

LECTURE NOTES ON DESIGN AND ANALYSIS OF ALGORITHMS

Department of Computer Science and Engineering



All JNTU World
Get The Most Out Of Imagineering

CONTENTS

CHAPTER 1 BASIC CONCEPTS

Algorithm
Performance of Programs
Algorithm Design Goals
Classification of Algorithms
Complexity of Algorithms
Rate of Growth
Analyzing Algorithms
The Rule of Sums
The Rule of products
The Running time of Programs
Measuring the running time of programs
Asymptotic Analyzing of Algorithms
Calculating the running time of programs
General rules for the analysis of programs

CHAPTER 2 Advanced Data Structures and Recurrence Relations

Priority Queue, Heap and Heap sort
Heap Sort
2.3 Priority Queue implementation using heap tree
Binary Search trees
Balanced Trees
Dictionary
Disjoint Set Operations
Recurrence Relations – Iterative Substitution Method
Recursion Tree
The Guess-and test
The Master Theorem Method
Cold Form expression
Solving Recurrence relations

CHAPTER 3 Divide And Conquer

General Method
Control Abstraction of Divide and Conquer
Binary Search
External and Internal path length
Merge Sort
Strassen's Matrix Multiplication
Quick Sort
Straight Insertion Sort

CHAPTER 4 Greedy Method

4.1 General Method
Control Abstraction
Knapsack Problem
Optimal Storage on Tapes
Job Sequencing with deadlines
Optimal Merge Patterns
Huffman Codes

Graph Algorithms

CHAPTER 5 *Dynamic programming*

Multi Storage graphs
All Pairs Shortest paths
Traveling Sales Person problem
Optimal Binary Search Tree
0/1 Knapsack
Reliability design

CHAPTER 6 *Basic Traversal and Search Techniques*

Techniques for traversal of Binary tree
Techniques for graphs
Representation of Graph and Digraphs
Depth First and Breadth First Spanning trees
Articulation Points and bi-connected components
Articulation points by Depth First Search
Game planning
Alpha-Beta pruning
AND/OR Graphs

CHAPTER 7 *Backtracking*

General method
Terminology
N-Queens problem
Sum of Subsets
Graph Coloring(for planar graphs)
Hamiltonian Cycles
0/1 Knapsack
Traveling Sales Person using Backtracking

CHAPTER 8 *Branch and Bound*

General method
Least Cost (LC) Search
Control Abstraction for LC-Search
Bounding
The 15-Puzzle problem
LC Search for 15-Puzzle Problem
Job Sequencing with deadlines
Traveling Sales Person problem
0/1 Knapsack

Chapter 1

Basic Concepts

Algorithm

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

Algorithm Design Goals

The three basic design goals that one should strive for in a program are:

1. Try to save Time
2. Try to save Space
3. Try to save Face

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to "save face" by preventing the program from locking up or generating reams of garbled data.

Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1. Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2 .

- n** When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.
- $n \cdot \log n$** This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.
- n^2** When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.
- n^3** Similarly, an algorithm that processes triples of data items (perhaps in a triple-nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.
- 2^n** Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute-force" solutions to problems. Whenever n doubles, the running time squares.

Complexity of Algorithms

The complexity of an algorithm M is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size ' n ' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size ' n '. Complexity shall refer to the running time of the algorithm.

The function $f(n)$, gives the running time of an algorithm, depends not only on the size ' n ' of the input data but also on the particular data. The complexity function $f(n)$ for certain cases are:

1. Best Case : The minimum possible value of $f(n)$ is called the best case.
2. Average Case : The expected value of $f(n)$.
3. Worst Case : The maximum value of $f(n)$ for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

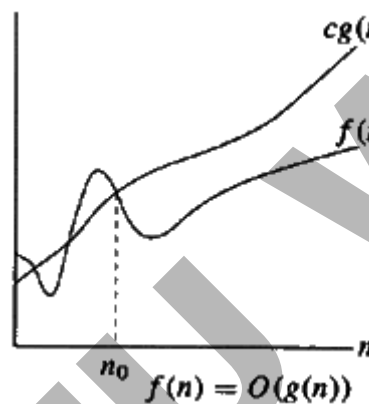
Rate of Growth:

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big-OH (O)¹,
2. Big-OMEGA (Ω),
3. Big-THETA (θ) and
4. Little-OH (o)

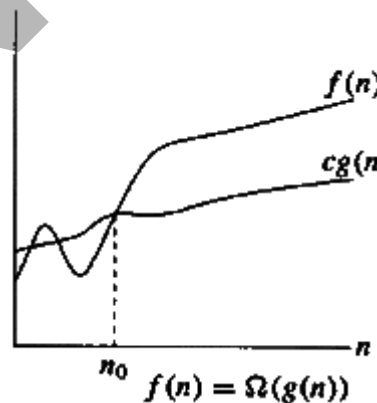
Big-OH O (Upper Bound)

$f(n) = O(g(n))$, (pronounced order of or big oh), says that the growth rate of $f(n)$ is less than or equal (\leq) that of $g(n)$.



Big-OMEGA Ω (Lower Bound)

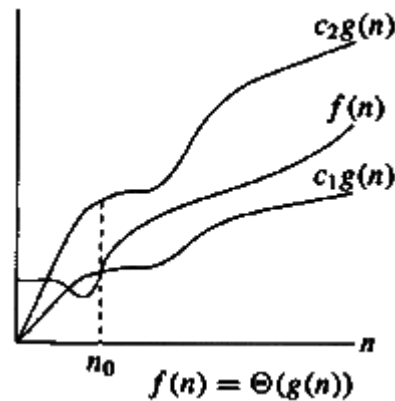
$f(n) = \Omega(g(n))$ (pronounced omega), says that the growth rate of $f(n)$ is greater than or equal (\geq) that of $g(n)$.



¹ In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as *big oh notation*.

Big-THETA Θ (Same order)

$f(n) = \Theta(g(n))$ (pronounced theta), says that the growth rate of $f(n)$ equals (=) the growth rate of $g(n)$ [if $f(n) = O(g(n))$ and $T(n) = \Omega(g(n))$].



Little-OH (o)

$T(n) = o(p(n))$ (pronounced little oh), says that the growth rate of $T(n)$ is less than the growth rate of $p(n)$ [if $T(n) = O(p(n))$ and $T(n) \neq \Theta(p(n))$].

Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity $f(n)$ of M increases as n increases. It is usually the rate of increase of $f(n)$ we want to examine. This is usually done by comparing $f(n)$ with some standard functions. The most common computing times are:

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n \cdot \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $n!$ and n^n

Numerical Comparison of Different Algorithms

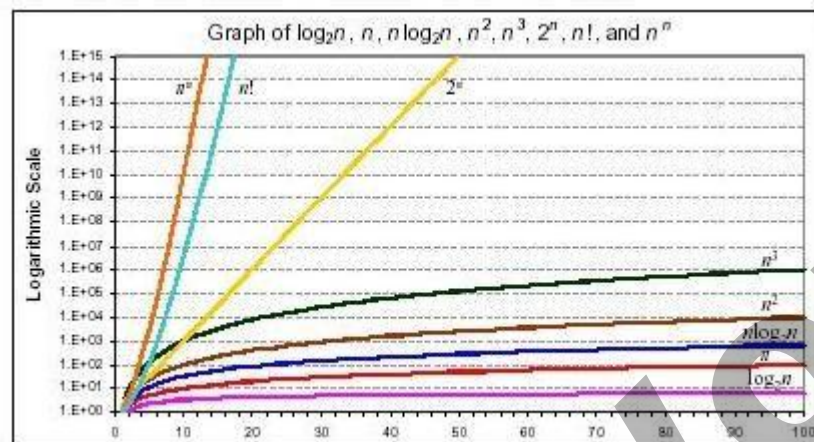
The execution time for six of the typical functions is given below:

n	$\log_2 n$	$n \cdot \log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billion years.

Graph of $\log n$, n , $n \log n$, n^2 , n^3 , 2^n , $n!$ and n^n



$O(\log n)$ does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low-order terms while computing a Big-Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function $f(n)$ with these standard function is to use the functional 'O' notation, suppose $f(n)$ and $g(n)$ are functions defined on the positive integers with the property that $f(n)$ is bounded by some multiple $g(n)$ for almost all 'n'. Then,

$$f(n) = O(g(n))$$

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is $O(n)$
- Binary search is $O(\log n)$
- Bubble sort is $O(n^2)$
- Merge sort is $O(n \log n)$

The rule of sums

Suppose that $T_1(n)$ and $T_2(n)$ are the running times of two programs fragments P_1 and P_2 , and that $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$. Then $T_1(n) + T_2(n)$, the running time of P_1 followed by P_2 is $O(\max f(n), g(n))$, this is called as rule of sums.

For example, suppose that we have three steps whose running times are respectively $O(n^2)$, $O(n^3)$ and $O(n \log n)$. Then the running time of the first two steps executed sequentially is $O(\max(n^2, n^3))$ which is $O(n^3)$. The running time of all three together is $O(\max(n^3, n \log n))$ which is $O(n^3)$.

The rule of products

If $T_1(n)$ and $T_2(n)$ are $O(f(n))$ and $O(g(n))$ respectively. Then $T_1(n) \cdot T_2(n)$ is $O(f(n) \cdot g(n))$. It follows from the product rule that $O(c \cdot f(n))$ means the same thing as $O(f(n))$ if 'c' is any positive constant. For example, $O(n^2/2)$ is same as $O(n^2)$.

Suppose that we have five algorithms A_1 – A_5 with the following time complexities:

$$\begin{aligned} A_1 &: n \\ A_2 &: n \log n \\ A_3 &: n^2 \\ A_4 &: n^3 \\ A_5 &: 2^n \end{aligned}$$

The time complexity is the number of time units required to process an input of size 'n'. Assuming that one unit of time equals one millisecond. The size of the problems that can be solved by each of these five algorithms is:

Algorithm	Time complexity	Maximum problem size		
		1 second	1 minute	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

The speed of computations has increased so much over last thirty years and it might seem that efficiency in algorithm is no longer important. But, paradoxically, efficiency matters more today than ever before. The reason why this is so is that our ambition has grown with our computing power. Virtually all applications of computing simulation of physical data are demanding more speed.

The faster the computer runs, the more need are efficient algorithms to take advantage of their power. As the computer becomes faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose the next generation of computers is ten times faster than the current generation, from the table we can see the increase in size of the problem.

Algorithm	Time Complexity	Maximum problem size before speed up	Maximum problem size after speed up
A_1	n	S_1	$10 S_1$
A_2	$n \log n$	S_2	$\approx 10 S_2$ for large S_2
A_3	n^2	S_3	$3.16 S_3$
A_4	n^3	S_4	$2.15 S_4$
A_5	2^n	S_5	$S_5 + 3.3$

Instead of an increase in speed consider the effect of using a more efficient algorithm. By looking into the following table it is clear that if minute as a basis for comparison, by replacing algorithm A_4 with A_3 , we can solve a problem six times larger; by replacing A_4 with A_2 we can solve a problem 125 times larger. These results are far more impressive than the two fold improvement obtained by a ten fold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant.

We therefore conclude that the asymptotic complexity of an algorithm is an important measure of the goodness of an algorithm.

The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

- i. We would like an algorithm that is easy to understand, code and debug.
- ii. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.
2. The quality of code generated by the compiler used to create the object program.
3. The nature and speed of the instructions on the machine used to execute the program, and
4. The time complexity of the algorithm underlying the program.

The running time depends not on the exact input but only the size of the input. For many programs, the running time is really a function of the particular input, and not just of the input size. In that case we define $T(n)$ to be the worst case running time, i.e. the maximum overall input of size 'n', of the running time on that input. We also consider $T_{avg}(n)$ the average, over all input of size 'n' of the running time on that input. In practice, the average running time is often much harder to determine than the worst case running time. Thus, we will use worst-case running time as the principal measure of time complexity.

Seeing the remarks (2) and (3) we cannot express the running time $T(n)$ in standard time units such as seconds. Rather we can only make remarks like the running time of such and such algorithm is proportional to n^2 . The constant of proportionality will remain un-specified, since it depends so heavily on the compiler, the machine and other factors.

Asymptotic Analysis of Algorithms:

Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

Rules for using big-O:

The most important property is that big-O gives an upper bound only. If an algorithm is $O(n^2)$, it doesn't have to take n^2 steps (or a constant multiple of n^2). But it can't

take more than n^2 . So any algorithm that is $O(n)$, is also an $O(n^2)$ algorithm. If this seems confusing, think of big-O as being like " $<$ ". Any number that is $< n$ is also $< n^2$.

