦ In the merging of two sorted array after one comparison is made the total number of elements in the two array still to be processed and sorted is reduced by one.

♦ Hence in the worst case neither of the two arrays becomes empty before the other one contains just one element . Therefore, for the worst case,

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

$C_{merge}(n) = n - 1$

and we have the recurrence

Hence, according to the Master Theorem, $C_{worst}(n) \square \square (n \log n)$. In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$

$$C_{worst}(n) = n \log_2 n - n + 1$$

2.3. BINARY SEARCH

♦ The binary search algorithm is one of the most efficient searching techniques which require the list to be sorted in ascending order.

♦ To search for an element in the list, the binary search algorithms split the list and locate the middle element of the list.

♦ The middle of the list is calculated as
Middle:=(l+r) div 2
n – number of element in list
♦ The algorithm works by comparing a search key element _k' with the array middle element a[m]

After comparison, any one of the following three conditions occur.

1. If the search key element _k' is greater than a[m],then the search element is only in the upper or second half and eliminate the element present in the lower half.Now the value of l is middle m+1.
2. If the search key element _k' is less than a[m], then the search element is only in the lower or first half. No need to check in the upper half. Now the value of r is middle m - 1.
3. If the search key element _k' is equal to a[m], then the search key element is found in the position m. Hence the search operation is complete.

EXAMPLE:

The list of element are 3,14,27,31,39,42,55,70,74,81,85,93,98 and searching for k=70 in the list.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
	l						m						r

m – middle element
m = n div 2
=13 div 2
m = 6

If k>a[m], then ,l =7. So, the search element is present in second half .

Now the array becomes

7	8	9	10	11	12
70	74	81	85	93	98
l					r

m = (l + r) div 2 = 19 div 2
m = 9

7	8	9	10	11	12
70	74	81	85	93	98
l		m			r

Here k<a[m]
70 < 81
So, the element is present in the first half

Now, the array becomes

7	8
70	74
l	r

Now m = (l +r) div 2
= (7 +8) div 2
m = 7

7	8
70	74
l,m	r

Now k = a[m]
70 =70

Hence, the search key element 70 is found in the position 7 and the search operation is completed.

ALGORITHM: *Binary search (A[0..n – 1],k)*

//Implements nonrecursive binary search

// Input:: An array A[0..n – 1] sorted in ascending order and a search key k

// Output: An index of the array's element that is equal to k or -1 if there is no

// such element

l← 0; r ← n – 1

while l<= r do

m ← ⌊(l + r)/ 2⌋

if k = A[m] return m

else if k < A[m] r ← m – 1

else l ← m+ 1

return -1

2.3.1. EFFICIENCY OF BINARY SEARCH

♦ The standard way to analyze the efficiency is to count number of times search key is compared with an element of the array.

WORST CASE ANALYSIS

♦ The worst case include all array that do not contain a search key.

♦ The recurrence relation for $C_{worst}(n)$ is

$C_{worst}(n)=C_w(\lfloor n/2 \rfloor) + 1$,for $n > 1$ --------(1)

$C_{worst}(1)=1$

For $n = 2^k$

Equation (1) becomes

$C_{worst}(2^k) = k + 1$

$C_{worst}(2^k) = \log_2 n + 1$ -------(2)

n is positive integer,

$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$ -------(3)

$C_{worst}(n) = \lfloor \log_2 (n+1) \rfloor$ -------(4)

In equation (1) put n = 2i.

L.H.S becomes

$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$

$= \lfloor \log_2 2i \rfloor + 1$

$= \lfloor og_2 2 \rfloor + \lfloor \log_2 i \rfloor + 1$

$= 1 + \lfloor \log_2 i \rfloor + 1$

$= 2 + \lfloor \log_2 i \rfloor$

$C_{worst}(n) = 2 + \lfloor \log_2 i \rfloor$

R.H.S becomes

$C_{worst} \lfloor n / 2 \rfloor + 1 = C_{worst} \lfloor 2i / 2 \rfloor + 1$

$= C_{worst}(i) + 1$

$= \log_2 i + 1 + 1$

$= 2 + \lfloor \log_2 i \rfloor$

$C_{worst} \lfloor n / 2 \rfloor + 1 = 2 + \lfloor \log_2 i \rfloor$

L.H.S =R.H.S

Hence

$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$ and

$C_{worst}(i) = \lfloor \log_2 i \rfloor + 1$ are same

Hence

$C_{worst}(n) = \square$ (log n)

From equation (4) to search a element in a array of 1000 elements ,binary search takes.

$\lfloor \log_2 10^3 \rfloor + 1 = 10$ key comparison

AVERAGE CASE ANALYSIS:

Average number of key comparison made by the binary search is slightly smaller than worst case.

$C_{avg}(n) \approx \log_2 n$

The average number of comparison in the successful search is

$C^{yes}_{avg}(n) \approx \log_2 n - 1$

The average number of comparison in the unsuccessful search is

$C^{no}_{avg}(n) \approx \log_2 (n + 1)$

ADVANTAGES

1. In this method elements are eliminated by half each time .So it is very faster than the sequential search.
2. It requires less number of comparisons than sequential search to locate the search key element.

DISADVANTAGES

1. An insertion and deletion of a record requires many records in the existing table be physically moved in order to maintain the records in sequential order.
2. The ratio between insertion/deletion and search time is very high.

2.4. GREEDY TECHNIQUES

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—and this is the central point of this technique—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.

2.5. GENERAL METHOD

2.5.1. PRIM'S ALGORITHM

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.

Two serious difficulties to construct Minimum Spanning Tree

1. The number of spanning trees grows exponentially with the graph size (at least for dense graphs).
2. Generating all spanning trees for a given graph is not easy.

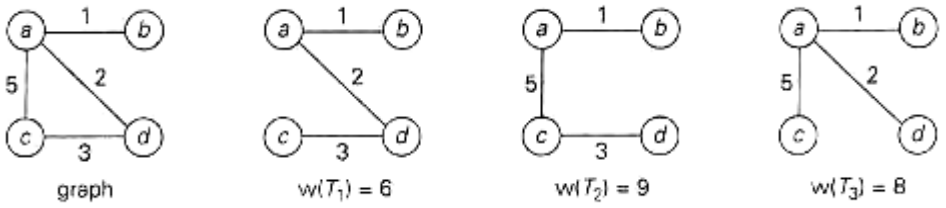


Figure: Graph and its spanning trees; T_1 is the Minimum Spanning Tree

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. 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, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. 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 used for the tree expansions.

Pseudocode of this algorithm

The nature of Prim‘s algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex. We can provide such information by attaching two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge. Vertices that are not adjacent to any of the tree vertices can be given the label indicating their ―infinite‖ distance to the tree vertices a null label for the name of the nearest tree vertex. With such labels, finding the next vertex to be added to the current tree $T = (V_T, E_T)$ become simple task of finding a vertex with the smallest distance label in the set $V - V_T$. 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 set $V—V_T$ to the set of tree vertices V_T.
- For each remaining vertex U in $V—V_T$ - that is connected to u* by a shorter edge than the u‘s current distance label, update its labels by u* and the weight of the edge between u* and u, respectively.

The following figure demonstrates the application of Prim‘s algorithm to a specific graph.

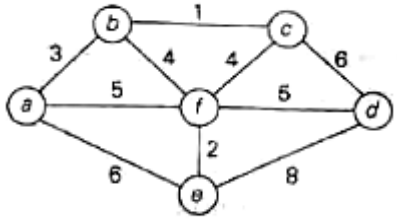


Fig: Graph given to find minimum spanning tree

2.6. CONTAINER LOADING

The greedy algorithm constructs the loading plan of a single container layer by layer from the bottom up. At the initial stage, the list of available surfaces contains only the initial surface of size *L* x *W* with its initial position at height 0. At each step, the algorithm picks the lowest usable surface and then determines the box type to be packed onto the surface, the number of the boxes and the rectangle area the boxes to be packed onto, by the procedure *select layer*.

The procedure *select layer* calculates a layer of boxes of the same type with the highest evaluation value. The procedure *select layer* uses breadth-limited tree search heuristic to determine the most promising layer, where the breadth is different depending on the different depth level in the tree search. The advantage is that the number of nodes expanded is polynomial to the maximal depth of the problem, instead of exponentially growing with regard to the problem size. After packing the specified number of boxes onto the surface according to the layer arrangement, the surface is divided into up to three sub-surfaces by the procedure *divide surfaces*.

Then, the original surface is deleted from the list of available surfaces and the newly generated sub-surfaces are inserted into the list. Then, the algorithm selects the new lowest usable surface and repeats the above procedures until no surface is available or all the boxes have been packed into the container. The algorithm follows a similar basic framework.

The pseudo-code of the greedy algorithm is given by the *greedy heuristic* procedure.

procedure greedy heuristic()
list of surface := initial surface of L x W at height 0
list of box type := all box types
while (there exist usable surfaces) and (not all boxes are packed) do
select the lowest usable surface as current surface
set depth := 0
set best layer := select layer(list of surface, list of box type, depth)
pack best layer on current surface
reduce the number of the packed box type by the packed amount
set a list of new surfaces := divide surfaces(current surface, best layer, list of box type)
delete current surface from the list of surfaces
insert each surface in list of new surfaces into list of surfaces
end while

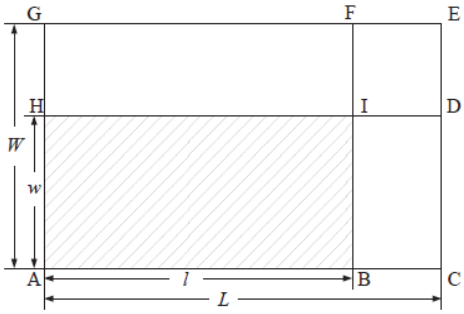


Fig: Division of the Loading Surface

Given a layer of boxes of the same type arranged by the G4-heuristic, the layer is always packed in the bottom-left corner of the loading surface.

As illustrated in above Figure, up to three sub-surfaces are to be created from the original loading surface by the procedure *divide surfaces*, including the top surface, which is above the layer just packed, and the possible spaces that might be left at the sides.
If $l = L$ or $w = W$, the original surface is simply divided into one or two sub-surfaces, the top surface and a possible side surface. Otherwise, two possible division variants exist, i.e., to divide into the top surface, the surface (*B,C,E,F*) and the surface (*F,G,H, I*), or to divide into the top surface, the surface (*B,C,D, I*) and the surface (*D,E,G,H*).

The divisions are done according to the following criteria, which are similar to those in [2] and [5]. The primary criterion is to minimize the total unusable area of the division variant. If none of the remaining boxes can be packed onto a sub-surface, the area of the sub-surface is unusable. The secondary criterion is to avoid the creation of long narrow strips.

–The underlying rationale is that narrow areas might be difficult to fill subsequently‖. More specifically, if $L-l \geq W-w$, the loading surface is divided into the top surface, the surface (*B,C,E,F*) and the surface (*F,G,H, I*). Otherwise, it is divided into the top surface, the surface (*B,C,D, I*) and the surface (*D,E,G,H*).

2.6.1. Algorithm for Container Loading

```
void containerLoading(container* c, int capacity,
                      int numberOfContainers, int* x)
{// Greedy algorithm for container loading.
 // Set x[i] = 1 iff container i, i >= 1 is loaded.
   // sort into increasing order of weight
   heapSort(c, numberOfContainers);

   int n = numberOfContainers;

   // initialize x
   for (int i = 1; i <= n; i++)
      x[i] = 0;

   // select containers in order of weight
   for (int i = 1; i <= n && c[i].weight <= capacity; i++)
   {// enough capacity for container c[i].id
      x[c[i].id] = 1;
      capacity -= c[i].weight;  // remaining capacity
   }
}
```

2.7. KNAPSACK PROBLEM

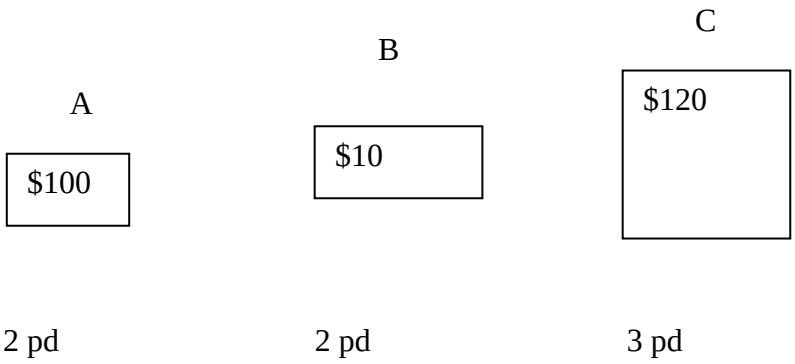
The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.

The problem often arises in resource allocation with financial constraints. A similar problem also appears in combinatorics, complexity theory, cryptography and applied mathematics.

The decision problem form of the knapsack problem is the question "can a value of at least V be achieved without exceeding the weight W?"

E.g.

A thief enters a store and sees the following items:



His Knapsack holds 4 pounds.
What should he steal to maximize profit?

Fractional Knapsack Problem
Thief can take a fraction of an item.

Solution = +

2 pounds of item A 2 pounds of item C

2 pds A $100	2 pds C $80

IMPORTANT QUESTIONS

PART-A

1. Give the time efficiency & Drawback of merge sort Algorithm.
2. What is the difference between DFS & BFS?
3. What is the Brute Force Algorithmic Strategy?
4. State the time complexity of following:
(i) Bubble sort
(ii) Selection sort
(iii) Sequential search
(iv) Brute force string matching
5. What are the features of Brute force String matching algorithm?
6. Define –Divide & Conquer Technique‖.
7. State Master‘s Theorem.
8. Define Merge sort & explain three steps of Merge sort.
9. Define Quick sort & explain three steps of Quick sort.
10. Define Binary Search.
11. What are the applications of binary search?
12. State advantages & Disadvantages of binary search.
13. Explain Binary search tree.
14. What is the recurrence relation for divide & conquer?
15. Explain Decrease & Conquer.
16. What are the variations of Decrease & Conquer?
17. What are the applications of decrease by constant?
18. Write any four advantages of insertion sort.
19. What is Greedy method?
20. Compare Greedy algorithm & Dynamic programming.
21. Define Knapsack‘s problem.

PART-B

1. Explain Quick sort algorithm with suitable Example. (16)
2. (a) Write an algorithm to sort a set of ‗N‘ numbers using insertion sort. (8)
(b) Expalin the difference between depth first search & Breadth first search.(8)
3. (a) Write a pseudo code for divide & conquer algorithm for merging two sorted arrays in to a single sorted one. Explain with example. (12)
(b) Set up & solve a recurrence relation for the number of key comparisons made by above pseudo code. (4)

4. Design a recursive Decrease-by-one algorithm for sorting the n real numbers in any array with an example & also determine the number of key comparisons & time efficiency of the algorithm. (16)
5. (a) Give an algorithm for selection sort & analyze your algorithm. (10)
(b) Give Strength & Weakness of Brute force algorithm. (6)

.

UNIT III DYNAMIC PROGRAMMING

Dynamic programming – General method – Multistage graphs – All-pair shortest paths – Optimal binary search trees – 0/1 Knapsack – Traveling salesperson problem

3.1. DYNAMIC PROGRAMMING

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. 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 we can then obtain a solution to the original problem.

E.g. Fibonacci Numbers

0,1,1,2,3,5,8,13,21,34,...,

which can be defined by the simple recurrence

F(0) = 0, F(1)=1.

and two initial conditions

F(n) = F(n-1) + F(n-2) for $n \geq 2$

3.2. GENERAL METHOD - COMPUTING A BINOMIAL COEFFICIENT

Computing a binomial coefficient is a standard example of applying dynamic programming to a nonoptimization problem.

Of the numerous properties of binomial coefficients, we concentrate on two:

C(n,k) = C(n-1,k-1) + C(n-1,k) for n > k > 0 and

C(n, 0) = C(n, n) = 1.

3.2.1. Pseudocode for Binomial Coefficient

3.3. MULTISTAGE GRAPHS –
ALL PAIR SHORTESET PATH - **FLOYD'S ALGORITHM**

Given a weighted connected graph (undirected or directed), the all-pair shortest paths problem asks to find the distances (the lengths of the shortest paths) from each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an n-by-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 ($1 \leq i,j \leq n$). We can generate the distance matrix with an algorithm called **Floyd's algorithm**. It is applicable to **both undirected and directed weighted** graphs provided that they do not contain a cycle of a negative length.

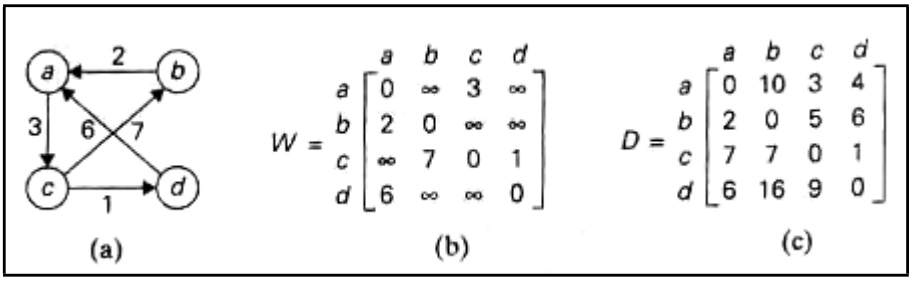


Fig: (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Floyd's algorithm computes the distance matrix of a weighted graph with vertices through a series of n-by-n matrices: $D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}$.

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix. Specifically, the element in the ith row and the jth column of matrix D (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. In particular, the series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is nothing but the weight matrix of the graph. The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths

that can use all n vertices as intermediate and hence is nothing but the distance matrix being sought.

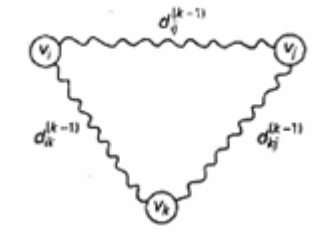


Fig: Underlying idea of Floyd's algorithm

3.3.1. Pseudocode for Floyd's Algorithms

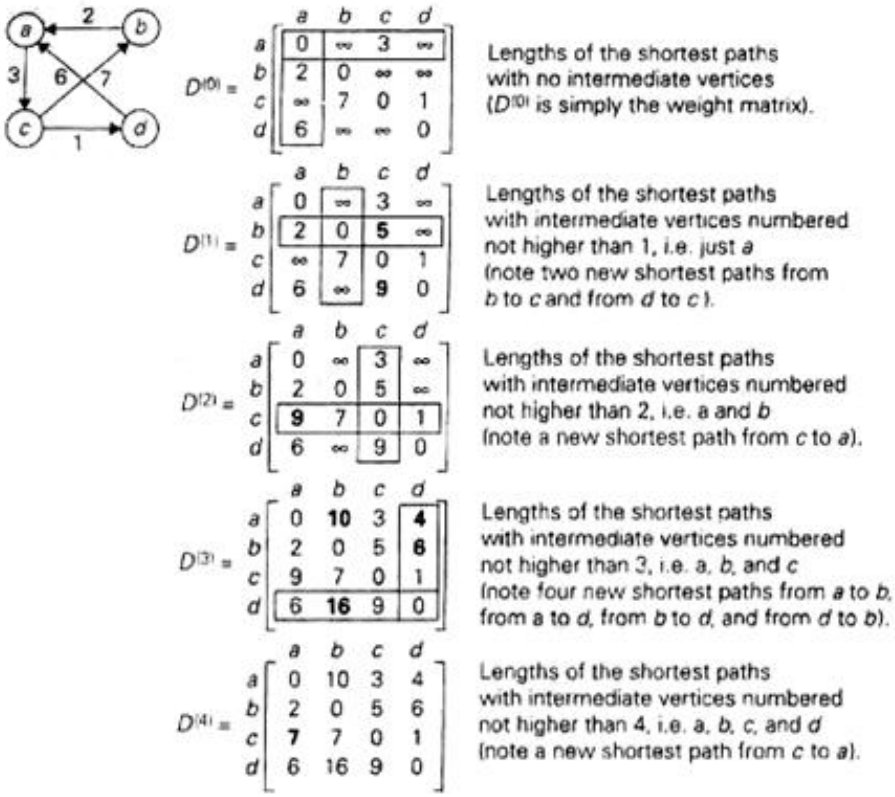


Fig: Application of Floyd's algorithm to the graph shown. Updated elements are shown in bold.

3.4. OPTIMAL BINARY SEARCH TREES

A binary search tree_s principal application is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

In an optimal binary search tree, the average number of comparisons in a search is the smallest possible.

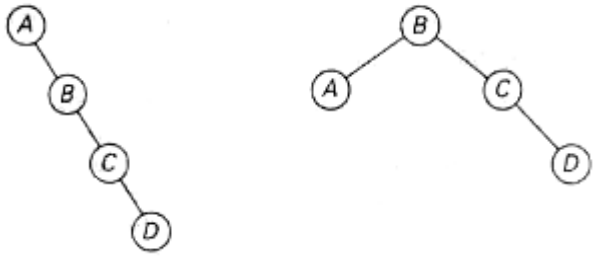


Fig: Two out of 14 possible binary search trees with keys A, B, C. and D

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 above 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 this trees is $0.1 \cdot 1 + 0.2 \cdot 2 + 0.4 \cdot 3 + 0.3 \cdot 4 = 2.9$ while for the second one it is $0.1 \cdot 2 + 0.2 \cdot 1 + 0.4 \cdot 2 + 0.3 \cdot 3 = 2.1$. Neither of these two trees is, in fact, optimal.

For this 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** which grows to infinity as fast as $4^n/n^{1.5}$.

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

$$
\begin{aligned}
C[i,j] &= \min_{i<k\le j}\{p_k\cdot 1+\sum_{s=i}^{k-1} p_s\cdot(\text{level of } a_s \text{ in } T_i^{k-1}+1)\\
&\qquad +\sum_{s=k+1}^{j} p_s\cdot(\text{level of } a_s \text{ in } T_{k+1}^{j}+1)\}\\
&= \min_{i\le k\le j}\{p_k+\sum_{s=i}^{k-1} p_s\cdot \text{level of } a_s \text{ in } T_i^{k-1}+\sum_{s=i}^{k-1} p_s\\
&\qquad +\sum_{s=k+1}^{j} p_s\cdot \text{level of } a_s \text{ in } T_{k+1}^{j}+\sum_{s=k+1}^{j} p_s\}\\
&= \min_{i\le k\le j}\{\sum_{s=i}^{k-1} p_s\cdot \text{level of } a_s \text{ in } T_i^{k-1}+\sum_{s=k+1}^{j} p_s\cdot \text{level of } a_s \text{ in } T_{k-1}^{j}+\sum_{s=i}^{j} p_s\}\\
&= \min_{i\le k\le j}\{C[i,k-1]+C[k+1,j]\}+\sum_{s=i}^{j} p_s.
\end{aligned}
$$

Thus, we have the recurrence

$$
C[i,j]=\min_{i\le k\le j}\{C[i,k-1]+C[k+1,j]\}+\sum_{s=i}^{j} p_s \text{ for } 1\le i\le j\le n.
$$

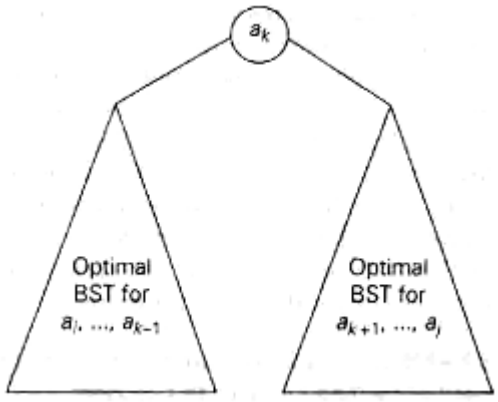


Fig: Binary search tree with root a_k and two optimal binary search subtrees and

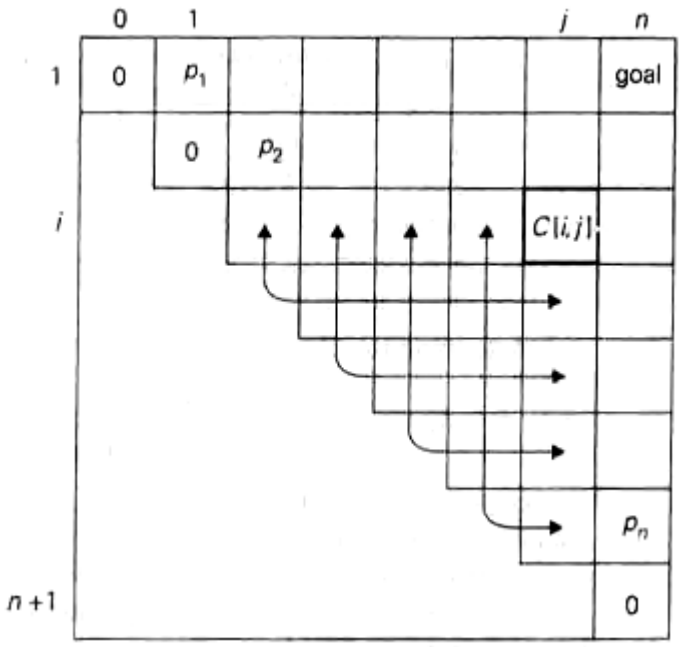


Fig: Table of the dynamic programming algorithm for constructing an optimal binary search tree

EXAMPLE 1: Let us illustrate the algorithm by applying it to the four-key set.

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]:

$$C[1,2] = \min \begin{matrix} k=1\text{: } C[1,0] + C[2,2] + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2\text{: } C[1,1] + C[3,2] + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{matrix} = 0.4$$

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.

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 three 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 optima] 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 V. The following figure presents the optimal tree in its entirety.

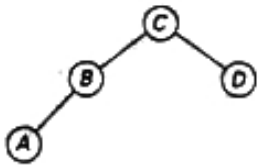


Figure: Optimal binary search tree for the example

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 $C[i, k-1] + C[k+1, j] < minval$
        $minval \leftarrow C[i, k-1] + C[k+1, j]$; $kmin \leftarrow k$
    $R[i, j] \leftarrow k$
    $sum \leftarrow P[i]$; **for** $s \leftarrow i+1$ **to** j **do** $sum \leftarrow sum + P[s]$
    $C[i, j] \leftarrow minval + sum$
return $C[1, n]$, R

3.5. 0/1 KNAPSACK PROBLEM

Let i be the highest-numbered item in an optimal solution S for W pounds. Then $S` = S - \{i\}$ is an optimal solution for $W - w_i$ pounds and the value to the solution S is V_i plus the value of the subproblem.