1. Ignoring constant factors: $O(c f(n)) = O(f(n))$, where c is a constant; e.g. $O(20 n^3) = O(n^3)$
2. Ignoring smaller terms: If $a < b$ then $O(a+b) = O(b)$, for example $O(n^2+n) = O(n^2)$
3. Upper bound only: If $a < b$ then an $O(a)$ algorithm is also an $O(b)$ algorithm. For example, an $O(n)$ algorithm is also an $O(n^2)$ algorithm (but not vice versa).
4. n and $\log n$ are "bigger" than any constant, from an asymptotic view (that means for large enough n). So if k is a constant, an $O(n + k)$ algorithm is also $O(n)$, by ignoring smaller terms. Similarly, an $O(\log n + k)$ algorithm is also $O(\log n)$.
5. Another consequence of the last item is that an $O(n \log n + n)$ algorithm, which is $O(n(\log n + 1))$, can be simplified to $O(n \log n)$.

Calculating the running time of a program:

Let us now look into how big-O bounds can be computed for some common algorithms.

Example 1:

Let's consider a short piece of source code:

```
x = 3*y + 2;
z = z + 1;
```

If y, z are scalars, this piece of code takes a *constant* amount of time, which we write as $O(1)$. In terms of actual computer instructions or clock ticks, it's difficult to say exactly how long it takes. But whatever it is, it should be the same whenever this piece of code is executed. $O(1)$ means *some* constant, it might be 5, or 1 or 1000.

Example 2:

$2n^2 + 5n - 6 = O(2^n)$ $2n^2 + 5n - 6 = O(n^3)$ $2n^2 + 5n - 6 = O(n^2)$ $2n^2 + 5n - 6 \neq O(n)$	$2n^2 + 5n - 6 \neq \Theta(2^n)$ $2n^2 + 5n - 6 \neq \Theta(n^3)$ $2n^2 + 5n - 6 = \Theta(n^2)$ $2n^2 + 5n - 6 \neq \Theta(n)$
$2n^2 + 5n - 6 \neq \Omega(2^n)$ $2n^2 + 5n - 6 \neq \Omega(n^3)$ $2n^2 + 5n - 6 = \Omega(n^2)$ $2n^2 + 5n - 6 = \Omega(n)$	$2n^2 + 5n - 6 = o(2^n)$ $2n^2 + 5n - 6 = o(n^3)$ $2n^2 + 5n - 6 \neq o(n^2)$ $2n^2 + 5n - 6 \neq o(n)$

Example 3:

If the first program takes $100n^2$ milliseconds and while the second takes $5n^3$ milliseconds, then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5n^3 / 100n^2 = n/20$$

for inputs $n < 20$, the program with running time $5n^3$ will be faster than those the one with running time $100n^2$. Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is $n/20$, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as $O(n)$ or $O(n \log n)$ are always better.

Example 4:

Analysis of simple for loop

Now let's consider a simple for loop:

```
for (i = 1; i <= n; i++)  
    v[i] = v[i] + 1;
```

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n . We write it as $O(n)$. The actual number of instructions might be $50n$, while the running time might be $17n$ microseconds. It might even be $17n+3$ microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it's a linear function which is proportional to n , the correct notation is $O(n)$ and the code is said to have *linear* running time.

Example 5:

Analysis for nested for loop

Now let's look at a more complicated example, a nested for loop:

```
for (i = 1; i <= n; i++)  
    for (j = 1; j <= n; j++)  
        a[i,j] = b[i,j] * x;
```

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes $n \times n = n^2$ times. The assignment statement in the inner loop takes constant time, so the running time of the code is $O(n^2)$ steps. This piece of code is said to have *quadratic* running time.

Example 6:

Analysis of matrix multiply

Lets start with an easy case. Multiplying two $n \times n$ matrices. The code to compute the matrix product $C = A * B$ is given below.

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        C[i, j] = 0;
        for (k = 1; k <= n; k++)
            C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes $n * n * n = n^3$ times. The innermost statement, which contains a scalar sum and product takes constant $O(1)$ time. So the algorithm overall takes $O(n^3)$ time.

Example 7:

Analysis of bubble sort

The main body of the code for bubble sort looks something like this:

```
for (i = n-1; i > 1; i--)
    for (j = 1; j <= i; j++)
        if (a[j] > a[j+1])
            swap a[j] and a[j+1];
```

This looks like the double. The innermost statement, the *if*, takes $O(1)$ time. It doesn't necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here. The outer loop executes n times, but the inner loop executes a number of times that depends on i . The first time the inner *for* executes, it runs $i = n-1$ times. The second time it runs $n-2$ times, etc. The total number of times the inner *if* statement executes is therefore:

$$(n-1) + (n-2) + \dots + 3 + 2 + 1$$

This is the sum of an arithmetic series.

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

The value of the sum is $n(n-1)/2$. So the running time of bubble sort is $O(n(n-1)/2)$, which is $O((n^2-n)/2)$. Using the rules for big-O given earlier, this bound simplifies to $O((n^2)/2)$ by ignoring a smaller term, and to $O(n^2)$, by ignoring a constant factor. Thus, bubble sort is an $O(n^2)$ algorithm.

Example 8:

Analysis of binary search

Binary search is a little harder to analyze because it doesn't have a for loop. But it's still pretty easy because the search interval halves each time we iterate the search. The sequence of search intervals looks something like this:

$n, n/2, n/4, \dots, 8, 4, 2, 1$

It's not obvious how long this sequence is, but if we take logs, it is:

$\log_2 n, \log_2 n - 1, \log_2 n - 2, \dots, 3, 2, 1, 0$

Since the second sequence decrements by 1 each time down to 0, its length must be $\log_2 n + 1$. It takes only constant time to do each test of binary search, so the total running time is just the number of times that we iterate, which is $\log_2 n + 1$. So binary search is an $O(\log_2 n)$ algorithm. Since the base of the log doesn't matter in an asymptotic bound, we can write that binary search is $O(\log n)$.

General rules for the analysis of programs

In general the running time of a statement or group of statements may be parameterized by the input size and/or by one or more variables. The only permissible parameter for the running time of the whole program is 'n' the input size.

1. The running time of each assignment read and write statement can usually be taken to be $O(1)$. (There are few exemptions, such as in PL/1, where assignments can involve arbitrarily larger arrays and in any language that allows function calls in arraignment statements).
2. The running time of a sequence of statements is determined by the sum rule. I.e. the running time of the sequence is, to with in a constant factor, the largest running time of any statement in the sequence.
3. The running time of an if-statement is the cost of conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally $O(1)$ the time for an if-then-else construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.
4. The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination (usually the latter is $O(1)$). Often this time is, neglected constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure.

Chapter 2

Advanced Data Structures and Recurrence Relations

Priority Queue, Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.

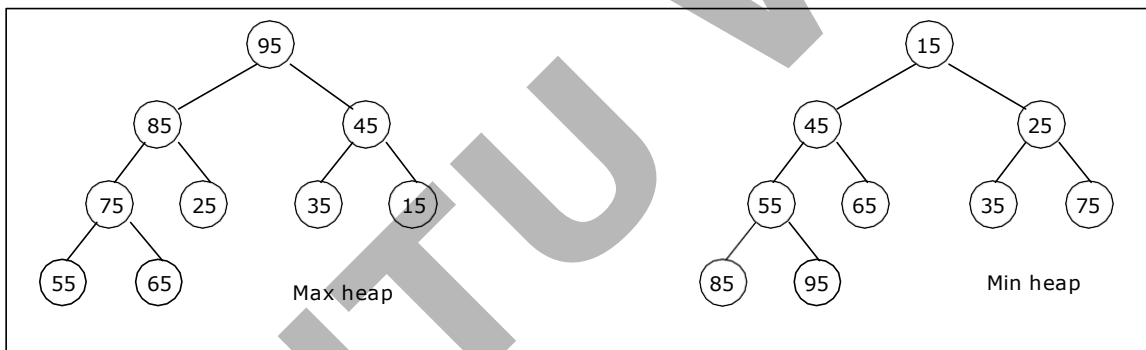


Figure 2. 1. Max. and Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children. Figure 2.1 shows the maximum and minimum heap tree.

Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location i can be found in location $2*i$.
- The right child of an element stored at location i can be found in location $2*i + 1$.
- The parent of an element stored at location i can be found at location $\text{floor}(i/2)$.

For example let us consider the following elements arranged in the form of array as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

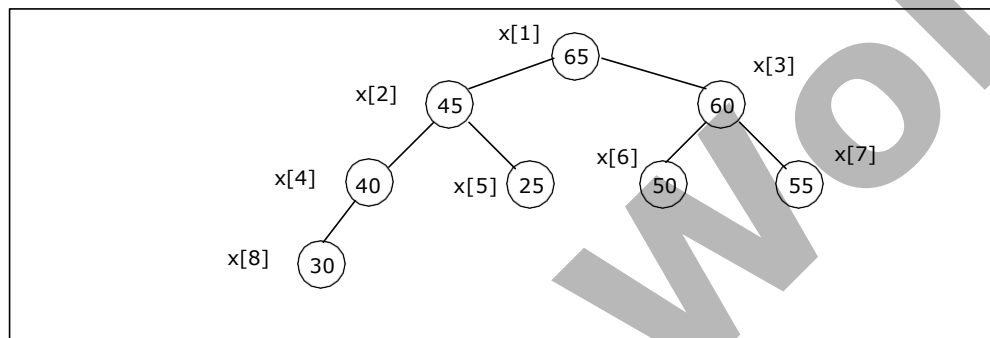


Figure 2. 2. Heap Tree

Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it

requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

```
Max_heap_insert (a, n)
{
    //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
    integer i, n;
    i = n;
    item = a[n] ;
    while ( (i > 1) and (a[⌊ i/2 ⌋] < item) ) do
    {
        a[i] = a[⌊ i/2 ⌋] ;           // move the parent down
        i = ⌊ i/2 ⌋ ;
    }
    a[i] = item ;
    return true ;
}
```

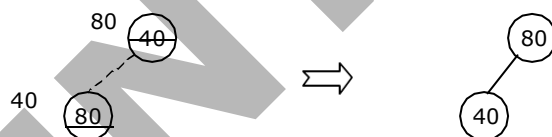
Example:

Form a heap by using the above algorithm for the given data 40, 80, 35, 90, 45, 50, 70.

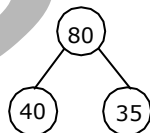
1. Insert 40:



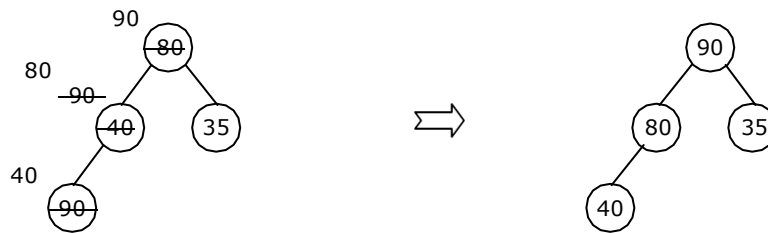
2. Insert 80:



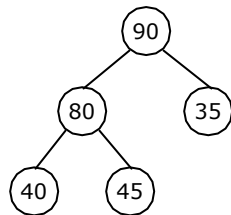
3. Insert 35:



4. Insert 90:



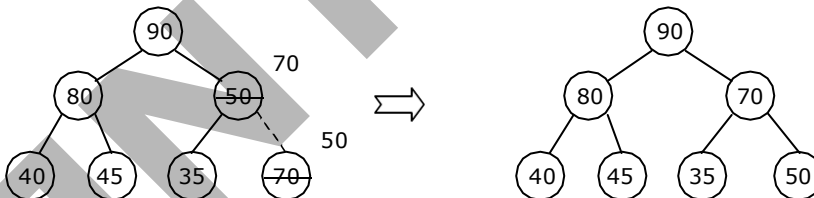
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
 - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
 - Make X as the current node.
 - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

// delete the maximum from the heap a[n] and store it in x

```
{
    if (n = 0) then
    {
        write ("heap is empty");
        return false;
    }
    x = a[1]; a[1] = a[n];
    adjust (a, 1, n-1);
    return true;
}
```

adjust (a, i, n)

// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form a single heap, $1 \leq i \leq n$. No node has an address greater than n or less than 1. //

```
{
    j = 2 * i ;
    item = a[i] ;
    while (j ≤ n) do
    {
        if ((j < n) and (a (j) < a (j + 1))) then j ← j + 1;
        // compare left and right child and let j be the larger child
        if (item ≥ a (j)) then break;
        // a position for item is found
        else a[ ⌊ j / 2 ⌋ ] = a[j] // move the larger child up a level
        j = 2 * j;
    }
    a [ ⌊ j / 2 ⌋ ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appear as the leave node, hence re-heap is completed. This is shown in figure 2.3.

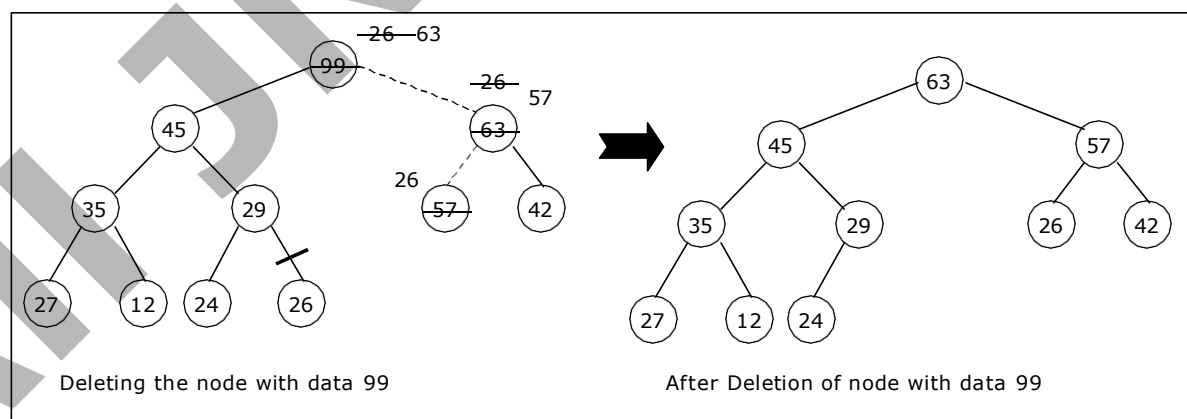


Figure 2.3.

Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap.

Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.

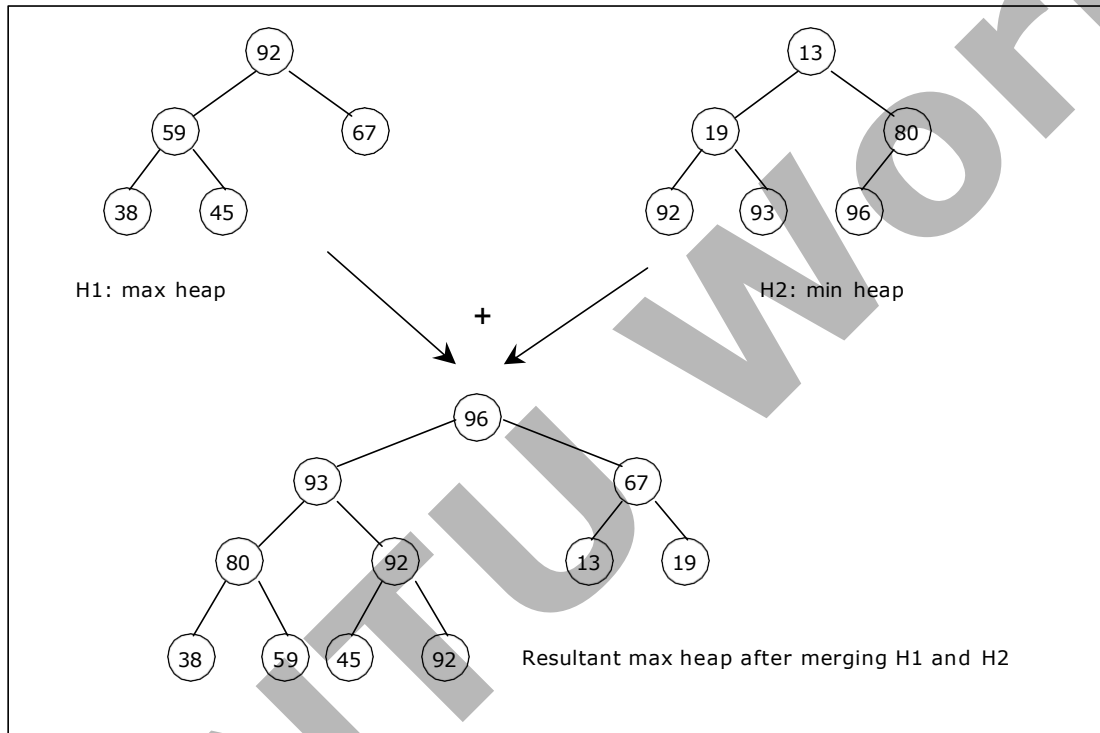


Figure 2. 4. Merging of two heaps.

Applications of heap tree:

They are two main applications of heap trees known:

1. Sorting (Heap sort) and
2. Priority queue implementation.

HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2.
 - a. Remove the top most item (the largest) and replace it with the last element in the heap.
 - b. Re-heapify the complete binary tree.
 - c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements $a[n]$. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
    heapify(a, n);
    for i = n to 2 by - 1 do
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        adjust (a, 1, i - 1);
    }
}
```

heapify (a, n)

```
//Readjust the elements in a[n] to form a heap.
{
    for i ←  $\lfloor n/2 \rfloor$  to 1 by - 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

```
// The complete binary trees with roots  $a(2*i)$  and  $a(2*i+1)$  are combined
// with  $a(i)$  to form a single heap,  $1 \leq i \leq n$ . No node has an address greater
// than n or less than 1.
{
    j = 2 * i ;
    item = a[i] ;
    while (j ≤ n) do
    {
        if ((j < n) and (a (j) < a (j + 1))) then j ← j + 1;
        // compare left and right child and let j be the larger child
        if (item ≥ a (j)) then break;
        // a position for item is found
        else a[  $\lfloor j / 2 \rfloor$  ] = a[j] // move the larger child up a level
        j = 2 * j;
    }
    a [  $\lfloor j / 2 \rfloor$  ] = item;
}
```

Time Complexity:

Each 'n' insertion operations takes $O(\log k)$, where 'k' is the number of elements in the heap at the time. Likewise, each of the 'n' remove operations also runs in time $O(\log k)$, where 'k' is the number of elements in the heap at the time. Since we always have $k \leq n$, each such operation runs in $O(\log n)$ time in the worst case.

Thus, for n elements it takes $O(n \log n)$ time, so the priority queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue.

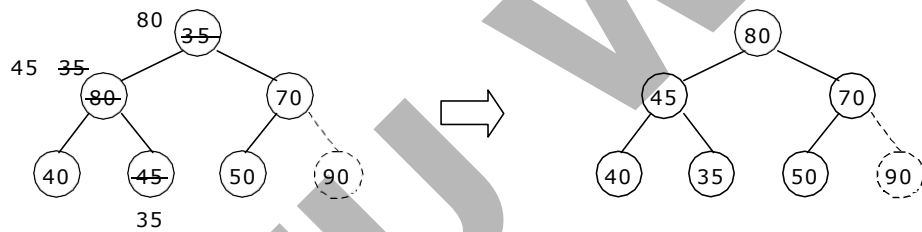
Example 1:

Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

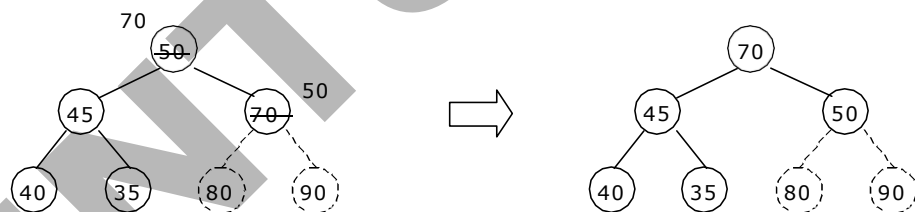
Solution:

First form a heap tree from the given set of data and then sort by repeated deletion operation:

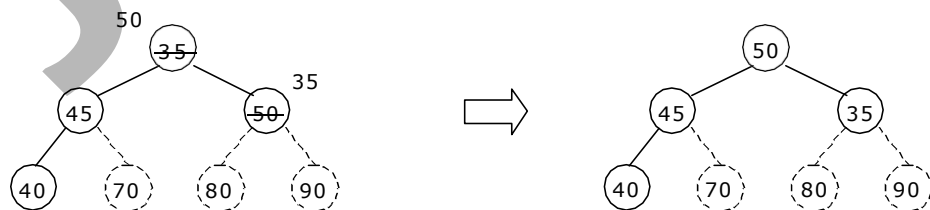
1. Exchange root 90 with the last element 35 of the array and re-heapify



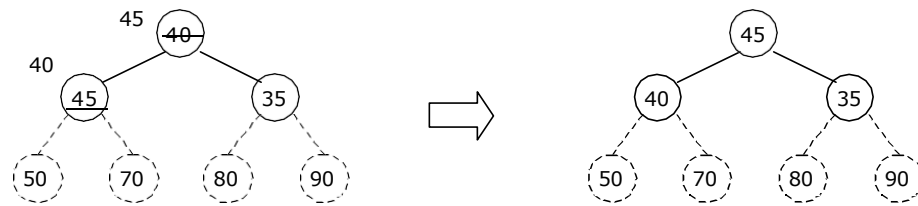
2. Exchange root 80 with the last element 50 of the array and re-heapify



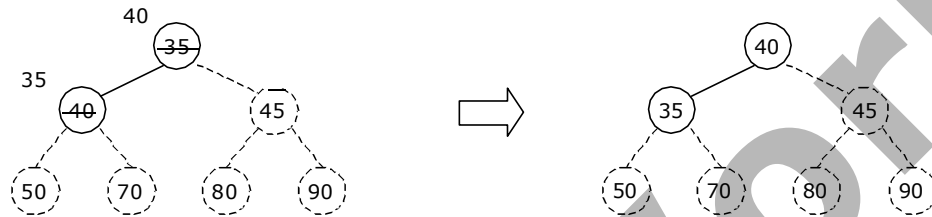
3. Exchange root 70 with the last element 35 of the array and re-heapify



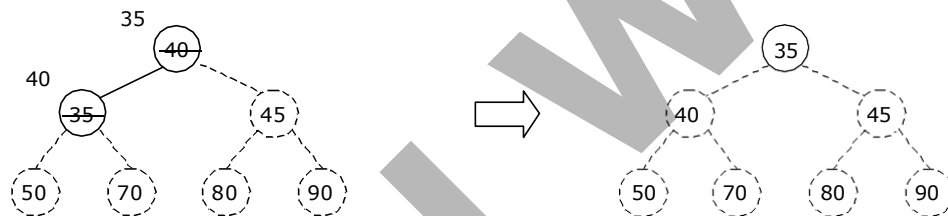
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on.

As an illustration, consider the following processes with their priorities:

Process	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇	P ₈	P ₉	P ₁₀
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Binary Search Trees:

A binary search tree has binary nodes and the following additional property. Given a node t , each node to the left is "smaller" than t , and each node to the right is "larger". This definition applies recursively down the left and right sub-trees. Figure shows a binary search tree where characters are stored in the nodes.

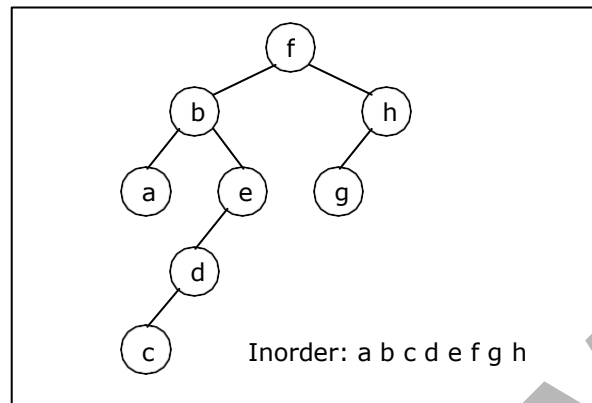


Figure 2.5. Binary Search tree

Figure 2.5 also shows what happens if you do an inorder traversal of a binary search tree: you will get a list of the node contents in sorted order. In fact, that's probably how the name inorder originated.

Binary Tree Searching:

The search operation starts from root node R , if item is less than the value in the root node R , we proceed to the left child; if item is greater than the value in the node R , we proceed to its right child. The process will be continued till the item is found or we reach to a dead end. The Figure 2.6 shows the path taken when searching for a node "c".

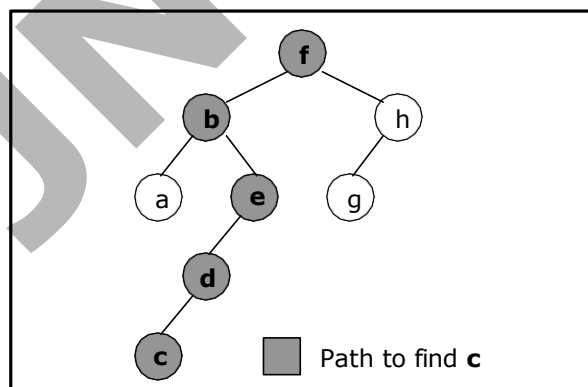


Figure 2.6. Searching a binary tree

Why use binary search trees?

Binary search trees provide an efficient way to search through an ordered collection of items. Consider the alternative of searching an ordered list. The search must proceed sequentially from one end of the list to the other. On average, $n/2$ nodes must be compared for an ordered list that contains n nodes. In the worst case, all n

nodes might need to be compared. For a large collection of items, this can get very expensive.

The inefficiency is due to the one-dimensionality of a linked list. We would like to have a way to jump into the middle of the list, in order to speed up the search process. In essence, that's what a binary search tree does. The longest path we will ever have to search is equal to the height of the tree. The efficiency of a binary search tree thus depends on the height of the tree. For a tree holding n nodes, the smallest possible height is $\log(n)$.

To obtain the smallest height, a tree must be balanced, where both the left and right sub trees have approximately the same number of nodes. Also, each node should have as many children as possible, with all levels being full except possibly the last. Figure 2.7 shows an example of a well-constructed tree.

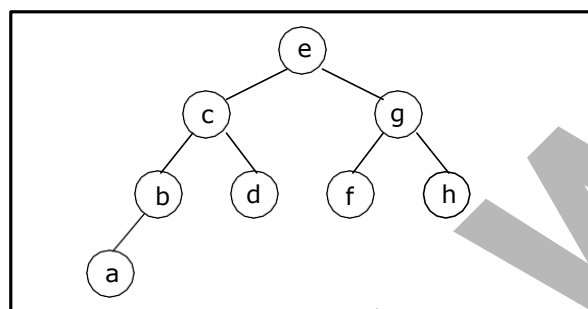


Figure 2.7. Well constructed binary search tree.

Unfortunately, trees can become so unbalanced that they're no better for searching than linked lists. Such trees are called *degenerate trees*. Figure 2.8 shows an example. For a degenerate tree, an average of $n/2$ comparisons are needed, with a worst case of n comparisons – the same as for a linked list.

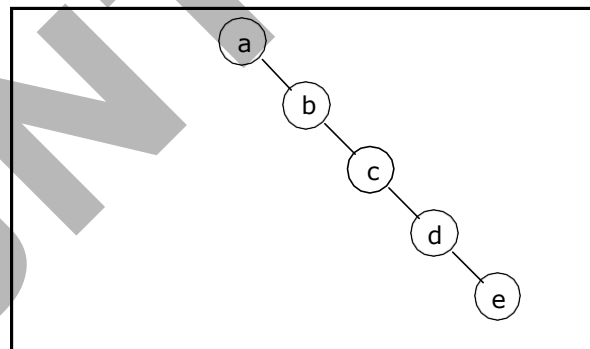


Figure 2.8. A degenerate binary search tree.

When nodes are being added and deleted in a binary search tree, it's difficult to maintain the balance of the tree. We will investigate methods of balancing trees in the next section.

Inserting Nodes into a Binary Search Tree:

When adding nodes to a binary search tree, we must be careful to maintain the binary search tree property. This can be done by first searching the tree to see whether the key we are about to add is already in the tree. If the key cannot be found, a new node is allocated and added at the same location where it would go if

the search had been successful. Figure 2.9 shows what happens when we add some nodes to a tree.

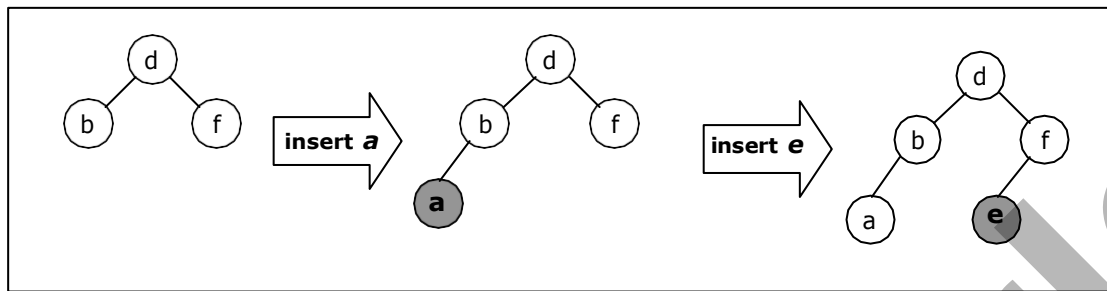


Figure 2.9. Inserting nodes into a binary search tree.

Deleting nodes from a Binary search tree:

Deletions from a binary search tree are more involved than insertions. Given a node to delete, we need to consider these tree cases:

1. The node is a leaf
2. The node has only one child.
3. The node has two children.

Case 1: It is easy to handle because the node can simply be deleted and the corresponding child pointer of its parent set to null. Figure 2.10 shows an example.

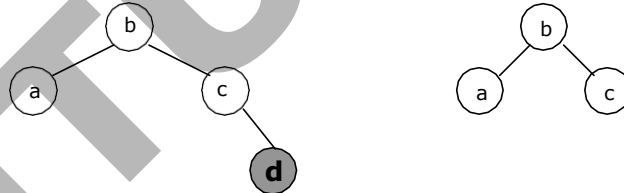


Figure 2. 10. Deleting a leaf node.

Case 2: It is almost as easy to manage. Here, the single child can be promoted up the tree to take the place of the deleted node, as shown in figure 2.11.

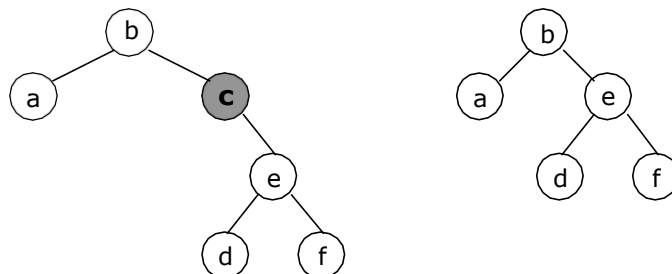


Figure 2. 11. Deleting a node that has one child.