We can express this fact in the following formula: define $c[i, w]$ to be the solution for items 1,2, . . . , i and maximum weight w. Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i\text{-}1, w] & \text{if } w_i \geq 0 \\ \max [v_i + c[i\text{-}1, w\text{-}w_i], c[i\text{-}1, w]\} & \text{if } i>0 \text{ and } w \geq w_i \end{cases}$$

This says that the value of the solution to i items either include i^{th} item, in which case it is v_i plus a subproblem solution for (i - 1) items and the weight excluding w_i, or does not include i^{th} item, in which case it is a subproblem's solution for (i - 1) items and the same weight.

That is, if the thief picks item i, thief takes v_i value, and thief can choose from items $w - w_i$, and get $c[i - 1, w - w_i]$ additional value. On other hand, if thief decides not to take item i, thief can choose from item 1,2, . . . , i- 1 upto the weight limit w, and get $c[i - 1, w]$ value. The better of these two choices should be made.

Although the 0-1 knapsack problem, the above formula for c is similar to LCS formula: boundary values are 0, and other values are computed from the input and "earlier" values of c. So the 0-1 knapsack algorithm is like the LCS-length algorithm given in CLR for finding a longest common subsequence of two sequences.

The algorithm takes as input the maximum weight W, the number of items n, and the two sequences $v = <v_1, v_2, . . . , v_n>$ and $w = <w_1, w_2, . . . , w_n>$. It stores the $c[i, j]$ values in the table, that is, a two dimensional array, $c[0 . . n, 0 . . w]$ whose entries are computed in a row-major order. That is, the first row of c is filled in from left to right, then the second row, and so on. At the end of the computation, $c[n, w]$ contains the maximum value that can be picked into the knapsack.

```
dynamic-0-1-knapsack (v, w, n, w)
for w = 0 to w
   do  c[0, w] = 0
for i=1 to n
   do c[i, 0] = 0
      for w=1 to w
         do iff wi ≤ w
            then if  vi + c[i-1, w-wi]
               then c[i, w] = vi + c[i-1, w-wi]
               else c[i, w] = c[i-1, w]
            else
               c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at $c[n.\ w]$ and tracing backwards where the optimal values came from. If $c[i, w] = c[i\text{-}1, w]$ item i is not part of the solution, and we are continue tracing with $c[i\text{-}1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i\text{-}1, w\text{-}W]$.

Analysis

This dynamic-0-1-kanpsack algorithm takes $\theta(nw)$ times, broken up as follows: $\theta(nw)$ times to fill the c-table, which has $(n+1).(w+1)$ entries, each requiring $\theta(1)$ time to compute. $O(n)$ time to trace the solution, because the tracing process starts in row n of the table and moves up 1 row at each step.

3.6. TRAVELING SALESMAN PROBLEM

We will be able to apply the dynamic programming 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 it. But there is a less obvious and more informative lower bound, which does not require a lot of work to compute.

It is not difficult to show that we can compute a lower bound on the length I of any tour as follows.

For each city i, $1 \le i \le n$, find the sum s_i of the distances from city i to the two nearest cities; compute the sums of these n numbers; divide the result by 2, and, it all the distances are integers, round up the result to the nearest integer:

$$\mathbf{lb} = \lceil \mathbf{S/2} \rceil$$

b) State-space tree of the branch-and- bound algorithm applied to this graph.

(The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.)

For example, for the instance of the above figure, formula yields

$$lb = \lceil [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2 \rceil = 14.$$

We now apply the branch and bound algorithm, with the bounding function given by formula, to find the shortest Hamiltonian circuit for the graph of the above figure (a). To reduce the amount of potential work, we take advantage of two observations.

First, without loss of generality, we can consider only tours that start at *a*.

Second, because our graph is undirected, we can generate only tours in which b is visited before c. 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 the above figure (b).

The comments we made at the end of the preceding section about the strengths and weaknesses of backtracking are applicable to branch-and-bound as well. To reiterate the main point: these state-space tree techniques enable 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.

Incorporation of additional information, such as symmetry of a game‘s board, can widen the range of solvable instances. Along this line, a branch-and- bound algorithm can be sometimes accelerated by knowledge of the objective function‘s value of some nontrivial feasible solution.

The information might be obtainable—say, by exploiting specifics of the data or even, for some problems, generated randomly—before we start developing a state-space tree. Then we can use such a solution immediately as the best one seen so far rather than waiting for the branch-and-bound processing to lead us to the first feasible solution.

In contrast to backtracking, solving a problem by branch-and-bound ha both the challenge and opportunity of choosing an order of node generation and finding a good bounding function.

Though the best-first rule we used above is a sensible approach, it mayor may not lead to a solution faster than other strategies.

IMPORTANT QUESTIONS

PART A

1. Define dynamic programming.
2. What are the multistage of graphs?
3. What are all the shortest paths in binary search?
4. Define Binary Search.
5. What are the applications of binary search?
6. State advantages & Disadvantages of binary search.
7. Explain Binary search tree.
8. What are the two types of searching algorithm?
9. Define the Following Terms:
(i) Tree Edge
(ii) Back Edge
(iii) Cross Edge
10. State the following terms:
(i) Balanced Tree
(ii) Unbalanced Tree
11. What is height of balanced tree?
12. What is balance factor?
13. Define rotation.

PART B

1Solve the all pair shortest path problem for the diagraph with the weighted matrix given below:-

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

2.Define AVL tree.Explain the construction sequence of AVL tree with simple example.
3 (a) Give an algorithm for selection sort & analyze your algorithm. (10)
(b) Give Strength & Weakness of Brute force algorithm. (6)

4 . Give a suitable example & explain the Breadth first search & Depth first search. (16)

5. Find the number of comparisons made by the sentinel version of sequential search algorithm for in,
(i)Worst case
(ii)Average case (16)

UNIT IV BACKTRACKING

Backtracking – General method – 8 Queens Problem – Sum of subsets – Graph coloring – Hamiltonian problem – Knapsack problem

4.1. BACKTRACKING

Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

4.1.1.. GENERAL METHOD

- 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.

4.1.2. STATE-SPACE TREE

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 **nonpromising**.

Leaves represent either **nonpromising dead ends** or **complete solutions** found by the algorithm.

4.2. N-QUEENS PROBLEM

The problem is to place it queens on an n-by-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. For n = 1, the problem has a trivial solution, and it is easy to see that there is no solution for n = 2

and n =3. 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 the following figure.

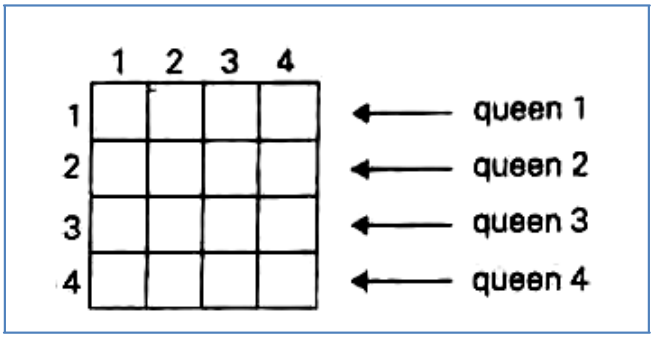


Fig: Board for the Four-queens problem

Steps to be followed

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 i 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.

(x 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.

4.3. SUBSET-SUM PROBLEM

Subset-Sum Problem is finding a subset of a given set S = $\{s_1,s_2 \ldots s_n\}$ of n positive integers whos**e** sum is equal to a given positive integer d.

For example, for S = {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

$$s_1 \leq s_2 \leq \ldots\ldots\ldots \leq s_n$$

The state-space tree can be constructed as a binary tree as that in the following figure for the instances S = (3, 5, 6, 7) and d = 15.

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 ofs1 in a set being sought.
Similarly, going to the left from a node of the first level corresponds to inclusion of s2, while going to the right corresponds to its exclusion, and soon.

Thus, a path from the root to a node on the *i*th 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, Ifs 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 he found, continue by backtracking to the node‘s parent.

If s‘ is not equal to d, we can terminate the node as nonpromising if either of the two inequalities holds:

$$s' + s_{i+1} > d \text{ (the sum } s' \text{ is too large)}$$

$$s' + \sum_{j=i+1}^{n} s_j < d \text{ (the sum } s' \text{ is too small).}$$

4.3.1. Pseudocode For Backtrack Algorithms

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 8
for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints **do**
$X[i+1] \leftarrow x$
Backtrack$(X[1..i+1])$

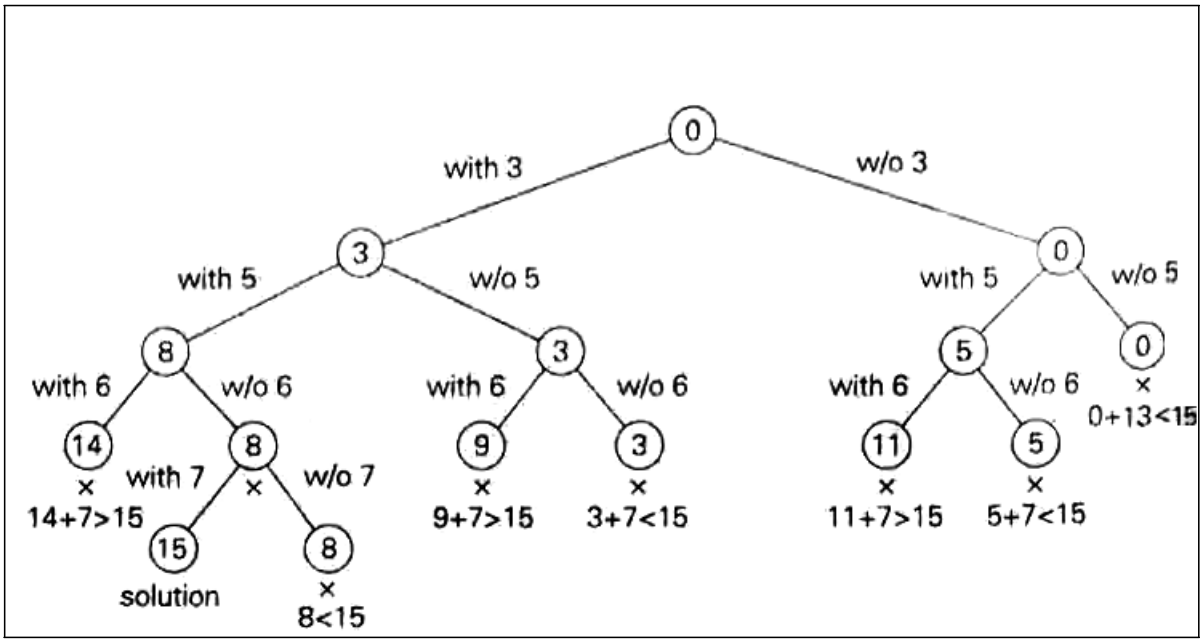


Fig: Complete state-space tree of the backtracking algorithm

(applied to the instance S = (3, 5, 6, 7) and d = 15 of the subset-sum problem. The number inside a node is the sum of the elements already included in subsets represented by the node. The inequality below a leaf indicates the reason for its termination)

4.4. GRAPH COLORING

A coloring of a graph is an assignment of a color to each vertex of the graph so that no two vertices connected by an edge have the same color. It is not hard to see that our problem is one of coloring the graph of incompatible turns using as few colors as possible.

The problem of coloring graphs has been studied for many decades, and the theory of algorithms tells us a lot about this problem. Unfortunately, coloring an arbitrary graph with as few colors as possible is one of a large class of problems called "NP-complete problems," for which all known solutions are essentially of the type "try all possibilities."

A k-coloring of an undirected graph G = (V, E) is a function c : V → {1, 2,..., k} such that c(u) ≠ c(v) for every edge (u, v) E. In other words, the numbers 1, 2,..., k represent the k colors, and adjacent vertices must have different colors. The graph-coloring problem is to determine the minimum number of colors needed to color a given graph.