Case 3: The node to be deleted has two children, is more difficult. We must find some node to take the place of the one deleted and still maintain the binary search tree property. There are two obvious cases:

- The inorder predecessor of the deleted node.
- The inorder successor of the deleted node.

We can detach one of these nodes from the tree and insert it where the node to be deleted.

The predecessor of a node can be found by going down to the left once, and then all the way to the right as far as possible. To find the successor, an opposite traversal is used: first to the right, and then down to the left as far as possible. Figure 2.12 shows the path taken to find both the predecessor and successor of a node.

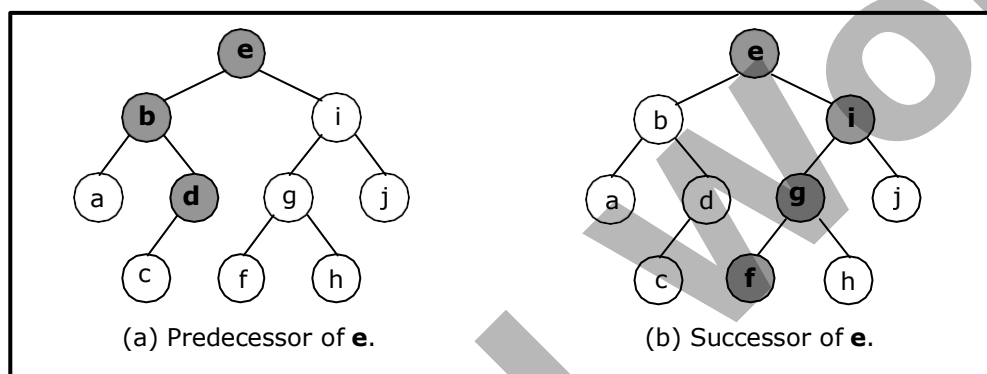


Figure 2.12. Finding predecessor and successor of nodes.

Both the predecessor and successor nodes are guaranteed to have no more than one child, and they may have none. Detaching the predecessor or successor reduces to either case 1 or 2, both of which are easy to handle. In figure 2.13 (a), we delete a node from the tree and use its predecessor as the replacement and in figure 2.13 (b), we delete a node from the tree and use its successor as the replacement.

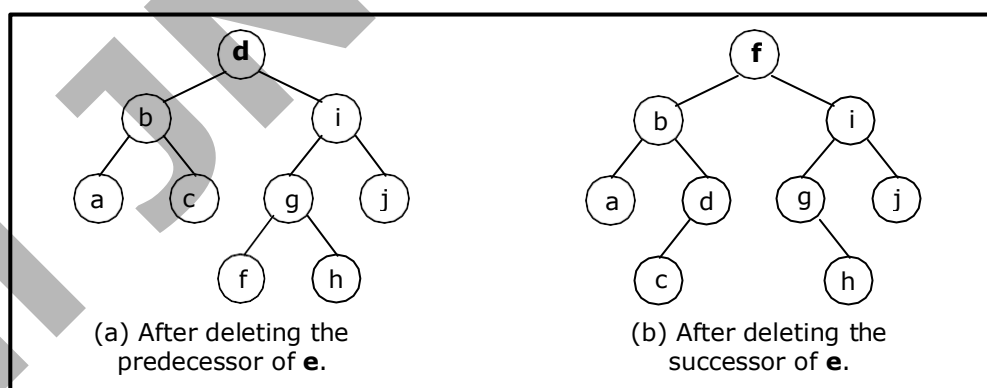


Figure 2.13. Deleting a node that has two children.

Balanced Trees:

For maximum efficiency, a binary search tree should be balanced. Every un-balanced tree is referred to as a degenerate tree, so called because their searching performance degenerates to that of linked lists. Figure 2.14 shows an example.

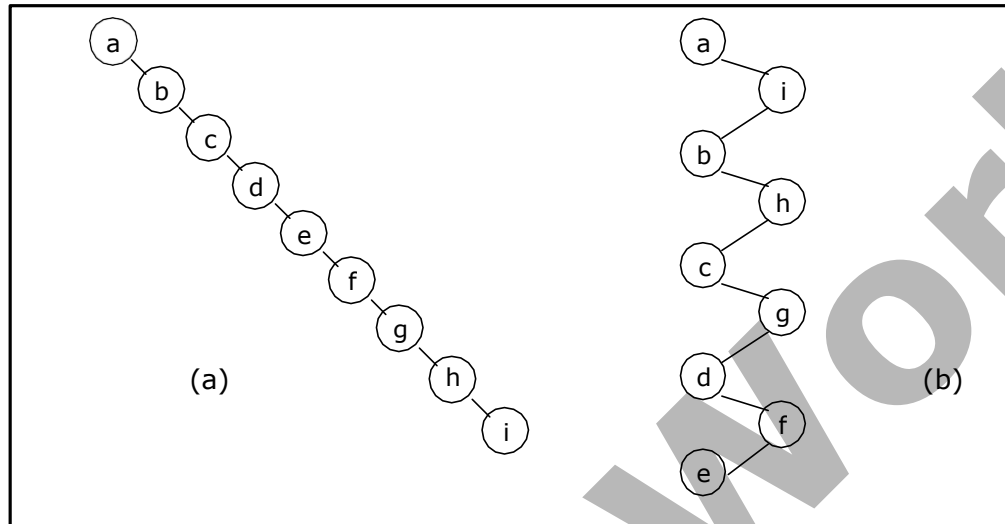


Figure 2.14. A degenerate binary search tree.

The first tree was built by inserting the keys 'a' through 'i' in sorted order into a binary search tree. The second tree was built using the insertion sequence: a-g-b-f-c-e-d. This pathological sequence is often used to test the balancing capacity of a tree. Figure 2.15 shows what the tree in 2.14-(b) above would look like if it were *balanced*.

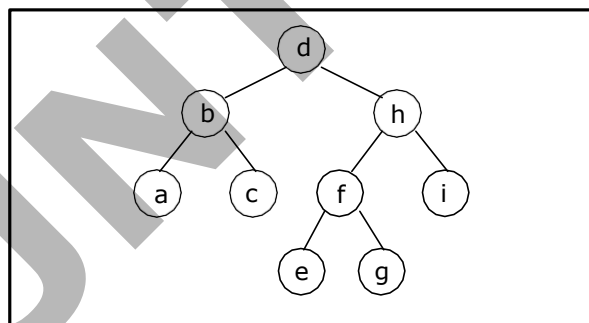


Figure 2.15 Balanced Tree

Balancing Acts:

There are two basic ways used to keep trees balanced.

- 1) Use tree rotations.
- 2) Allow nodes to have more than two children.

Tree Rotations:

Certain types of tree-restructurings, known as rotations, can aid in balancing trees. Figure 2.16 shows two types of single rotations.

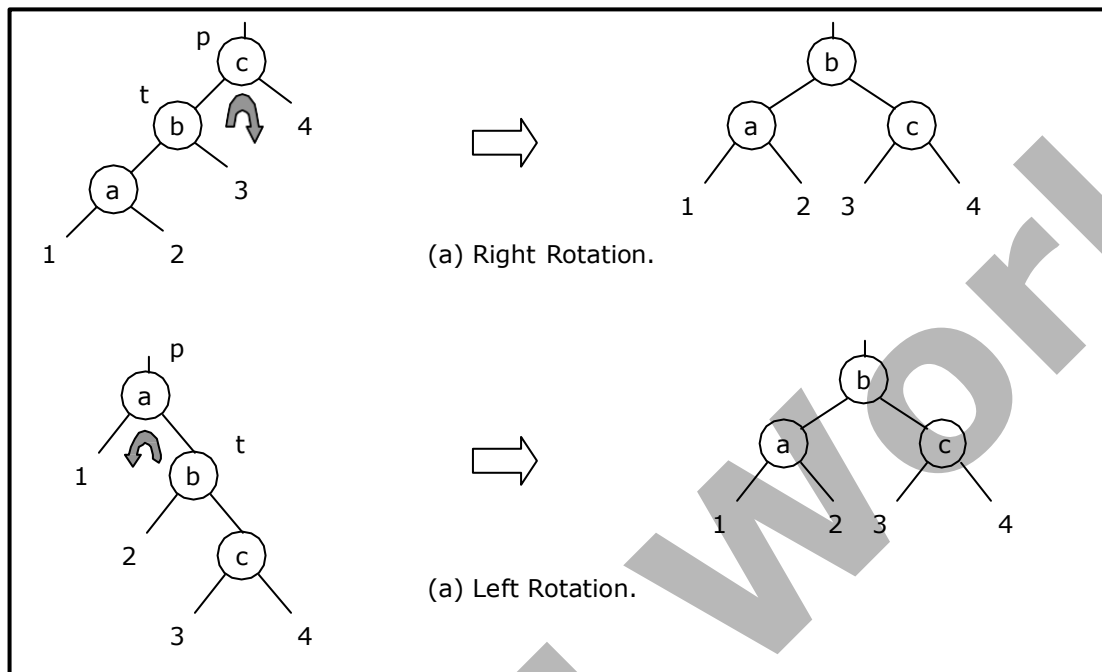


Figure 2.16. Single Rotations

Another type of rotation is known as a double rotation. Figure 2.17 shows the two symmetrical cases.

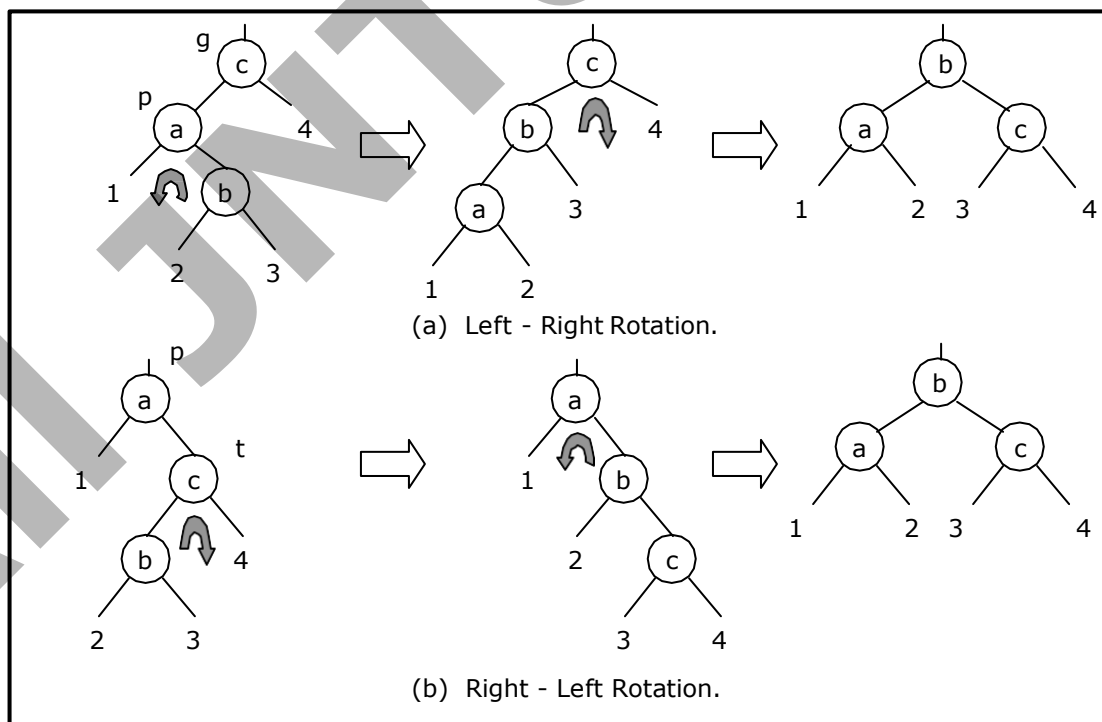


Figure 2.17. Double Rotations

Both single and double rotations have one important feature: in order traversals of the trees before and after the rotations are preserved. Thus, rotations help balance trees, but still maintain the binary search tree property.

Rotations don't guarantee a balanced tree, however for example, figure 2.18 shows a right rotations that makes a tree more un-balanced. The trick lies in determining when to do rotation and what kind of notations to do.

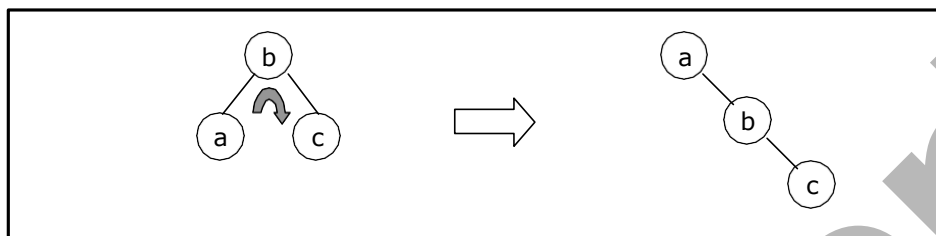


Figure 2.18. Un-balanced rotation.

The Red-black trees have certain rules to be used to determine when a how to rotate.

Dictionary:

A Dictionary is a collection of pairs of form (k, e) , where k is a key and e is the element associated with the key k (equivalently, e is the element whose key is k). No two pairs in a dictionary should have the same key.

Examples:

1. A word dictionary is a collection of elements; each element comprises a word, which is the key and the meaning of the word, pronunciation and etymologies etc. are associated with the word.
2. A telephone directory with a collection of telephone numbers and names.
3. The list of students enrolled for the data structures course, compiler and Operating System course. The list contains (CourseName, RollID) pairs.

The above last two examples are dictionaries, which permit two or more (key, element) pairs having the same key.

Operations on Dictionaries:

- Get the element associated with a specified key from the dictionary. For example, **get(k)** returns the element with the key k .
- Insert or put an element with a specified key into the dictionary. For example, **put(k, e)** puts the element e whose key is k into the dictionary.
- Delete or remove an element with a specified key. For example, **remove(k)** removes the element with key k .

Disjoint Set Operations

Disjoint Set Operations

Set:

A set is a collection of distinct elements. The Set can be represented, for examples, as $S1 = \{1, 2, 5, 10\}$.

Disjoint Sets:

The disjoint sets are those do not have any common element. For example $S1 = \{1, 7, 8, 9\}$ and $S2 = \{2, 5, 10\}$, then we can say that $S1$ and $S2$ are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

1. Union
2. Find

Disjoint set Union:

If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ consists of all the elements x such that x is in S_i or S_j .

Example:

$S1 = \{1, 7, 8, 9\}$ $S2 = \{2, 5, 10\}$

$S1 \cup S2 = \{1, 2, 5, 7, 8, 9, 10\}$

Find:

Given the element I , find the set containing i .

Example:

$S1 = \{1, 7, 8, 9\}$

$S2 = \{2, 5, 10\}$

$S3 = \{3, 4, 6\}$

Then,

$\text{Find}(4) = S3$

$\text{Find}(5) = S2$

$\text{Find}(9) = S1$

Set Representation:

The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

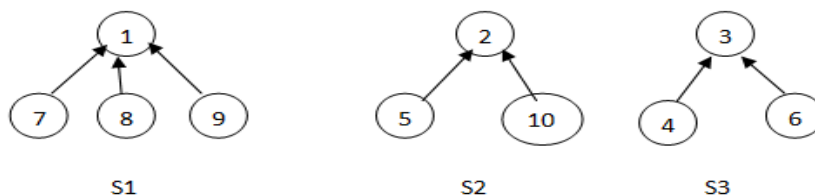
Example:

$S1 = \{1, 7, 8, 9\}$

$S2 = \{2, 5, 10\}$

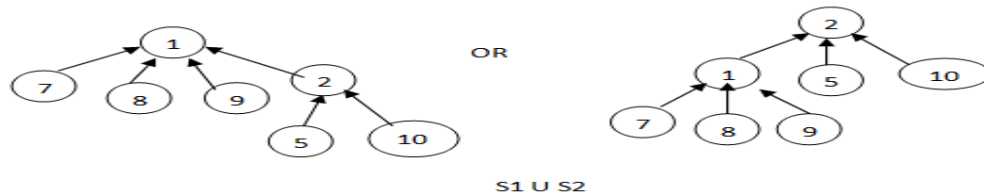
$S3 = \{3, 4, 6\}$

Then these sets can be represented as



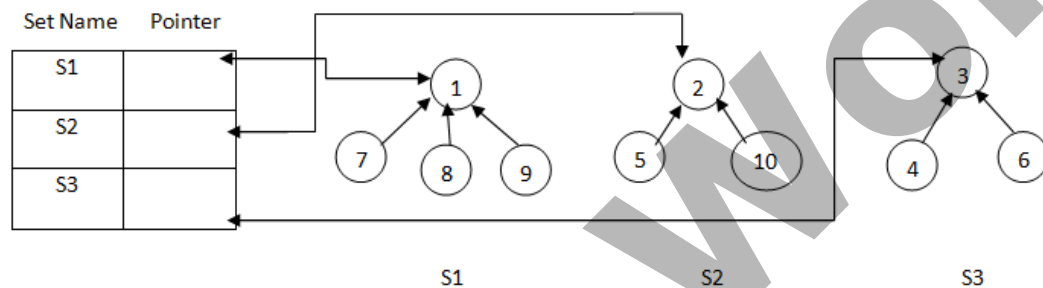
Disjoint Union:

To perform disjoint set union between two sets S_i and S_j can take any one root and make it sub-tree of the other. Consider the above example sets $S1$ and $S2$ then the union of $S1$ and $S2$ can be represented as any one of the following.



Find:

To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.

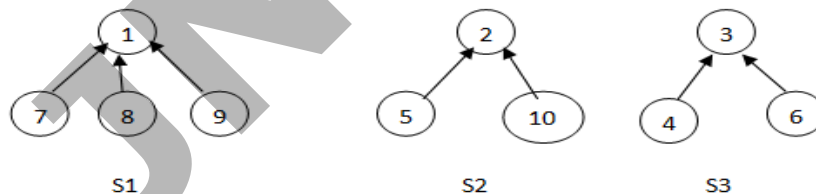


Union and Find Algorithms:

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:

For the following sets the array representation is as shown below.



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j. And make the parent of i as j i.e, make the second root as the parent of first root.

Algorithm SimpleUnion(i,j)

```
{
    P[i] := j;
}
```

Algorithm for find operation:

The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at i until it reaches a node with parent value -1.

Algorithms SimpleFind(i)

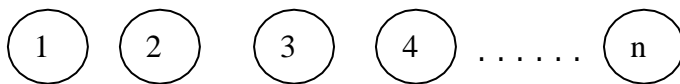
```

{
    while( P[i] ≥ 0) do i:=P[i];
    return i;
}

```

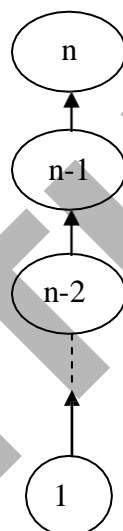
Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets



Then if we want to perform following sequence of operations Union(1,2) , Union(2,3)..... Union(n-1,n) and sequence of Find(1), Find(2)..... Find(n).

The sequence of Union operations results the degenerate tree as below.



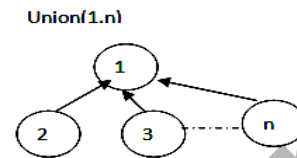
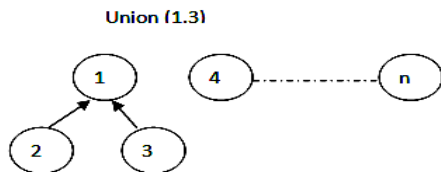
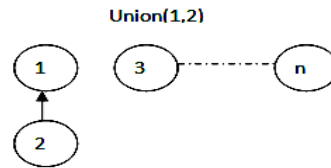
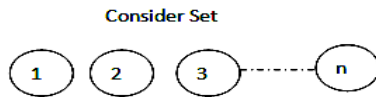
Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time $O(n)$. And for the sequence of Find operations it will take time

complexity of $O\left(\sum_{i=1}^n i\right) = O(n^2)$.

We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

Weighting rule for Union:

If the number of nodes in the tree with root i is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.



To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then $\text{count}[i]$ equals to number of nodes in tree with root i. Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

Algorithm WeightedUnion(i,j)

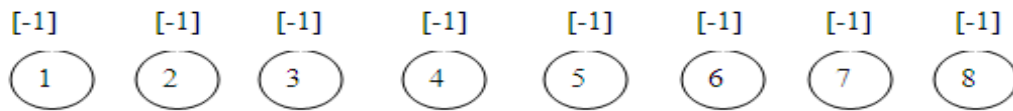
//Union sets with roots i and j , $i \neq j$ using the weighted rule

// $P[i] = -\text{count}[i]$ and $p[j] = -\text{count}[j]$

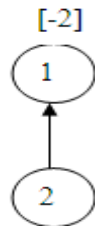
```
{
    temp := P[i] + P[j];
    if (P[i] > P[j]) then
    {
        // i has fewer nodes
        P[i] := j;
        P[j] := temp;
    }
    else
    {
        // j has fewer nodes
        P[j] := i;
        P[i] := temp;
    }
}
```

Collapsing rule for find:

If j is a node on the path from i to its root and $p[i] \neq \text{root}[i]$, then set $P[j]$ to $\text{root}[i]$. Consider the tree created by WeightedUnion() on the sequence of $1 \leq i \leq 8$. Union(1,2), Union(3,4), Union(5,6) and Union(7,8)



Union(1,2)



Union(3,4)



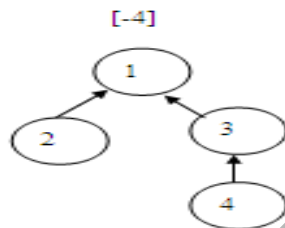
Union(5,6)



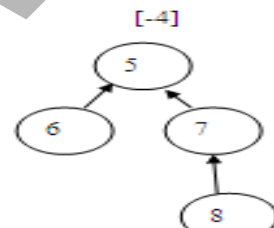
Union(7,8)



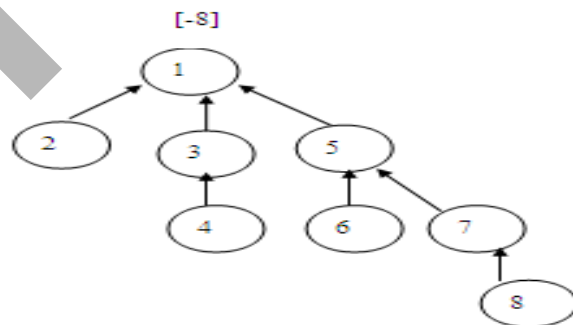
Union(1,3)



Union(5,7)



Union(1,5)



Now process the following eight find operations

Find(8), Find(8).....Find(8)

If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24 moves .

When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

Algorithm CollapsingFind(i)**// Find the root of the tree containing element i****// use the collapsing rule to collapse all nodes from i to root.**

```
{  
    r:=i;  
    while(P[r]>0) do r:=P[r]; //Find root  
    while(i≠r)  
    {  
        //reset the parent node from element i to the root  
        s:=P[i];  
        P[i]:=r;  
        i:=s;  
    }  
}
```

Recurrence Relations

Recurrence Relation for a sequence of numbers S is a formula that relates all but a finite number of terms of S to previous terms of the sequence, namely, $\{a_0, a_1, a_2, \dots, a_{n-1}\}$, for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer. Recurrence relations are also called as difference equations.

Sequences are often most easily defined with a recurrence relation; however the calculation of terms by directly applying a recurrence relation can be time consuming. The process of determining a closed form expression for the terms of a sequence from its recurrence relation is called solving the relation. Some guess and check with respect to solving recurrence relation are as follows:

- Make simplifying assumptions about inputs
- Tabulate the first few values of the recurrence
- Look for patterns, guess a solution
- Generalize the result to remove the assumptions

Examples: Factorial, Fibonacci, Quick sort, Binary search etc.

Recurrence relation is an equation, which is defined in terms of itself. There is no single technique or algorithm that can be used to solve all recurrence relations. In fact, some recurrence relations cannot be solved. Most of the recurrence relations that we encounter are linear recurrence relations with constant coefficients.

Several techniques like substitution, induction, characteristic roots and generating function are available to solve recurrence relations.

The Iterative Substitution Method:

One way to solve a divide-and-conquer recurrence equation is to use the iterative substitution method. This is a "plug-and-chug" method. In using this method, we assume that the problem size n is fairly large and we then substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, performing such a substitution with the merge sort recurrence equation yields the equation.

$$\begin{aligned}T(n) &= 2 (2 T(n/2^2) + b (n/2)) + b n \\&= 2^2 T(n/2^2) + 2 b n\end{aligned}$$

Plugging the general equation for T again yields the equation.

$$\begin{aligned}T(n) &= 2^2 (2 T(n/2^3) + b (n/2^2)) + 2 b n \\&= 2^3 T(n/2^3) + 3 b n\end{aligned}$$

The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general closed-form equation (with T only

appearing on the left-hand side). In the case of merge-sort recurrence equation, the general form is:

$$T(n) = 2^i T(n/2^i) + i b n$$

Note that the general form of this equation shifts to the base case, $T(n) = b$, where $n = 2^i$, that is, when $i = \log n$, which implies:

$$T(n) = b n + b n \log n.$$

In other words, $T(n)$ is $O(n \log n)$. In a general application of the iterative substitution technique, we hope that we can determine a general pattern for $T(n)$ and that we can also figure out when the general form of $T(n)$ shifts to the base case.

The Recursion Tree:

Another way of characterizing recurrence equations is to use the recursion tree method. Like the iterative substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach.

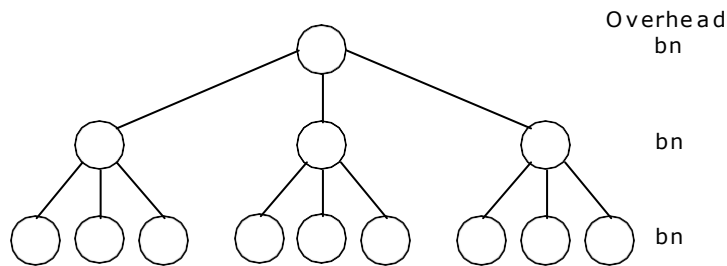
In using the recursion tree method, we draw a tree R where each node represents a different substitution of the recurrence equation. Thus, each node in R has a value of the argument n of the function $T(n)$ associated with it. In addition, we associate an overhead with each node v in R , defined as the value of the non-recursive part of the recurrence equation for v .

For divide-and-conquer recurrences, the overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v . The recurrence equation is then solved by summing the overheads associated with all the nodes of R . This is commonly done by first summing values across the levels of R and then summing up these partial sums for all the levels of R .

For example, consider the following recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 3 \\ 3T(n/3) + b n & \text{if } n \geq 3 \end{cases}$$

This is the recurrence equation that we get, for example, by modifying the merge sort algorithm so that we divide an unsorted sequence into three equal – sized sequences, recursively sort each one, and then do a three-way merge of three sorted sequences to produce a sorted version of the original sequence. In the recursion tree R for this recurrence, each internal node v has three children and has a size and an overhead associated with it, which corresponds to the time needed to merge the subproblem solutions produced by v 's children. We illustrate the tree R as follows:



The overheads of the nodes of each level, sum to bn . Thus, observing that the depth of R is $\log_3 n$, we have that $T(n)$ is $O(n \log n)$.

The Guess-and-Test Method:

Another method for solving recurrence equations is the guess-and-test technique. This technique involves first making an educated guess as to what a closed-form solution of the recurrence equation might look like and then justifying the guesses, usually by induction. For example, we can use the guess-and-test method as a kind of "binary search" for finding good upper bounds on a given recurrence equation. If the justification of our current guess fails, then it is possible that we need to use a faster-growing function, and if our current guess is justified "too easily", then it is possible that we need to use a slower-growing function. However, using this technique requires case careful, in each mathematical step we take, in trying to justify that a certain hypothesis holds with respect to our current "guess".

Example 2.10.1: Consider the following recurrence equation:

$$T(n) = 2T(n/2) + b n \log n. \text{ (assuming the base case } T(n) = b \text{ for } n < 2)$$

This looks very similar to the recurrence equation for the merge sort routine, so we might make the following as our first guess:

$$\text{First guess: } T(n) < c n \log n.$$

for some constant $c > 0$. We can certainly choose c large enough to make this true for the base case, so consider the case when $n > 2$. If we assume our first guesses an inductive hypothesis that is true for input sizes smaller than n , then we have:

$$\begin{aligned} T(n) &= 2T(n/2) + b n \log n \\ &\leq 2(c(n/2) \log(n/2)) + b n \log n \\ &\leq c n (\log n - \log 2) + b n \log n \\ &\leq c n \log n - c n + b n \log n. \end{aligned}$$

But there is no way that we can make this last line less than or equal to $c n \log n$ for $n \geq 2$. Thus, this first guess was not sufficient. Let us therefore try:

$$\text{Better guess: } T(n) \leq c n \log^2 n.$$

for some constant $c > 0$. We can again choose c large enough to make this true for the base case, so consider the case when $n \geq 2$. If we assume this guess as an

inductive hypothesis that is true for input sizes smaller than n , then we have inductive hypothesis that is true for input sizes smaller than n , then we have:

$$\begin{aligned}
 T(n) &= 2T(n/2) + b n \log n \\
 &\leq 2(c(n/2) \log^2(n/2)) + b n \log n \\
 &\leq c n (\log^2 n - 2 \log n + 1) + b n \log n \\
 &\leq c n \log^2 n - 2 c n \log n + c n + b n \log n \\
 &\leq c n \log^2 n
 \end{aligned}$$

Provided $c \geq b$. Thus, we have shown that $T(n)$ is indeed $O(n \log^2 n)$ in this case. We must take care in using this method. Just because one inductive hypothesis for $T(n)$ does not work, that does not necessarily imply that another one proportional to this one will not work.