a. Give an efficient algorithm to determine a 2-coloring of a graph if one exists.

b. Cast the graph-coloring problem as a decision problem. Show that your decision problem is solvable in polynomial time if and only if the graph-coloring problem is solvable in polynomial time.

c. Let the language 3-COLOR be the set of graphs that can be 3-colored. Show that if 3-COLOR is NP-complete, then your decision problem from part (b) is NP-complete.

To prove that 3-COLOR is NP-complete, we use a reduction from 3-CNF-SAT. Given a formula φ of m clauses on n variables x, x,..., x, we construct a graph G = (V, E) as follows.

The set V consists of a vertex for each variable, a vertex for the negation of each variable, 5 vertices for each clause, and 3 special vertices: TRUE, FALSE, and RED. The edges of the graph are of two types: "literal" edges that are independent of the clauses and "clause" edges that depend on the clauses. The literal edges form a triangle on the special vertices and also form a triangle on x, ¬x, and RED for i = 1, 2,..., n.

d. Argue that in any 3-coloring c of a graph containing the literal edges, exactly one of a variable and its negation is colored c(TRUE) and the other is colored c(FALSE).
Argue that for any truth assignment for φ, there is a 3-coloring of the graph containing just the literal edges.

The widget is used to enforce the condition corresponding to a clause (x y z). Each clause requires a unique copy of the 5 vertices that are heavily shaded; they connect as shown to the literals of the clause and the special vertex TRUE.

e. Argue that if each of x, y, and z is colored c(TRUE) or c(FALSE), then the widget is 3-colorable if and only if at least one of x, y, or z is colored c(TRUE).

f. Complete the proof that 3-COLOR is NP-complete.

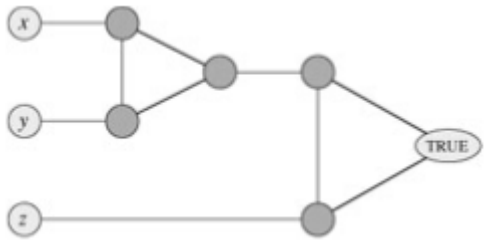


Fig: The widget corresponding to a clause (x y z), used in Problem

4.5. HAMILTONIAN CIRCUIT PROBLEM

As our next example, let us consider the problem of finding a Hamiltonian circuit in the graph of Figure 11.3a. Without loss of generality, we can assume that if a Hamiltonian circuit exists, it starts at vertex a. Accordingly, we make vertex a the root of the state-space tree (Figure 11.3b).

The first component of our future solution, if it exists, is a first intermediate vertex of a Hamiltonian cycle to be constructed. Using the alphabet order to break the three-way tie among the vertices adjacent to a, we select vertex b. From b, the algorithm proceeds to c, then to d, then to e, and finally to f, which proves to be a dead end. So the algorithm backtracks from f to e, then to d. and then to c, which provides the first alternative for the algorithm to pursue.

Going from c to e eventually proves useless, and the algorithm has to backtrack from e to c and then to b. From there, it goes to the vertices f, e, c, and d, from which it can legitimately return to a, yielding the Hamiltonian circuit a, b, f, e, c, d, a. If we wanted to find another Hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.

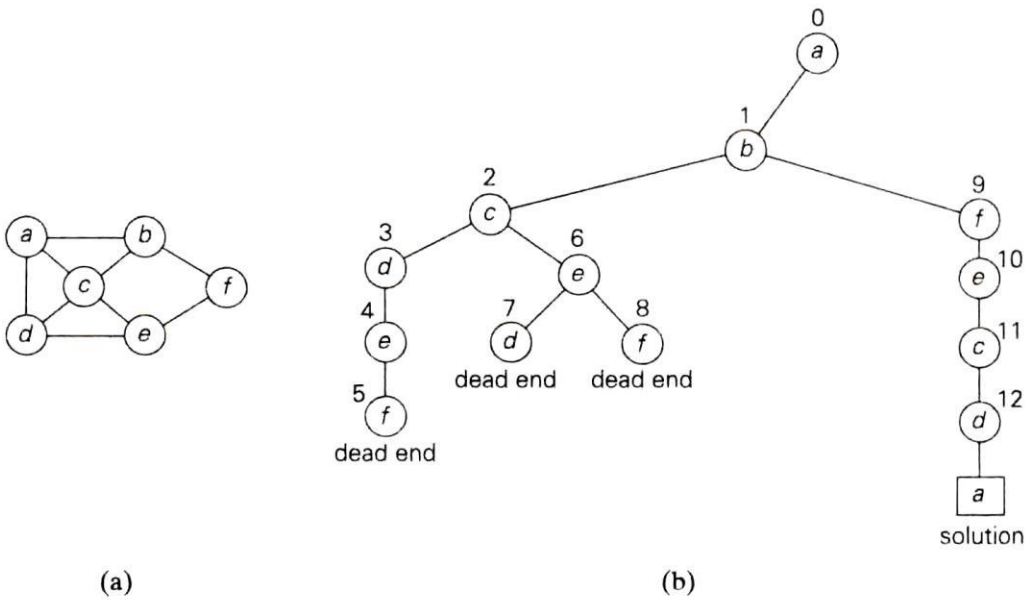


Figure 11.3: (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

4.6. KNAPSACK PROBLEM

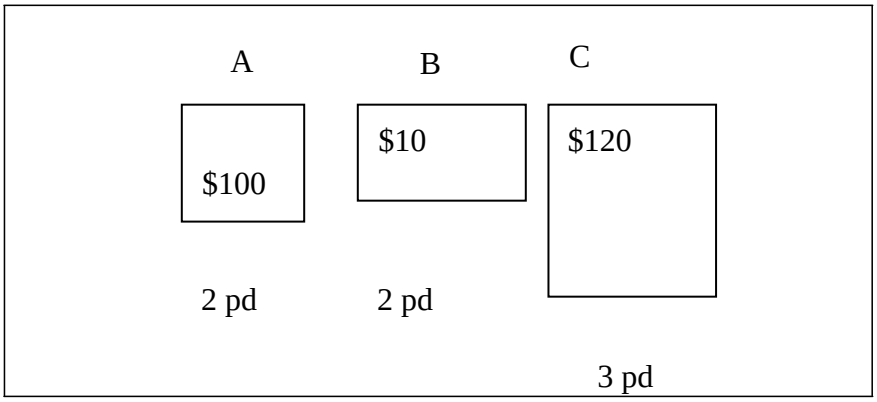
The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.

It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.

The problem often arises in resource allocation with financial constraints. A similar problem also appears in combinatorics, complexity theory, cryptography and applied mathematics.

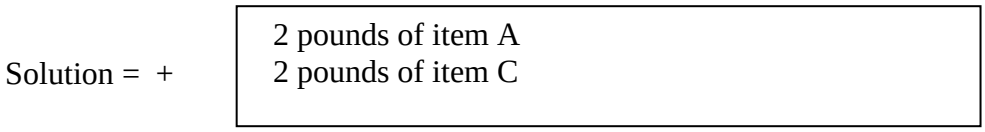
The decision problem form of the knapsack problem is the question "can a value of at least V be achieved without exceeding the weight W?"

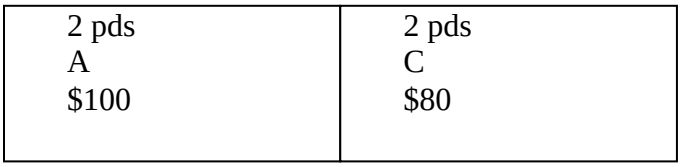
E.g. A thief enters a store and sees the following items:



His Knapsack holds 4 pounds.
What should he steal to maximize profit?

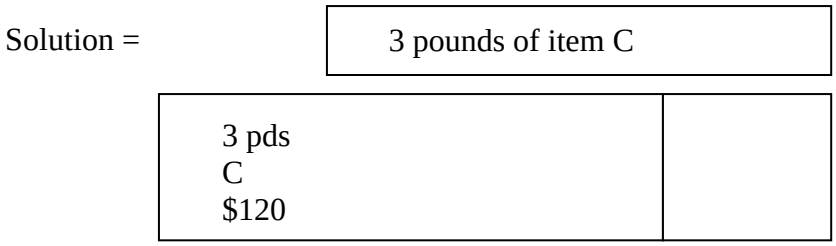
Fractional Knapsack Problem
Thief can take a fraction of an item.





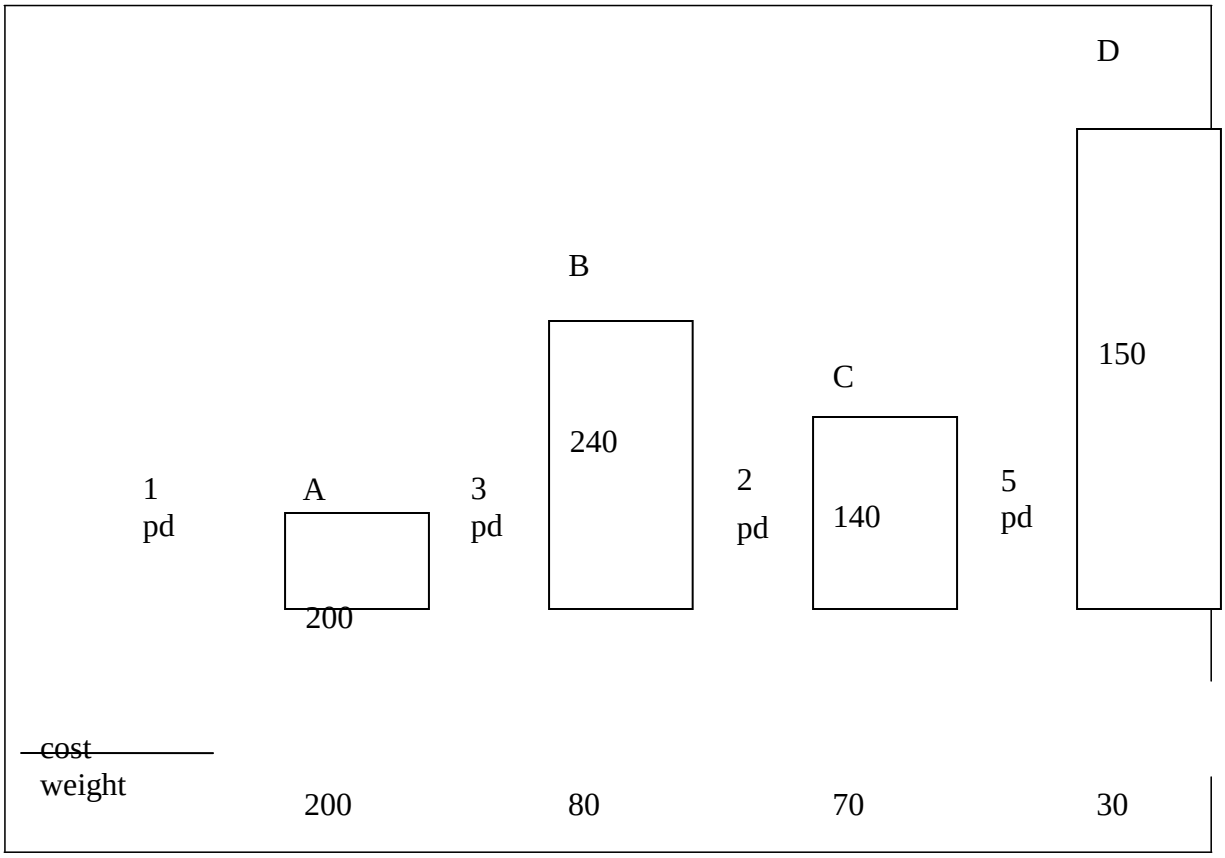
0-1 Knapsack Problem

Thief can only take or leave item. He can‘t take a fraction.



Fractional Knapsack has a greedy solution

Sort items by decreasing cost per pound



If knapsack holds k=5 pds, solution is:

		1 pds	A	
		3 pds	B	
		1 pds	C	

General Algorithm-O(n):
Given:

weight	w1	w2	…	wn
cost	c1	c2	…	cn

Knapsack weight limit K

1. Calculate vi = ci / wi for i = 1, 2, …, n
2. Sort the item by decreasing vi
3. Find j, s.t.

$w1 + w2 + \ldots + wj \le k < w1 + w2 + \ldots + wj+1$

Answer is { wi pds item i, for $i \le j$
$K-\Sigma i \le j$ wi pds item j+1

IMPORTANT QUESTIONS

PART-A

1. Define Backtracking.
2. What are the applications of backtracking?
3. What are the algorithm design techniques?
4. Define n-queens problem.
5. Define Hamiltonian Circuit problem.
6. Define sum of subset problem.
7. What is state space tree?
8. Define Branch & Bound method.
9. Define assignment problem.
10. What is promising & non-promising node?
11. Define Knapsack‘s problem.
12. Define Travelling salesman problem.
13. State principle of backtracking.
14. Compare Backtracking & Branch and Bound techniques with an example.
15. What are the applications of branch & bound?(or) What are the examples of branch & bound?
16.In Backtracking method,how the problem can be categorized?
17.How should be determine the solution in backtracking algorithm?
18.Obtain all possible solutions to 4-Queen‘s problem.
19.Generate atleast 3-solutions for 5-Queen‘s problem.
20.Draw a pruned state space tree for a given sum of subset problem: S={3,4,5,6} and d=13

PART-B:

1. Explain the n-Queen‘s problem & discuss the possible solutions. (16)
2. Solve the following instance of the knapsack problem by the branch & bound algorithm. (16)
3. Discuss the solution for Travelling salesman problem using branch & bound technique. (16)
4. Apply backtracking technique to solve the following instance of subset sum problem : S={1,3,4,5} and d=11 (16)
5. Explain subset sum problem & discuss the possible solution strategies using backtracking. (16)

UNIT V TRAVERSALS, BRANCH AND BOUND

Graph traversals – Connected components – Spanning trees – Biconnected components – Branch and Bound – General methods (FIFO and LC) – 0/1 Knapsack problem – Introduction to NP-hard and NP-completeness

5.1. GRAPH TRAVERSALS - BREADTH FIRST SEARCH

WORKING PRINCIPLE:

1. It starts from the arbitrary vertex
2. It visits all vertices adjacent to starting vertex.
3. Then all unvisited vertices between two edjes apart from it.
4. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

♦ Queue is used to trace BFS.

♦ The queue is initialized with the traversal's starting vertex, which is marked as visited. ♦ On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

5.1.1. BREADTH FIRST SEARCH FOREST

♦ Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called **breadth-first search forest**.

1. The traversal's starting vertex serves as the root of the first tree in such a forest.
2. New unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a **tree edge**.
3. If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as **a cross edge**.

♦ Here is a pseudo code of the breadth-first search.

ALGORITHM *BFS(G)*

//Implements a breadth-first search traversal of a given graph

//Input: Graph G = *(V, E)*

//Output: Graph G with its vertices marked with consecutive integers in the order they have been visited by the BFS traversal mark each vertex in *V* with 0 as a mark of being

"unvisited"
count ← 0
for each vertex v in V **do**
if v is marked with 0
bfs(v)
bfs(v)
//visits all the unvisited vertices connected to vertex v and assigns them the numbers in the order they are visited via global variable *count*
count ← count + 1; mark *v* with *count* and initialize a queue with *v*
while the queue is not empty **do**
for each vertex *w* in *V* adjacent to the front's vertex *v* **do**
if *w* is marked with 0
count ← count + 1; mark *w* with *count* add *w* to the queue
remove vertex *v* from the front of the queue

5.1.2. EFFICIENCY

♦ Breadth-first search has the same efficiency as depth-first search:

♦ BFS can be used to check the connectivity and acyclicity of a graph as same as DFS. for the adjacency linked list representation.

5.1.3. ORDERING OF VERTICES

♦ It yields a single ordering of vertices because the queue is a FIFO (first-in first-out) structure, and hence the order in which vertices are added to the queue is the same order in which they are removed from it.

♦ As to the structure of a BFS forest, it can also have two kinds of edges: tree edges and cross edges. Tree edges are the ones used to reach previously unvisited vertices. Cross edges connect vertices to those visited before but, unlike back edges in a DFS tree, they connect either siblings or cousins on the same or adjacent levels of a BFS tree.

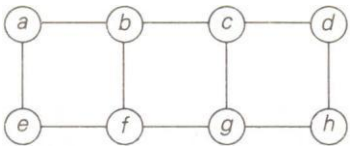
5.1.4. APPLICATION OF BFS

♦ Finally, BFS can be used to

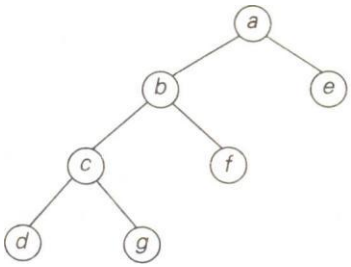
♦ Check connectivity and acydicity of a graph

♦ BFS can be used for finding a path with the fewest number of edges between two given vertices.

♦ This figure explains the BFS algorithm to find the minimum edge path.



(a)



(b)

a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from *a* to *g*.

5.1.5. PROCEDURE TO FIND THE FEWEST NUMBER OF EDGES BETWEEN TWO VERTICES:

♦ Start a BFS traversal at one of the given edge

♦ Stop it as soon as the other vertex is reached.

♦ For example, path a-b-e-g in the graph has the fewest number of edges among all the paths between

5.2. DEPTH FIRST SEARCH

WORKING PRINCIPLE

♦ Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited.

♦ On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in.

♦ The algorithm stops, when there is no unvisited adjacent unvisited vertex.

♦ At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.

♦ The algorithm eventually halts, when there is no unvisited unvisited vertex.

♦ Stack is used totrace the operation of depth-first search.

♦ Push a vertex onto the stack when the vertex is reached for the first time (i.e., the visit of the vertex starts), and pop a vertex off the stack when it becomes a dead end (i.e., the visit of the vertex ends).

5.2.1. DEPTH FIRST SEARCH FOREST

♦ It is also very useful to accompany a depth-first search traversal by constructing the so called **depth-first search forest**.

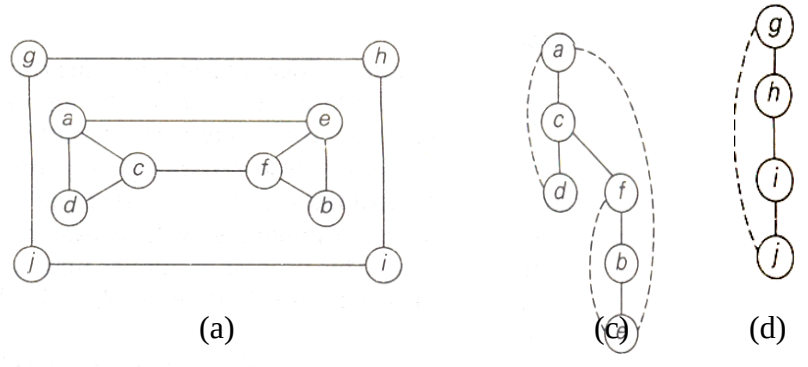