Example 2.10.2: Consider the following recurrence equation (assuming the base case $T(n) = b$ for $n < 2$): $T(n) = 2T(n/2) + \log n$

This recurrence is the running time for the bottom-up heap construction. Which is $O(n)$. Nevertheless, if we try to prove this fact with the most straightforward inductive hypothesis, we will run into some difficulties. In particular, consider the following:

First guess: $T(n) \leq c n$.

For some constant $c > 0$. We can choose c large enough to make this true for the base case, so consider the case when $n \geq 2$. If we assume this guess as an inductive hypothesis that is true of input sizes smaller than n , then we have:

$$\begin{aligned}
 T(n) &= 2T(n/2) + \log n \\
 &\leq 2(c(n/2)) + \log n \\
 &= c n + \log n
 \end{aligned}$$

But there is no way that we can make this last line less than or equal to cn for $n > 2$. Thus, this first guess was not sufficient, even though $T(n)$ is indeed $O(n)$. Still, we can show this fact is true by using:

Better guess: $T(n) \leq c(n - \log n)$

For some constant $c > 0$. We can again choose c large enough to make this true for the base case; in fact, we can show that it is true any time $n < 8$. So consider the case when $n \geq 8$. If we assume this guess as an inductive hypothesis that is true for input sizes smaller than n , then we have:

$$\begin{aligned}
 T(n) &= 2T(n/2) + \log n \\
 &\leq 2c((n/2) - \log(n/2)) + \log n \\
 &= c n - 2c \log n + 2c + \log n \\
 &= c(n - \log n) - c \log n + 2c + \log n \\
 &\leq c(n - \log n)
 \end{aligned}$$

Provided $c \geq 3$ and $n \geq 8$. Thus, we have shown that $T(n)$ is indeed $O(n)$ in this case.

The Master Theorem Method:

Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively. There is, nevertheless, one method for solving divide-and-conquer recurrence equations that is quite general and does not require explicit use of induction to apply correctly. It is the master method. The master method is a "cook-book" method for determining the asymptotic characterization of a wide variety of recurrence equations. It is used for recurrence equations of the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ a T(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Where $d > 1$ is an integer constant, $a > 0$, $c > 0$, and $b > 1$ are real constants, and $f(n)$ is a function that is positive for $n \geq d$.

The master method for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing $f(n)$ to the special function $n^{\log_b a}$ (we will show later why this special function is so important).

The master theorem: Let $f(n)$ and $T(n)$ be defined as above.

1. If there is a small constant $\epsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \epsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.
2. If there is a constant $K \geq 0$, such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.
3. If there are small constant $\epsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ and $af(n/b) < \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.

Case 1 characterizes the situation where $f(n)$ is polynomial smaller than the special function, $n^{\log_b a}$.

Case 2 characterizes the situation when $f(n)$ is asymptotically close to the special function, and

Case 3 characterizes the situation when $f(n)$ is polynomially larger than the special function.

We illustrate the usage of the master method with a few examples (with each taking the assumption that $T(n) = c$ for $n < d$, for constants $c > 1$ and $d > 1$).

Example 2.11.1: Consider the recurrence $T(n) = 4 T(n/2) + n$

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. Thus, we are in case 1, for $f(n)$ is $O(n^{2-\epsilon})$ for $\epsilon = 1$. This means that $T(n)$ is $\Theta(n^2)$ by the master.

Example 2.11.2: Consider the recurrence $T(n) = 2 T(n/2) + n \log n$

In this case, $n^{\log_b a} = n^{\log_2 2} = n$. Thus, we are in case 2, with $k=1$, for $f(n)$ is $\Theta(n \log n)$. This means that $T(n)$ is $\Theta(n \log^2 n)$ by the master method.

Example 2.11.3: consider the recurrence $T(n) = T(n/3) + n$

In this case $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for $f(n)$ is $\Omega(n^{\epsilon})$, for $\epsilon = 1$, and $af(n/b) = n/3 = (1/3) f(n)$. This means that $T(n)$ is $\Theta(n)$ by the master method.

Example 2.11.4: Consider the recurrence $T(n) = 9 T(n/3) + n^{2.5}$

In this case, $n^{\log_b a} = n^{\log_3 9} = n^2$. Thus, we are in Case 3, since $f(n)$ is $\Omega(n^{2+\epsilon})$ (for $\epsilon=1/2$) and $af(n/b) = 9 (n/3)^{2.5} = (1/3)^{1/2} f(n)$. This means that $T(n)$ is $\Theta(n^{2.5})$ by the master method.

Example 2.11.5: Finally, consider the recurrence $T(n) = 2T(n^{1/2}) + \log n$

Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable $k = \log n$, which lets us write:

$$T(n) = T(2^k) = 2 T(2^{k/2}) + k$$

Substituting into this the equation $S(k) = T(2^k)$, we get that

$$S(k) = 2 S(k/2) + k$$

Now, this recurrence equation allows us to use master method, which specifies that $S(k)$ is $O(k \log k)$. Substituting back for $T(n)$ implies $T(n)$ is $O(\log n \log \log n)$.

CLOSED FORM EXPRESSION

There exists a closed form expression for many summations

Sum of first n natural numbers (Arithmetic progression)

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

Sum of squares of first n natural numbers

$$\sum_{i=1}^n i^2 = 1 + 4 + 9 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Sum of cubes

$$\sum_{i=1}^n i^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4}$$

Geometric progression

$$\sum_{i=0}^n 2^i = 2^0 + 2^1 + 2^2 + \dots + 2^n = 2^{n+1} - 1$$

$$\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}$$

$$\sum_{i=1}^n r^i = \frac{r^n - 1}{r - 1}$$

SOLVING RECURRENCE RELATIONS

Example 2.13.1. Solve the following recurrence relation:

$$T(n) = \begin{cases} 2 & , n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + 7 & , n > 1 \end{cases}$$

Solution: We first start by labeling the main part of the relation as Step 1:

Step 1: Assume $n > 1$ then,

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 7$$

Step 2: Figure out what $T\left(\frac{n}{2}\right)$ is; everywhere you see n , replace it with $\frac{n}{2}$.

$$T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{2^2}\right) + 7$$

Now substitute this back into the last $T(n)$ definition (last line from step 1):

$$\begin{aligned} T(n) &= 2 \cdot \left[2 \cdot T\left(\frac{n}{2^2}\right) + 7 \right] + 7 \\ &= 2^2 \cdot T\left(\frac{n}{2^2}\right) + 3 \cdot 7 \end{aligned}$$

Step 3: let's expand the recursive call, this time, it's $T\left(\frac{n}{2^2}\right)$:

$$T\left(\frac{n}{2^2}\right) = 2 \cdot T\left(\frac{n}{2^3}\right) + 7$$

Now substitute this back into the last $T(n)$ definition (last line from step 2):

$$T(n) = 2^2 \cdot \left[2 \cdot T\left(\frac{n}{2^3}\right) + 7 \right] + 7$$

$$2^3 \cdot T\left(\frac{n}{2^3}\right) + 7 \cdot 7$$

From this, first, we notice that the power of 2 will always match i , current. Second, we notice that the multiples of 7 match the relation $2^i - 1 : 1, 3, 7, 15$. So, we can write a general solution describing the state of $T(n)$.

Step 4: Do the i^{th} substitution.

$$T(n) = 2^i \cdot T\left(\frac{n}{2^i}\right) + (2^i - 1) \cdot 7$$

However, how many times could we take these "steps"? Indefinitely? No... it would stop when n has been cut in half so many times that it is effectively 1. Then, the original definition of the recurrence relation would give us the terminating condition $T(1) = 1$ and us restrict size $n = 2^i$

$$\text{When, } 1 = \frac{n}{2^i}$$

$$\Rightarrow 2^i = n$$

$$\Rightarrow \log_2 2^i = \log_2 n$$

$$\Rightarrow i \cdot \log_2 2 = \log_2 n$$

$$\Rightarrow i = \log_2 n$$

Now we can substitute this "last value for i " back into our general Step 4 equation:

$$\begin{aligned} T(n) &= 2^i \cdot T\left(\frac{n}{2^i}\right) + (2^i - 1) \cdot 7 \\ &= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + (2^{\log_2 n} - 1) \cdot 7 \\ &= n \cdot T(1) + (n - 1) \cdot 7 \\ &= n \cdot 1 + (n - 1) \cdot 7 \\ &= 9 \cdot n - 7 \end{aligned}$$

This implies that $T(n)$ is **$O(n)$** .

Example 2.13.2. Imagine that you have a recursive program whose run time is described by the following recurrence relation:

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + 4 & , n > 1 \end{cases}$$

Solve the relation with iterated substitution and use your solution to determine a tight big-oh bound.

Solution:

Step 1: Assume $n > 1$ then,

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

Step 2: figure out what $T\left(\frac{n}{2}\right)$ is:

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

Now substitute this back into the last $T(n)$ definition (last line from step 1):

$$\begin{aligned} T(n) &= 2 \cdot \left[2 \cdot T\left(\frac{n}{2^2}\right) + 4 \cdot \frac{n}{2} \right] + 4 \cdot n \\ &= 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2 \cdot 4 \cdot n \end{aligned}$$

Step 3: figure out what $T\left(\frac{n}{2^2}\right)$ is:

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

Now substitute this back into the last $T(n)$ definition (last time from step 2):

$$\begin{aligned} T(n) &= 2^2 \cdot \left[2 \cdot T\left(\frac{n}{2^3}\right) + 4 \cdot \frac{n}{2^2} \right] + 2 \cdot 4 \cdot n \\ &= 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3 \cdot 4 \cdot n \end{aligned}$$

Step 4: Do the i^{th} substitution.

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

The parameter to the recursive call in the last line will equal 1 when $i = \log_2 n$ (use the same analysis as the previous example). In which case $T(1) = 1$ by the original definition of the recurrence relation.

Now we can substitute this back into our general Step 4 equation:

$$\begin{aligned}
T(n) &= 2^i \cdot T\left(\frac{n}{2^i}\right) + i \cdot 4 \cdot n \\
&= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \cdot 4 \cdot n \\
&= n \cdot T(1) + 4 \cdot n \cdot \log_2 n \\
&= n + 4 \cdot n \cdot \log_2 n
\end{aligned}$$

This implies that $T(n)$ is **$O(n \log n)$** .

Example 2.13.3. Write a recurrence relation and solve the recurrence relation for the following fragment of program:

Write $T(n)$ in terms of T for fractions of n , for example, $T(n/k)$ or $T(n - k)$.