♦ The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a **tree edge** because the set of all such edges forms a forest.

♦ The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). Such an edge is called a **back edge** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest.

♦ Here is a pseudo code of the depth-first search.

5.2.2. ALGORITHM DFS(G)

Algorithm Dfs(G)
//Implements a depth-first search traversal of a given graph
//Input: Graph G = (V, E)
//0utput: Graph G with its vertices marked with consecutive integers in the order they've been first encountered by the DFS traversal mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for** each vertex v in V **do
if v is marked with 0
dfs (v)
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v and assigns them the numbers in the order they are encountered via global variable count
count ←count + 1; mark v with count
for** each vertex w in V adjacent to v **do
***if** w is marked with 0*
dfs(w)



(b)

Fig: Example of a DFS traversal

(a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto the stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest (with the tree edges shown with solid lines and the back edges shown with dashed lines).

♦ A DFS traversal itself and the forest-like representation of a graph it provides have proved to be extremely helpful for the development of efficient algorithms for checking many important properties of graphs.

♦ DFS yields two orderings of vertices:

1. The order in which the vertices are reached for the first time (pushed onto the stack)
2. The order in which the vertices become dead ends (popped off the stack).

These orders are qualitatively different, and various applications can take advantage of either of them.

5.2.3. APPLICATIONS OF DFS

♦ DFS is used for

- ♦ Checking connectivity of the graph
- ♦ Checking a cyclicity of a graph.

Checking connectivity of the graph

- ♦ Checking a graph's connectivity can be done as follows.
- ♦ Start a DFS traversal at an arbitrary vertex
- ♦ Check, after the algorithm halts, whether all the graph's vertices will have been visited.
- ♦ If they have, the graph is connected; otherwise, it is not connected

Checking a cyclicity of a graph

- ♦ If there is a back edge in DFS forest , then the graph is acyclic.
- ♦ If there is a back edge,from some vertex u to its ancestor *v* (e.g., the back edge from *d* to *a* in Figure 5.5c), the graph has a cycle that comprises the path from :J to *u* via a sequence of tree edges in the DFS forest followed by the back edge from *u* to *v*.

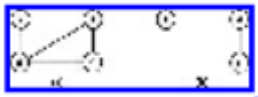
Articulation point

♦ A vertex of a connected graph is said to be its articulation point if its removal with all edges incident to it breaks the graph into disjoint pieces.

5.3. CONNECTED COMPONENT

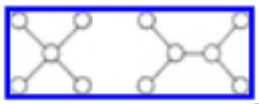
A connected component of a graph G is a maximal connected induced subgraph, that is, a connected induced subgraph that is not itself a proper subgraph of any other connected

subgraph of G



A graph and one of its subgraphs.

The above Figure is a connected graph. It has only one connected component, namely itself. Following figure is a graph with two connected components.



An unconnected graph.

5.3.1. BICONNECTED COMPONENTS

An articulation point of a graph is a vertex v such that when we remove v and all edges incident upon v , we break a connected component of the graph into two or more pieces.

A connected graph with no articulation points is said to be biconnected. Depth-first search is particularly useful in finding the biconnected components of a graph.

The problem of finding articulation points is the simplest of many important problems concerning the connectivity of graphs. As an example of applications of connectivity algorithms, we may represent a communication network as a graph in which the vertices are sites to be kept in communication with one another.

A graph has connectivity k if the deletion of any k-1 vertices fails to disconnect the graph. For example, a graph has connectivity two or more if and only if it has no articulation points, that is, if and only if it is biconnected.

The higher the connectivity of a graph, the more likely the graph is to survive the failure of some of its vertices, whether by failure of the processing units at the vertices or external attack.

We shall here give a simple depth-first search algorithm to find all the articulation points of a connected graph, and thereby test by their absence whether the graph is biconnected.

1. Perform a depth-first search of the graph, computing dfnumber [v] for each vertex v. In essence, dfnumber orders the vertices as in a preorder traversal of the depth-first spanning tree.

2. For each vertex v, compute low [v], which is the smallest dfnumber of v or of any vertex w reachable from v by following down zero or more tree edges to a descendant x of v (x may be v) and then following a back edge (x, w). We compute low [v] for all vertices v by visiting the vertices in a postorder traversal. When we process v, we have computed low [y] for every child y of v. We take low [v] to be the minimum of a. dfnumber [v],b. dfnumber [z] for any vertex z for which there is a back edge (v, z) and c. low [y] for any child y of v.

Now we find the articulation points as follows.

a. The root is an articulation point if and only if it has two or more children. Since there are no cross edges, deletion of the root must disconnect the subtrees rooted at its children, as a disconnects {b, d, e} from {c, f, g} in Fig..

b. A vertex v other than the root is an articulation point if and only if there is some child w of v such that low [w] ≥ dfnumber [v]. In this case, v disconnects w and its descendants from the rest of the graph. Conversely, if low [w] < dfnumber [v], then there must be a way to get from w down the tree and back to a proper ancestor of v (the vertex whose dfnumber is low [w]), and therefore deletion of v does not disconnect w or its descendants from the rest of the graph.

5.4. SPANNING TREES

A spanning tree for G is a free tree that connects all the vertices in V

5.4.1. MINIMUM-COST SPANNING TREES

Suppose G = (V, E) is a connected graph in which each edge (u, v) in E has a cost c(u, v) attached to it. The cost of a spanning tree is the sum of the costs of the edges in the tree.

A typical application for minimum-cost spanning trees occurs in the design of communications networks. The vertices of a graph represent cities and the edges possible communications links between the cities. The cost associated with an edge represents the cost of selecting that link for the network. A minimum-cost spanning tree represents a communications network that connects all the cities at minimal cost.

5.4.2. The MST Property

There are several different ways to construct a minimum-cost spanning tree. Many of these methods use the following property of minimum-cost spanning trees, which we call the MST property

Let G = (V, E) be a connected graph with a cost function defined on the edges. Let U be some proper subset of the set of vertices V. If (u, v) is an edge of lowest cost such that u

U and v ∈ V-U, then there is a minimum-cost spanning tree that includes (u, v) as an edge.

The proof hat every minimum-cost spanning tree satisfies the MST property is not hard. Suppose to the contrary that there is no minimum-cost spanning tree for G that includes (u, v). Let T be any minimum-cost spanning tree for G. Adding (u, v) to T must introduce a cycle, since T is a free tree and therefore satisfies property (2) for free trees. This cycle involves edge (u, v). Thus, there must e another edge (u', v') in T such that u' ∈ U and v' ∈ V-U, as illustrated in Fig. 7.5. If not, there could be no way for the cycle to get from u to v without following the edge (u, v) a second time. Deleting the edge (u', v') breaks the cycle and yields a spanning tree T' whose

5.4.3. PRIM'S ALGORITHM

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.

Two serious difficulties to construct Minimum Spanning Tree

1. The number of spanning trees grows exponentially with the graph size (at least for dense graphs).
2. Generating all spanning trees for a given graph is not easy.

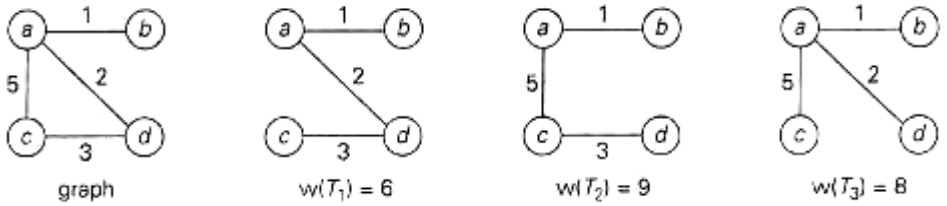


Figure: Graph and its spanning trees; T_1 is the Minimum Spanning Tree

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. 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, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. 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 used for the tree expansions.

The nature of Prim's algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex.

We can provide such information by attaching two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge.
Vertices that are not adjacent to any of the tree vertices can be given the label indicating their ―infinite‖ distance to the tree vertices a null label for the name of the nearest tree vertex.

With such labels, finding the next vertex to be added to the current tree $T = (V_T, E_T)$ become simple task of finding a vertex with the smallest distance label in the set $V - V_T$. 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 set V—V_T to the set of tree vertices V_T.
- For each remaining vertex U in V—V_T - that is connected to u* by a shorter edge than the u's current distance label, update its labels by u* and the weight of the edge between u* and u, respectively.

Pseudocode of this algorithm

The following figure demonstrates the application of Prim's algorithm to a specific graph.

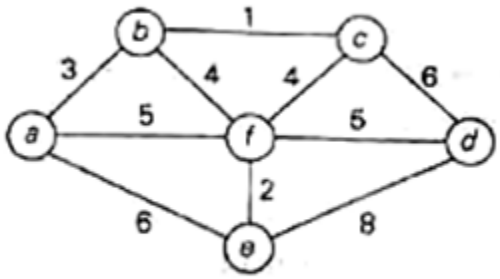


Fig: Graph given to find minimum spanning tree

5.4.4. KRUSKAL'S ALGORITHM

This is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution.

Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G = {V, E} as an acyclic subgraph 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 subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

ALGORITHM *Kruskal*(G)

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//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_i1) ≤ ... ≤ w(e_i|E|)
E_T ← ∅;  ecounter ← 0   //initialize the set of tree edges and its size
k ← 0                    //initialize the number of processed edges
while ecounter < |V| − 1
    k ← k + 1
    if E_T ∪ {e_ik} is acyclic
        E_T ← E_T ∪ {e_ik};  ecounter ← ecounter + 1
return E_T
```

The correctness of Kruskal's algorithm can be proved by repeating the essential steps of the proof of Prim's algorithm. The fact that E_T is actually a tree in Prim's algorithm, but generally just an acyclic subgraph in Kruskal's algorithm turns out to be an obstacle that can be overcome.

Applying Prim's and Kruskal's algorithms to the same small graph by hand may create an impression that the latter is simpler than the former. This impression is wrong because, on each of its iterations, Kruskal's algorithm has to check whether the addition of the next edge to the edges already selected would create a cycle.

It is not difficult to see that a new cycle is created if and only if the new edge connects two vertices already connected by a path, i.e., if and only if the two vertices belong to the same connected component. Note also that each connected component of a subgraph generated by Kruskal's algorithm is a tree because it has no cycles.

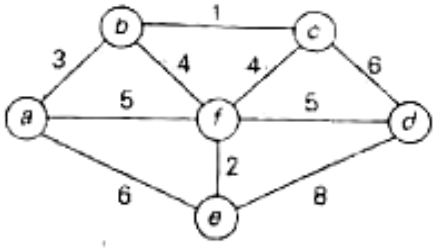


Fig: Graph given to find minimum spanning tree

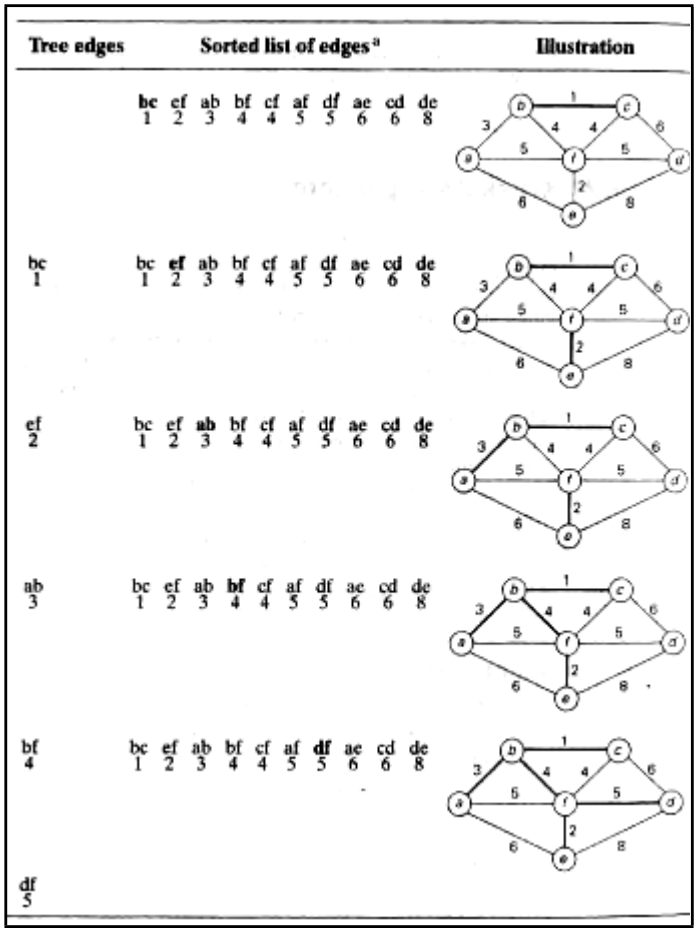


Fig: Application of Kruskal's algorithm

5.5. BRANCH-AND-BOUND

Backtracking 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, one that seeks to minimize or maximize an objective function, usually subject to some constraints.

Note that in the standard terminology of optimization problems, a feasible solution is a

point in the problem‘s search space that satisfies all the problem‘s constraints

(e.g. a Hamiltonian circuit in the traveling salesman problem, a subset of items whose total weight does not exceed the knapsack‘s capacity), while an optimal solution is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit, the most valuable subset of items that fit the knapsack).

Compared to backtracking, branch-and-bound requires two additional items.

A way to provide, for every node of a state-space tree, a bound on the best value of the objective functions on any solution that can be obtained by adding further components to the partial solution represented by the node

The value of the best solution seen so far

5.6. GENERAL METHOD

If this information is available, we can compare a node‘s bound value with the value of the best solution seen so far:

if the bound value is not better than the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated (some people say the branch is pruned) because no solution obtained from it can yield a better solution than the one already available.

E.g. Termination of search path

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.

5.6.1. 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-by-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.

Demonstrate of solving this problem using the branch-and-bound

This is done by considering the same small instance of the problem:

	Job 1	Job 2	Job 3	Job 4	
$C =$	9	2	7	8	Person a
	6	4	3	7	Person b
	5	8	1	8	Person c
	7	6	9	4	Person d

To find a lower bound on the cost of an optimal selection without actually solving the problem, we can do 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.

It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection.

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.

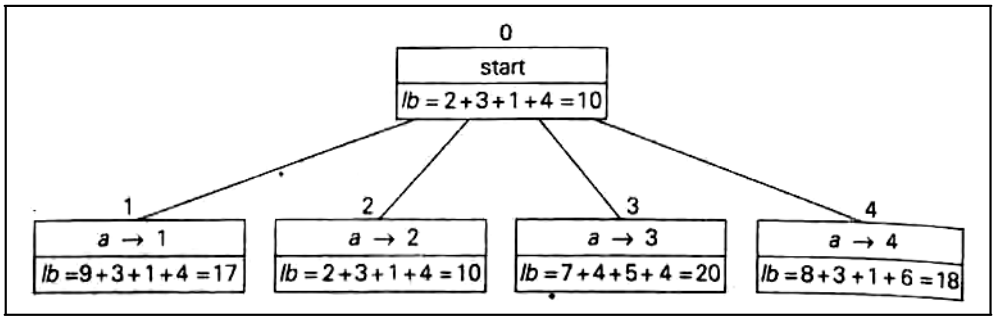


Fig: Levels 0 and 1 of the State-space tree for the instance of the assignment problem
(being solved with the best-first branch-and- bound algorithm. The number above a node shows the order in which the node was generated. A node‘s fields indicate the job number assigned to person *a*, and the lower bound value, *lb* for this node.)

This problem deals with the order in which the tree‘s nodes will he generated. Rather than generating a single child of the last promising node as we did in backtracking, we wilt generate all the children of the most promising node among non-terminated leaves in the current tree. (Non-terminated, je., still promising, leaves are also called live.)

To find which of the nodes is most promising, we are comparing the lower bounds of the live node. 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**. Returning to the instance of the assignment problem given earlier, we start with the root that corresponds to no elements selected from the cost matrix. As the lower bound value for the root, denoted *lb* is 10.

The nodes on the first level of the free correspond to four elements (jobs) in the first row of the matrix since they are each a potential selection for the first component of the solution. 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 lower bound 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.

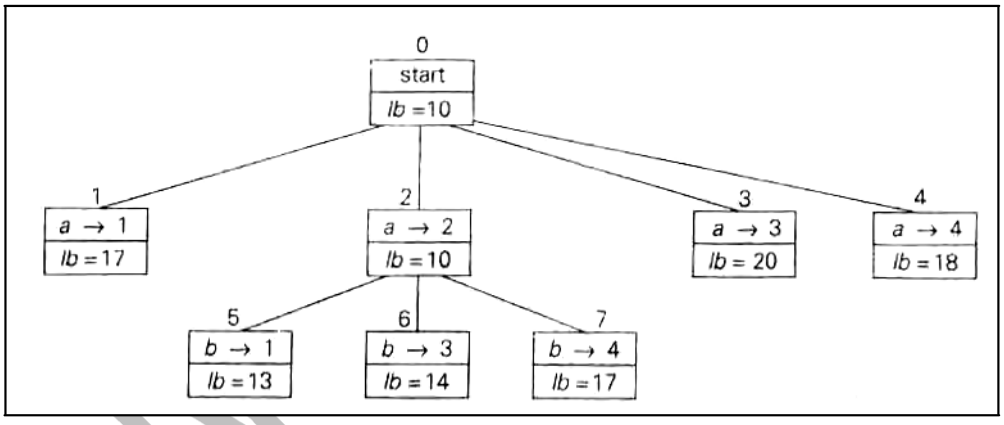


Figure: Levels 0, 1. and 2 of the state-space tree for the instance of the assignment problem
(being solved with the best-first branch-and- bound algorithm)

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 leafs that corresponds to the feasible solution (a →2, b→1, c→3, d →4) with (he 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 leafs, node 9 is simply terminated.

Note that 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.

Now, as we inspect each of the live leaves of the last state-space tree (nodes 1, 3, 4, 6, and 7 in the following figure), 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 optima) solution to the problem.

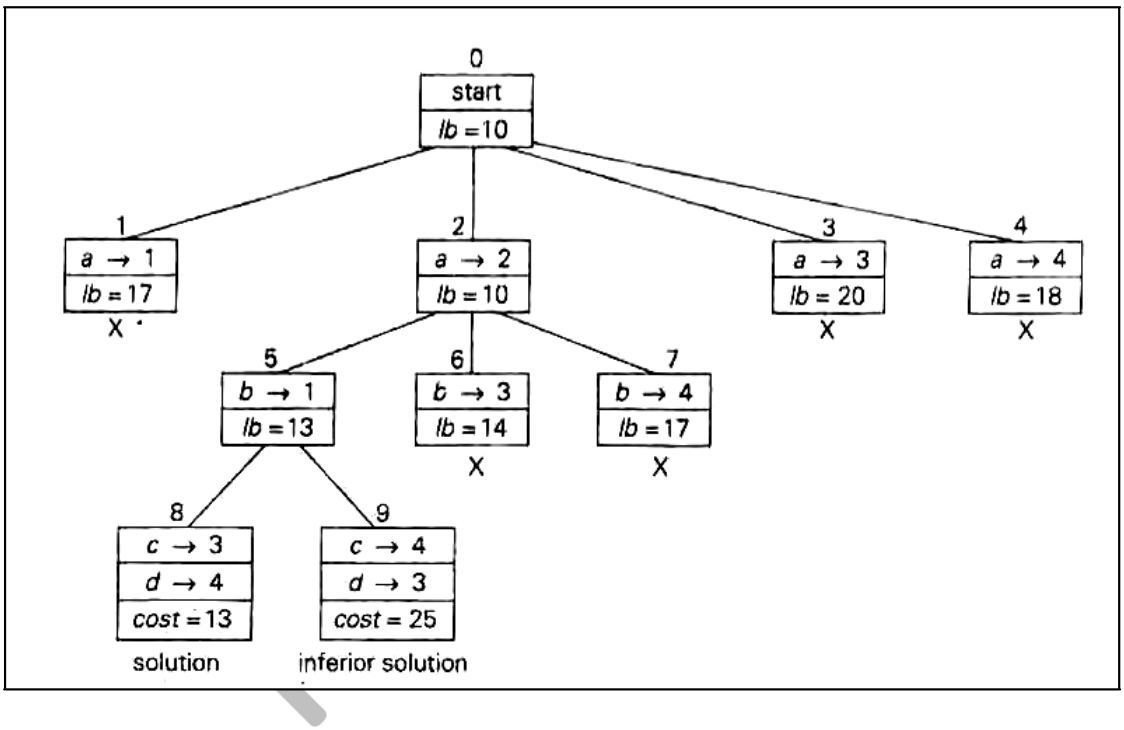


Figure: Complete state-space tree for the instance of the assignment problem
(Solved with the best-first branch-and-bound algorithm)

5.7. KNAPSACK PROBLEM

Illustration

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. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios.
Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n.$$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows (following figure).
Each node on the *i*th level of this tree, $0 \leq i \leq n$, 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 a path from the root to the node: a branch going to the left indicates the inclusion of the next item while the 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 v_{i+1}/w_{i+1} :

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$

As a specific example, let us apply the branch-and-bound algorithm to die same instance of the knapsack problem , At the root of the state-space tree (in the following figure), 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 computed by formula $\boldsymbol{(ub=v+(W-w)(v_{i+1}/w_{i+1})}$ 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.

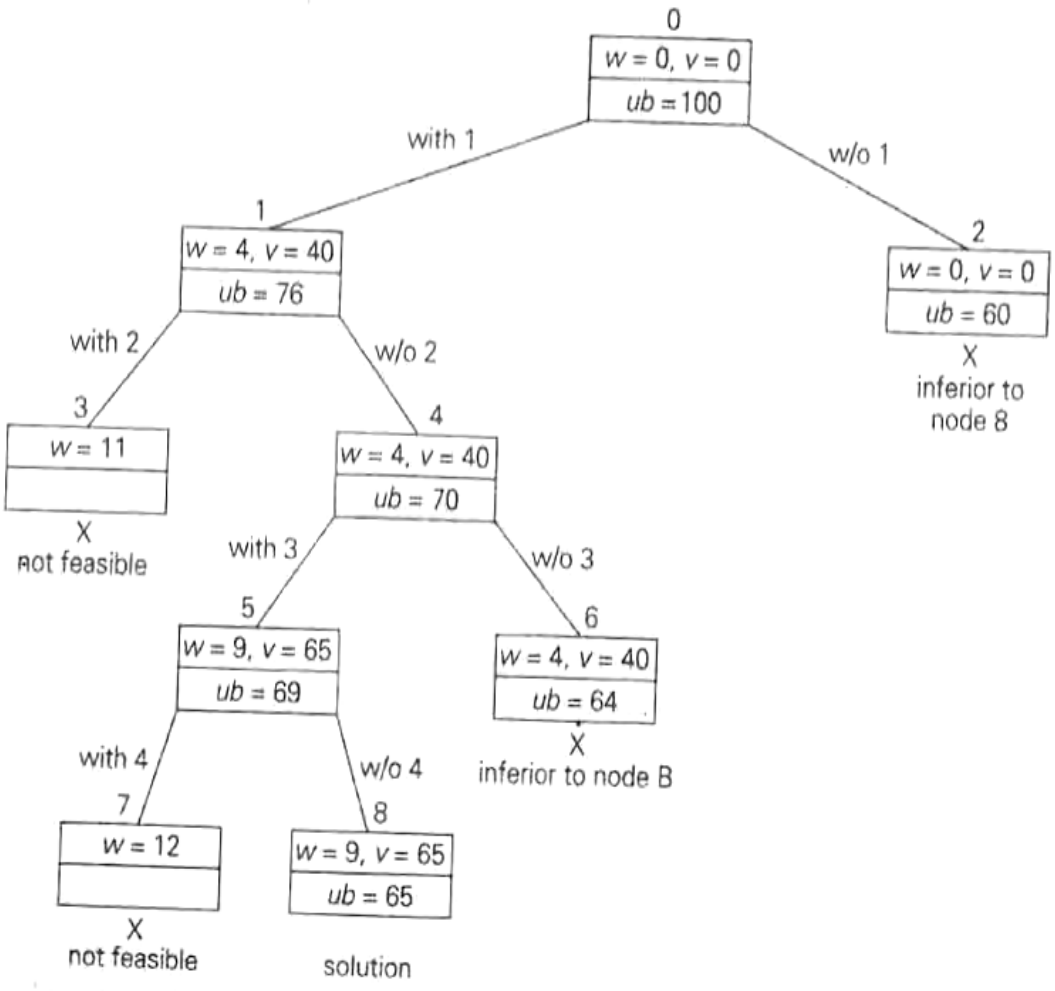


Fig: State-space tree of the branch-and-bound algorithm for the instance of the knapsack problem

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 *u* 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, 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, represents no feasible solutions and node 8 that represents just a single subset {1, 3}.

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.

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. 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 did this for the instance investigated above, we could have terminated nodes 2 & 6 before node 8 was generated because they both are inferior to the subset of value $65 of node 5.

5.8. INTRODUCTION TO NP-HARD AND NP-COMPLETENESS

P: the class of decision problems that are solvable in O($p(n)$) time, where $p(n)$ is a polynomial of problem‘s input size n

Examples:

- searching
- element uniqueness
- graph connectivity
- graph acyclicity

primality testing (finally proved in 2002)

NP (*nondeterministic polynomial*): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:

- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

E.g. Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true?

This problem is in *NP*. Nondeterministic algorithm:

- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

Example: (A | ¬B | ¬C) & (A | B) & (¬B | ¬D | E) & (¬D | ¬E)

Truth assignments:

A B C D E

0 0 0 0 0

. . .

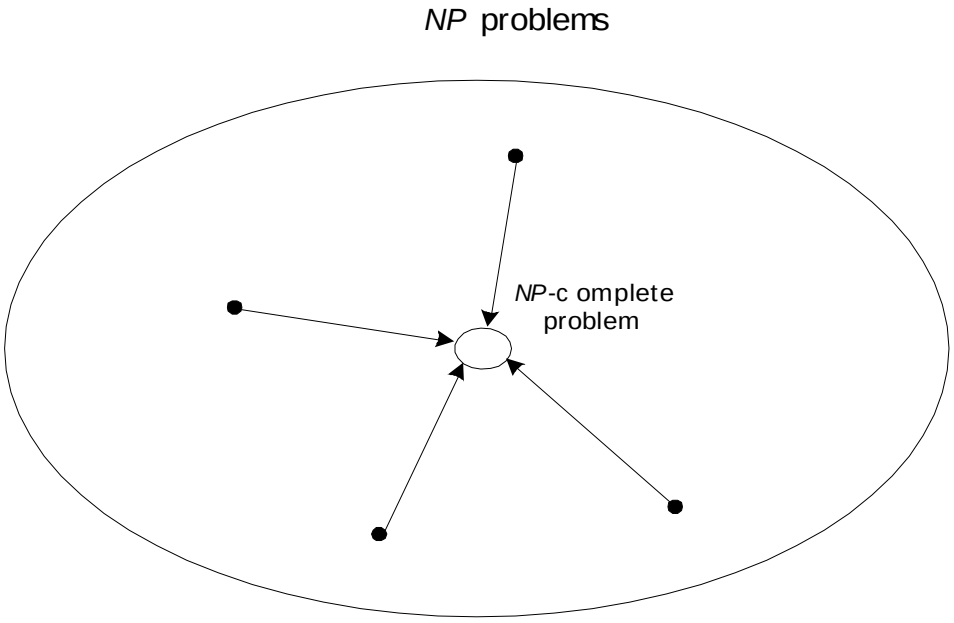
1 1 1 1 1

Checking phase: O(*n*)

5.8.1. PROBLEMS ARE IN *NP*?

- Hamiltonian circuit existence
- Partition problem: Is it possible to partition a set of *n* integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)
- All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we have:

  $P \subseteq NP$
- Big question: $P = NP$?



NP Hard problem:

- Most problems discussed are efficient (poly time)
- An interesting set of hard problems: NP-complete.

- A decision problem *D* is *NP*-complete if it's as hard as any problem in *NP*, i.e.,
  - *D* is in *NP*
  - every problem in *NP* is polynomial-time reducible to *D*

Cook's theorem (1971): CNF-sat is *NP*-complete

Other *NP*-complete problems obtained through polynomial-time reductions from a known *NP*-complete problem

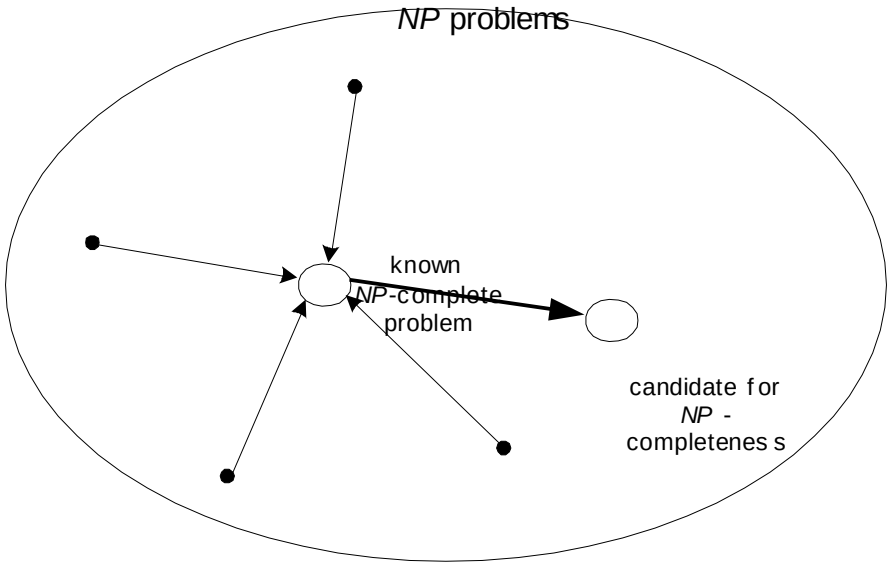


Fig: NP Problems

Examples:

TSP, knapsack, partition, graph-coloring and hundreds of other problems of combinatorial nature

- *P* = *NP* would imply that every problem in *NP*, including all *NP*-complete problems, could be solved in polynomial time
- If a polynomial-time algorithm for just one *NP*-complete problem is discovered,

then every problem in *NP* can be solved in polynomial time, i.e., $P = NP$

- Most but not all researchers believe that $P \neq NP$, i.e. P is a proper subset of NP

IMPORTANT QUESTIONS

PART-A

1. what is back tracking ?
2. Define n-queen‘s problem.
3. Define hamiltonian circuit problem
4. what is subset-sum problem.
5. Define branch and bound.
6. Knapsack problem.
7. Traveling salesman problem.

PART-B

1. Explain Graph traversals.
2. Explain in detail about Knapsack problem.
3. Explain traveling salesman problem with example.

M 2032

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2006.

Third Semester

Computer Science and Engineering

CS 1201 — DESIGN AND ANALYSIS OF ALGORITHMS

(Common to B.E. (P.T.) R 2005 Second Semester Computer Science and Engineering)

(Regulation 2004)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Define algorithm.
2. What is Big 'Oh' notation?
3. What is an activation frame?
4. Define external path length.
5. Give the recurrence equation for the worst case behavior of merge sort.
6. Name any two methods for pattern matching.
7. When do you say a tree as minimum spanning tree?
8. How will you construct an optimal binary search tree?
9. Define backtracking.
10. What is Hamiltonian cycle in an undirected graph?

PART B — (5 × 16 = 80 marks)

11. (i) Explain the various criteria used for analyzing algorithms. (10)

(ii) List the properties of various asymptotic notations. (6)

12. (a) (i) Explain the necessary steps for analyzing the efficiency of recursive algorithms. (10)

(ii) Write short notes on algorithm visualization. (6)

Or

(b) (i) Explain the activation frame and activation tree for finding the fibonacci series. (10)

(ii) What are the pros and cons of the empirical analysis of algorithm? (6)

13. (a) (i) Sort the following set of elements using Quick Sort. (10)

12, 24, 8, 71, 4, 23, 6

(ii) Give a detailed note on divide and conquer techniques. (6)

Or

(b) (i) Write an algorithm for searching an element using binary search method. Give an example. (12)

(ii) Compare and contrast BFS and DFS. (4)

14. (a) Explain the method of finding the minimum spanning tree for a connected graph using Prim's algorithm. (16)

Or

(b) How will you find the shortest path between two given vertices using Dijkstra's algorithm? Explain. (16)

15. (a) (i) Describe the travelling salesman problem and discuss how to solve it using dynamic programming. (10)

(ii) Write short notes on n–Queen's problem. (6)

Or

(b) Discuss the use of Greedy method in solving Knapsack problem and subset-sum problem. (16)

PART B — (5 × 16 = 80 marks)

11. (a) Describe briefly the notions of complexity of an algorithm. (16)

Or

(b) (i) What is pseudo-code? Explain with an example. (8)

(ii) Find the complexity (C(n)) of the algorithm for the worst case, best case and average case. (evaluate average case complexity for n = 3, where n is number of inputs) (8)

12. (a) Write an algorithm for a given numbers n to generate the n^{th} number of the Fibonacci sequence. (16)

Or

(b) Explain the pros and cons of the empirical analysis of algorithm. (16)

13. (a) (i) Write an algorithm to sort a set of 'N' numbers using insertion sort. (8)

(ii) Trace the algorithm for the following set of numbers : 20, 35, 18, 8, 14, 41, 3, 39. (8)

Or

(b) (i) Explain the difference between depth-first and breadth-first searches. (8)

(ii) Mention any three search algorithms which is preferred in general? Why? (8)

14. (a) Write the Dijikstra's algorithm. (16)

(b) Explain KRUSKAL'S algorithm. (16)

15. (a) What is back tracking? Explain in detail (16)

GLOSSARY TERMS

Acceptance Testing: Testing conducted to enable a user/customer to determine whether to accept a software product. Normally performed to validate the software meets a set of agreed acceptance criteria.

Accessibility Testing: Verifying a product is accessible to the people having disabilities (deaf, blind, mentally disabled etc.).

Ad Hoc Testing: A testing phase where the tester tries to 'break' the system by randomly trying the system's functionality. Can include negative testing as well.

Agile Testing: Testing practice for projects using agile methodologies, treating development as the customer of testing and emphasizing a test-first design paradigm.

Application Binary Interface (ABI): A specification defining requirements for portability of applications in binary forms across defferent system platforms and environments.

Application Programming Interface (API): A formalized set of software calls and routines that can be referenced by an application program in order to access supporting system or network services.

Automated Software Quality (ASQ): The use of software tools, such as automated testing tools, to improve software quality.

Automated Testing:

- Testing employing software tools which execute tests without manual intervention. Can be applied in GUI, performance, API, etc. testing.
- The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.

Backus-Naur Form: A metalanguage used to formally describe the syntax of a language.

Basic Block: A sequence of one or more consecutive, executable statements containing no branches.

Basis Path Testing: A white box test case design technique that uses the algorithmic flow of the program to design tests.

Basis Set: The set of tests derived using basis path testing.

Baseline: The point at which some deliverable produced during the software engineering process is put under formal change control.

Benchmark Testing: Tests that use representative sets of programs and data designed to evaluate the performance of computer hardware and software in a given configuration.

Beta Testing: Testing of a rerelease of a software product conducted by customers.

Binary Portability Testing: Testing an executable application for portability across system platforms and environments, usually for conformation to an ABI specification.

Black Box Testing: Testing based on an analysis of the specification of a piece of software without reference to its internal workings. The goal is to test how well the component conforms to the published requirements for the component.

Bottom Up Testing: An approach to integration testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.

Boundary Testing: Test which focus on the boundary or limit conditions of the software being tested. (Some of these tests are stress tests).

Boundary Value Analysis: In boundary value analysis, test cases are generated using the extremes of the input domaini, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values. BVA is similar to Equivalence Partitioning but focuses on "corner cases".

Branch Testing: Testing in which all branches in the program source code are tested at least once.

Breadth Testing: A test suite that exercises the full functionality of a product but does not test features in detail.

Bug: A fault in a program which causes the program to perform in an unintended or unanticipated manner.

CAST: Computer Aided Software Testing.

Capture/Replay Tool: A test tool that records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time. Most commonly applied to GUI test tools.

CMM: The Capability Maturity Model for Software (CMM or SW-CMM) is a model for judging the maturity of the software processes of an organization and for identifying the key practices that are required to increase the maturity of these processes.

Cause Effect Graph: A graphical representation of inputs and the associated outputs effects which can be used to design test cases.

Code Complete: Phase of development where functionality is implemented in entirety; bug fixes are all that are left. All functions found in the Functional Specifications have been implemented.

Code Coverage: An analysis method that determines which parts of the software have been executed (covered) by the test case suite and which parts have not been executed and therefore may require additional attention.

Code Inspection: A formal testing technique where the programmer reviews source code with a group who ask questions analyzing the program logic, analyzing the code with respect to a checklist of historically common programming errors, and analyzing its compliance with coding standards.

Code Walkthrough: A formal testing technique where source code is traced by a group with a small set of test cases, while the state of program variables is manually monitored, to analyze the programmer's logic and assumptions.

Coding: The generation of source code.

Compatibility Testing: Testing whether software is compatible with other elements of a system with which it should operate, e.g. browsers, Operating Systems, or hardware.

Component: A minimal software item for which a separate specification is available.

Concurrency Testing: Multi-user testing geared towards determining the effects of accessing the same application code, module or database records. Identifies and measures the level of locking, deadlocking and use of single-threaded code and locking semaphores.

Conformance Testing: The process of testing that an implementation conforms to the specification on which it is based. Usually applied to testing conformance to a formal standard.

Context Driven Testing: The context-driven school of software testing is flavor of Agile Testing that advocates continuous and creative evaluation of testing opportunities in light of the potential information revealed and the value of that information to the organization right now.

Conversion Testing: Testing of programs or procedures used to convert data from existing systems for use in replacement systems.

Cyclomatic Complexity: A measure of the logical complexity of an algorithm, used in white-box testing.

Data Dictionary: A database that contains definitions of all data items defined during analysis.

Data Flow Diagram: A modeling notation that represents a functional decomposition of a system.

Data Driven Testing: Testing in which the action of a test case is parameterized by externally defined data values, maintained as a file or spreadsheet. A common technique in Automated Testing.

Debugging: The process of finding and removing the causes of software failures.

Defect: Nonconformance to requirements or functional / program specification

Dependency Testing: Examines an application's requirements for pre-existing software, initial states and configuration in order to maintain proper functionality.

Depth Testing: A test that exercises a feature of a product in full detail.

Dynamic Testing: Testing software through executing it..

Emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

Endurance Testing: Checks for memory leaks or other problems that may occur with prolonged execution.

End-to-End testing: Testing a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Equivalence Class: A portion of a component's input or output domains for which the component's behaviour is assumed to be the same from the component's specification.

Equivalence Partitioning: A test case design technique for a component in which test cases are designed to execute representatives from equivalence classes.

Error: A mistake in the system under test; usually but not always a coding mistake on the part of the developer.

Exhaustive Testing: Testing which covers all combinations of input values and preconditions for an element of the software under test.

Functional Decomposition: A technique used during planning, analysis and design; creates a functional hierarchy for the software.

Functional Specification: A document that describes in detail the characteristics of the product with regard to its intended features.

Functional Testing:

- Testing the features and operational behavior of a product to ensure they correspond to its specifications.
- Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

Glass Box Testing: A synonym for White Box Testing.

Gorilla Testing: Testing one particular module, functionality heavily.

Gray Box Testing: A combination of Black Box and White Box testing methodologies: testing a piece of software against its specification but using some knowledge of its internal workings.

High Order Tests: Black-box tests conducted once the software has been integrated.

Independent Test Group (ITG): A group of people whose primary responsibility is software testing.

Inspection: A group review quality improvement process for written material. It consists of two aspects; product (document itself) improvement and process improvement (of both document production and inspection).

Integration Testing: Testing of combined parts of an application to determine if they function together correctly. Usually performed after unit and functional testing. This type of testing is especially relevant to client/server and distributed systems.

Installation Testing: Confirms that the application under test recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power out conditions.

Localization Testing: This term refers to making software specifically designed for a specific locality.

Loop Testing: A white box testing technique that exercises program loops.

Metric: A standard of measurement. Software metrics are the statistics describing the structure or content of a program. A metric should be a real objective measurement of something such as number of bugs per lines of code.

Monkey Testing: Testing a system or an Application on the fly, i.e just few tests here and there to ensure the system or an application does not crash out.

Mutation Testing: Testing done on the application where bugs are purposely added to it.

Negative Testing: Testing aimed at showing software does not work. Also known as "test to fail". See also Positive Testing.

N+1 Testing: A variation of Regression Testing. Testing conducted with multiple cycles in which errors found in test cycle N are resolved and the solution is retested in test cycle N+1. The cycles are typically repeated until the solution reaches a steady state and there are no errors. **Path Testing:** Testing in which all paths in the program source code are tested at least once.

Performance Testing: Testing conducted to evaluate the compliance of a system or component with specified performance requirements. Often this is performed using an automated test tool to simulate large number of users. Also know as "Load Testing".

Positive Testing: Testing aimed at showing software works. Also known as "test to pass".

Quality Assurance: All those planned or systematic actions necessary to provide adequate confidence that a product or service is of the type and quality needed and expected by the customer.

Quality Audit: A systematic and independent examination to determine whether quality activities and related results comply with planned arrangements and whether these arrangements are implemented effectively and are suitable to achieve objectives.

Quality Circle: A group of individuals with related interests that meet at regular intervals to consider problems or other matters related to the quality of outputs of a process and to the correction of problems or to the improvement of quality.

Quality Control: The operational techniques and the activities used to fulfill and verify requirements of quality.

Quality Management: That aspect of the overall management function that determines and implements the quality policy.

Quality Policy: The overall intentions and direction of an organization as regards quality as formally expressed by top management.

Quality System: The organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.

Race Condition: A cause of concurrency problems. Multiple accesses to a shared resource, at least one of which is a write, with no mechanism used by either to moderate simultaneous access.

Ramp Testing: Continuously raising an input signal until the system breaks down.

Recovery Testing: Confirms that the program recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power out conditions.

Regression Testing: Retesting a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

Release Candidate: A pre-release version, which contains the desired functionality of the final version, but which needs to be tested for bugs (which ideally should be removed before the final version is released).

Sanity Testing: Brief test of major functional elements of a piece of software to determine if its basically operational.

Scalability Testing: Performance testing focused on ensuring the application under test gracefully handles increases in work load.

Security Testing: Testing which confirms that the program can restrict access to authorized personnel and that the authorized personnel can access the functions available to their security level.

Smoke Testing: A quick-and-dirty test that the major functions of a piece of software work. Originated in the hardware testing practice of turning on a new piece of hardware for the first time and considering it a success if it does not catch on fire.

Soak Testing: Running a system at high load for a prolonged period of time. For example, running several times more transactions in an entire day (or night) than would be expected in a busy day, to identify and performance problems that appear after a large number of transactions have been executed.

Software Requirements Specification: A deliverable that describes all data, functional and behavioral requirements, all constraints, and all validation requirements for software/

Software Testing: A set of activities conducted with the intent of finding errors in software.

Static Analysis: Analysis of a program carried out without executing the program.

Static Analyzer: A tool that carries out static analysis.

Static Testing: Analysis of a program carried out without executing the program.

Storage Testing: Testing that verifies the program under test stores data files in the correct directories and that it reserves sufficient space to prevent unexpected termination resulting from lack of space. This is external storage as opposed to internal storage.

Stress Testing: Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how. Often this is performance testing using a very high level of simulated load.

Structural Testing: Testing based on an analysis of internal workings and structure of a piece of software.

System Testing: Testing that attempts to discover defects that are properties of the entire system rather than of its individual components.

Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

Testing:

- The process of exercising software to verify that it satisfies specified requirements and to detect errors.
- The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs), and to evaluate the features of the software item (Ref. IEEE Std 829).
- The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

Test Automation:

Test Bed: An execution environment configured for testing. May consist of specific hardware, OS, network topology, configuration of the product under test, other application or system software, etc. The Test Plan for a project should enumerated the test beds(s) to be used.

Test Case:

- Test Case is a commonly used term for a specific test. This is usually the smallest unit of testing. A Test Case will consist of information such as requirements testing, test steps, verification steps, prerequisites, outputs, test environment, etc.
- A set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test Driven Development: Testing methodology associated with Agile Programming in which every chunk of code is covered by unit tests, which must all pass all the time, in an effort to eliminate unit-level and regression bugs during development. Practitioners of TDD write a lot of tests, i.e. an equal number of lines of test code to the size of the production code.

Test Driver: A program or test tool used to execute a tests. Also known as a Test Harness.

Test Environment: The hardware and software environment in which tests will be run, and any other software with which the software under test interacts when under test including stubs and test drivers.

Test First Design: Test-first design is one of the mandatory practices of Extreme Programming (XP).It requires that programmers do not write any production code until they have first written a unit test.

Test Harness: A program or test tool used to execute a tests. Also known as a Test Driver.

Test Plan: A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning. Ref IEEE Std 829.

Test Procedure: A document providing detailed instructions for the execution of one or more test cases.

Test Scenario: Definition of a set of test cases or test scripts and the sequence in which they are to be executed.

Test Script: Commonly used to refer to the instructions for a particular test that will be carried out by an automated test tool.

Test Specification: A document specifying the test approach for a software feature or combination or features and the inputs, predicted results and execution conditions for the associated tests.

Test Suite: A collection of tests used to validate the behavior of a product. The scope of a Test Suite varies from organization to organization. There may be several Test Suites for a particular product for example. In most cases however a Test Suite is a high level concept, grouping together hundreds or thousands of tests related by what they are intended to test.

Test Tools: Computer programs used in the testing of a system, a component of the system, or its documentation.

Thread Testing: A variation of top-down testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by successively lower levels.

Top Down Testing: An approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested.

Total Quality Management: A company commitment to develop a process that achieves high quality product and customer satisfaction.

Traceability Matrix: A document showing the relationship between Test Requirements and Test Cases.

Usability Testing: Testing the ease with which users can learn and use a product.

Use Case: The specification of tests that are conducted from the end-user perspective. Use cases tend to focus on operating software as an end-user would conduct their day-to-day activities.

User Acceptance Testing: A formal product evaluation performed by a customer as a condition of purchase.

Unit Testing: Testing of individual software components.

Validation: The process of evaluating software at the end of the software development process to ensure compliance with software requirements. The techniques for validation is testing, inspection and reviewing.

Verification: The process of determining whether of not the products of a given phase of the software development cycle meet the implementation steps and can be traced to the incoming objectives established during the previous phase. The techniques for verification are testing, inspection and reviewing.

Volume Testing: Testing which confirms that any values that may become large over time (such as accumulated counts, logs, and data files), can be accommodated by the program and will not cause the program to stop working or degrade its operation in any manner.

Walkthrough: A review of requirements, designs or code characterized by the author of the material under review guiding the progression of the review.

White Box Testing: Testing based on an analysis of internal workings and structure of a piece of software. Includes techniques such as Branch Testing and Path Testing. Also known as Structural Testing and Glass Box Testing. Contrast with Black Box Testing.

Workflow Testing: Scripted end-to-end testing which duplicates specific workflows which are expected to be utilized by the end-user.