```

int findmax (int a[ ], int start, int end)
{
    int mid, max1, max2;

    if (start == end)                // easy case (2 steps)
        return (a [start]);

    mid = (start + end) / 2;          // pre-processing (3)
    max1 = findmax (a, start, mid);   // recursive call:  $T(n/2) + 1$ 
    max2 = findmax (a, mid+1, end);   // recursive call:  $T(n/2) + 1$ 
    if (max1 >= max2)                 // post-processing (2)
        return (max1);
    else
        return (max2);
}

```

Solution:

The Recurrence Relation is:

$$T(n) = \begin{cases} 2, & \text{if } n = 1 \\ 2 T(n/2) + 8, & \text{if } n > 1 \end{cases}$$

Solving the Recurrence Relation by **iteration method**:

Assume $n > 1$ then $T(n) = 2 T(n/2) + 8$

Step 1: Substituting $n = n/2, n/4, \dots$ for n in the relation:

Since, $T(n/2) = 2 T((n/2)/2) + 8$, and $T(n/4) = 2 T((n/4)/2) + 8$,

$$T(n) = 2 [T(n/2)] + 8 \quad (1)$$

$$T(n) = 2 [2 T(n/4) + 8] + 8 \quad (2)$$

$$\begin{aligned} T(n) &= 2 [2 [2 T(n/8) + 8] + 8] + 8 \quad (3) \\ &= 2 \times 2 \times 2 T(n/2 \times 2 \times 2) + 2 \times 2 \times 8 + 2 \times 8 + 8, \end{aligned}$$

Step 2: Do the i^{th} substitution:

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + 8(2^{i-1} + \dots + 2^2 + 2 + 1) \\ &= 2^i T(n/2^i) + 8 \left(\sum_{k=0}^{i-1} 2^k \right) \\ &= 2^i T(n/2^i) + 8(2^i - 1) \text{ (the formula for geometric series)} \end{aligned}$$

Step 3: Define i in terms of n :

$$\text{If } n = 2^i, \text{ then } i = \log n \text{ so, } T(n/2^i) = T(1) = 2$$

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + 8(2^i - 1) \\ &= n T(1) + 8(n-1) \\ &= 2n + 8n - 8 = 10n - 8 \end{aligned}$$

Step 4: Representing in big-O function:

$$T(n) = 10n - 8 \text{ is } \mathbf{O(n)}$$

Example 2.13.4. If K is a non negative constant, then prove that the recurrence

$$T(n) = \begin{cases} k, & n = 1 \\ 3.T\left(\frac{n}{2}\right) + k \cdot n, & n > 1 \end{cases}$$

has the following solution (for n a power of 2)

$$T(n) = 3k \cdot n^{\log_2 3} - 2k \cdot n$$

Solution:

$$\text{Assuming } n > 1, \text{ we have } T(n) = 3T\left(\frac{n}{2}\right) + K \cdot n \quad (1)$$

Substituting $n = \frac{n}{2}$ for n in equation (1), we get

$$T\left(\frac{n}{2}\right) = 3T\left(\frac{n}{4}\right) + k \frac{n}{2} \quad (2)$$

Substituting equation (2) for $T\left(\frac{n}{2}\right)$ in equation (1)

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

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

Substituting $n = \frac{n}{4}$ for n equation (1)

$$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{8}\right) + k \frac{n}{4} \quad (4)$$

Substituting equation (4) for $T = \frac{n}{4}$ in equation (3)

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

$$= 3^3 T\left(\frac{n}{8}\right) + \frac{9}{4} k \cdot n + \frac{3}{2} k \cdot n + k \cdot n \quad (5)$$

Continuing in this manner and substituting $n = 2^i$, we obtain

$$T(n) = 3^i T\left(\frac{n}{2^i}\right) + \frac{3^{i-1}}{2^{i-1}} k \cdot n + \dots + \frac{9kn}{4} + \frac{3}{2} k \cdot n + k \cdot n$$

$$T(n) = 3^i T\left(\frac{n}{2^i}\right) + \left[\frac{3^i - 1}{3 - 1} \right] k \cdot n \quad \text{as } \sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1}$$

$$= 3^i T\left(\frac{n}{2^i}\right) + 2 \cdot \left[\frac{3^i}{2} \right] k \cdot n - 2kn \quad (6)$$

As $n = 2^i$, then $i = \log_2 n$ and by definition as $T(1) = k$

$$T(n) = 3^i k + 2 \cdot \frac{3^i}{n} \cdot kn - 2kn$$

$$= 3^i (3k) - 2kn$$

$$= 3k \cdot 3^{\log_2 n} - 2kn$$

$$= 3kn^{\log_2 3} - 2kn$$

Example 2.13.5. Towers of Hanoi

The Towers of Hanoi is a game played with a set of donut shaped disks stacked on one of three posts. The disks are graduated in size with the largest on the bottom. The objective of the game is to transfer all the disks from post B to post A moving one disk at a time without placing a larger disk on top of a smaller one. What is the minimum number of moves required when there are n disks?

In order to visualize the most efficient procedure for winning the game consider the following:

1. Move the first $n-1$ disks, as per the rules in the most efficient manner possible, from post B to post C.
2. Move the remaining, largest disk from post B to post A.
3. Move the $n - 1$ disks, as per the rules in the most efficient manner possible, from post C to post A.

Let m_n be the number of moves required to transfer n disks as per the rules in the most efficient manner possible from one post to another. Step 1 requires moving $n - 1$ disks or m_{n-1} moves. Step 2 requires 1 move. Step 3 requires m_{n-1} moves again. We have then,

$$M_n = m_{n-1} + 1 + m_{n-1}, \text{ for } n > 2 = 2m_{n-1} + 1$$

Because only one move is required to win the game with only one disk, the initial condition for this sequence is $m_1 = 1$. Now we can determine the number of moves required to win the game, in the most efficient manner possible, for any number of disks.

$$m_1 = 1$$

$$m_2 = 2(1) + 1 = 3$$

$$m_3 = 2(3) + 1 = 7$$

$$m_4 = 2(7) + 1 = 15$$

$$m_5 = 2(15) + 1 = 31$$

$$m_6 = 2(31) + 1 = 63$$

$$m_7 = 2(63) + 1 = 127$$

Unfortunately, the recursive method of determining the number of moves required is exhausting if we wanted to solve say a game with 69 disks. We have to calculate each previous term before we can determine the next.

Lets look for a pattern that may be useful in determining an explicit formula for m_n . In looking for patterns it is often useful to forego simplification of terms. $m_n = 2m_{n-1} + 1, m_1 = 1$

$$m_1 = 1$$

$$m_2 = 2(1) + 1 = 2+1$$

$$m_3 = 2(2+1) + 1 = 2^2+2+1$$

$$m_4 = 2(2^2+2+1) + 1 = 2^3+2^2+2+1$$

$$m_5 = 2(2^3+2^2+2+1) + 1 = 2^4+2^3+2^2+2+1$$

$$m_6 = 2(2^4 + 2^3 + 2^2 + 2 + 1) + 1 = 2^5 + 2^4 + 2^3 + 2^2 + 2 + 1$$

$$m_7 = 2(2^5 + 2^4 + 2^3 + 2^2 + 2 + 1) + 1 = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2 + 1$$

So we guess an explicit formula:

$$M_k = 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2 + 1$$

By sum of a Geometric Sequence, for any real number r except 1, and any integer $n \geq 0$.

$$\sum_{i=0}^n r^i = \frac{r^{n+1} - 1}{r - 1}$$

our formula is then

$$m_k = \sum_{k=0}^{n-1} 2^k = \frac{2^n - 1}{2 - 1} = 2^n - 1$$

This is nothing but **$O(2^n)$**

Thus providing an explicit formula for the number of steps required to win the Towers of Hanoi Game.

Example 2.13.6 Solve the recurrence relation $T(n) = 5T(n/5) + n/\log n$

Using iteration method,

$$\begin{aligned} T(n) &= 5T(n/5) + n/\log(n/5) + n/\log n \\ &= 125T(n/125) + n/\log(n/25) + n/\log(n/5) + n/\log n \\ &= 5^k T(n/5^k) + n/\log(n/5^{k-1}) + n/\log(n/5^{k-2}) + \dots + n/\log n \end{aligned}$$

When $n = 5^k$

$$\begin{aligned} &= n * T(1) + n/\log 5 + n/\log 25 + n/\log 125 + \dots + n/\log n \\ &= c n + n (1/\log 5 + 1/\log 25 + 1/\log 125 + \dots + 1/\log n) \\ &= c n + n \log_5 2 (1/1 + 1/2 + 1/3 + \dots + 1/k) \\ &= c n + \log_5 2 n \log(k) \\ &= c n + \log_5 2 n \log(\log n) \\ &= \theta(n \log(\log n)) \end{aligned}$$

Example 2.13.7. Solve the recurrence relation $T(n) = 3T(n/3 + 5) + n/2$

Use the substitution method, guess that the solution is $T(n) = O(n \log n)$

We need to prove $T(n) \leq c n \log n$ for some c and all $n > n_0$.

Substitute $c(n/3 + 5) \log(n/3 + 5)$ for $T(n/3 + 5)$ in the recurrence

$$T(n) \leq 3 * c(n/3 + 5) * (\log(n/3 + 5)) + n/2$$

If we take n_0 as 15, then for all $n > n_0$, $n/3 + 5 \leq 2n/3$, so

$$\begin{aligned} T(n) &\leq (cn + 15c) * (\log n/3 + \log 2) + n/2 \\ &\leq cn \log n - cn \log 3 + 15c \log n - 15c \log 3 + cn + 15c + n/2 \\ &\leq cn \log n - (cn(\log 3 - 1) + n/2 - 15c \log n - 15c(\log 3 - 1)) \\ &\leq cn \log n, \text{ by choosing an appropriately large constant } c \end{aligned}$$

Which implies $T(n) = O(n \log n)$

Similarly, by guessing $T(n) \geq c_1 n \log n$ for some c_1 and all $n > n_0$ and substituting in the recurrence, we get.

$$\begin{aligned} T(n) &\geq c_1 n \log n - c_1 n \log 3 + 15c_1 \log n - 15c_1 \log 3 + c_1 n + 15c_1 + n/2 \\ &\geq c_1 n \log n + n/2 + c_1 n(1 - \log 3) + 15c_1 + 15c_1(\log n - \log 3) \\ &\geq c_1 n \log n + n(0.5 + c_1(1 - \log 3)) + 15c_1 + 15c_1(\log n - \log 3) \end{aligned}$$

by choosing an appropriately small constant c_1 (say $c_1 = 0.01$) and for all $n > 3$

$$T(n) \geq c_1 n \log n$$

Which implies $T(n) = \Omega(n \log n)$

Thus, $T(n) = \theta(n \log n)$

Example 2.13.8. Solve the recurrence relation $T(n) = 2T(n/2) + n/\log n$.

Using iteration method,

$$\begin{aligned} T(n) &= 4T(n/4) + n/\log(n/2) + n/\log n \\ &= 8T(n/8) + n/\log(n/4) + n/\log(n/2) + n/\log n \\ &= 2^k T(n/2^k) + n/\log(n/2^{k-1}) + n/\log(n/2^{k-2}) + \dots + n/\log n \end{aligned}$$

When $n = 2^k$

$$\begin{aligned} &= n * T(1) + n/\log 2 + n/\log 4 + n/\log 8 + \dots + n/\log n \\ &= cn + n(1/\log 2 + 1/\log 4 + 1/\log 8 + \dots + 1/\log n) \\ &= cn + n(1/1 + 1/2 + 1/3 + \dots + 1/k) \\ &= cn + n \log(k) \\ &= cn + n \log(\log n) \end{aligned}$$

$$= \theta(n \log(\log n))$$

Example 2.13.9: Solve the recurrence relation:

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

Use the substitution method, guess that the solution is $T(n) = O(n \log n)$.

We need to prove $T(n) \leq c n \log n$ for some c and all $n > n_0$.

Substituting the guess in the given recurrence, we get

$$\begin{aligned} T(n) &\leq c(n/2) \log(n/2) + c(n/4) \log(n/4) + c(n/8) \log(n/8) + n \\ &\leq c(n/2)(\log n - 1) + c(n/4)(\log n - 2) + c(n/8)(\log n - 3) + n \\ &\leq c n \log n (1/2 + 1/4 + 1/8) - c n/2 - c n/2 - 3 c n/8 + n \\ &\leq c n \log n (7/8) - (11 c n - 8 n) / 8 \\ &\leq c n \log n (7/8) \quad (\text{if } c = 1) \\ &\leq c n \log n \end{aligned}$$

From the recurrence equation, it can be inferred that $T(n) \geq n$. So, $T(n) = \Omega(n)$

Example 2.13.10: Solve the recurrence relation: $T(n) = 28 T(n/3) + cn^3$

Let us consider : $n^{\log_a b} \Rightarrow \frac{f(n)}{n^{\log_a b}} = \frac{cn^3}{n^{\log_3 28}}$

According to the law of indices: $\frac{a^x}{a^y}$, we can write a^{x-y}

$$\frac{cn^3}{n^{\log_3 28}} = c n^{3 - \log_3 28} = c n^r \quad \text{where } r = 3 - \log_3 28 < 0$$

It can be written as: $h(n) = O(n^r)$ where $r < 0$

$f(n)$ for these from the table can be taken $O(1)$

$$\begin{aligned} T(n) &= n \log_3^{28} [T(1) + h(n)] \\ &= n \log_3^{28} [T(1) + O(1)] = \theta\left(n \log_3^{28}\right) \end{aligned}$$

Example 2.13.12: Solve the recurrence relation $T(n) = T(\sqrt{n}) + c, \quad n > 4$

$$\begin{aligned}
T(n) &= T(n^{1/2}) + C \\
T(n)^{1/2} &= T(n^{1/2})^{1/2} + C \\
T(n^{1/2}) &= T(n^{1/4}) + C + C \\
T(n) &= T(n^{1/4}) + 2C \\
T(n) &= T(n^{1/2}) + 3C \\
&= T(n^{1/2^i}) + iC \\
&= T(n^{1/n}) + C \log_2 n \\
T(\sqrt[n]{n}) + C \log_2 n &= \theta(\log n)
\end{aligned}$$

Example 2.13.13: Solve the recurrence relation

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \log n & n > 4 \end{cases}$$

$$\begin{aligned}
T(n) &= 2 \left[2T(n^{1/2})^{1/2} + \log \sqrt{n} \right] + \log n \\
&= 4T(n^{1/4}) + 2 \log n^{1/2} + \log n \\
T(n^{1/4}) &= 2T(n^{1/2})^{1/4} + \log n^{1/4} \\
T(n) &= 4 \left[2T(n^{1/2})^{1/4} + \log n^{1/4} \right] + 2 \log n^{1/4} + \log n \\
&= 8T(n^{1/8}) + 4 \log n^{1/4} + 2 \log n^{1/2} + \log n \\
T(n^{1/8}) &= 2T(n^{1/2})^{1/8} + \log n^{1/8} \\
T(n) &= 8 \left[2T(n^{1/16}) + \log n^{1/8} \right] + 4 \log n^{1/4} + 2 \log n^{1/2} + \log n \\
&= 16T(n^{1/16}) + 8 \log n^{1/8} + 4 \log n^{1/4} + 2 \log n^{1/2} + \log n \\
&= 2^i T(n^{1/2^i}) + 2^{i-1} \log n^{1/2^{i-1}} + 2^{i-2} \log n^{1/2^{i-2}} + 2^{i-3} \log n^{1/2^{i-3}} + 2^{i-4} \log n^{1/2^{i-4}} \\
&= 2^i T(n^{1/2^i}) + \sum_{k=1}^n 2^{i-k} \log n^{1/2^{i-k}} \\
&= nT(n^{1/n}) + \sum \frac{n}{2^k} \log n^{k/n} \\
&= \theta(\log n)
\end{aligned}$$

All JNTU World

Chapter 3

Divide and Conquer

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from $O(n^2)$ to $O(n \log n)$ to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide-and-conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

DANDC (P)

```
{
    if SMALL (P) then return S (p);
    else
    {
        divide p into smaller instances  $p_1, p_2, \dots, p_k, k \geq 1$ ;
        apply DANDC to each of these sub problems;
        return (COMBINE (DANDC ( $p_1$ ) , DANDC ( $p_2$ ),..., DANDC ( $p_k$ )));
    }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p_1, p_2, \dots, p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, $T(n)$ is the time for DANDC on 'n' inputs

$g(n)$ is the time to complete the answer directly for small inputs and

$f(n)$ is the time for Divide and Combine

Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that $a[j] = x$ (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key $a[mid]$, and compare 'x' with $a[mid]$. If $x = a[mid]$ then the desired record has been found. If $x < a[mid]$ then 'x' must be in that portion of the file that precedes $a[mid]$, if there at all. Similarly, if $a[mid] > x$, then further search is only necessary in that past of the file which follows $a[mid]$. If we use recursive procedure of finding the middle key $a[mid]$ of the un-searched portion of a file, then every un-successful comparison of 'x' with $a[mid]$ will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and $a[mid]$, and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm Algorithm

```
BINSRCH (a, n, x)
//   array a(1 : n) of elements in increasing order,  $n \geq 0$ ,
//   determine whether 'x' is present, and if so, set j such that  $x = a(j)$ 
//   else return j

{
    low := 1 ; high := n ;
    while (low  $\leq$  high) do
    {
        mid := |(low + high)/2|
        if (x < a [mid]) then high:=mid - 1;
        else if (x > a [mid]) then low:= mid + 1
        else return mid;
    }
    return 0;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for $x = 101$

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
found		

Number of comparisons = 4

2. Searching for $x = 82$

low	high	mid
1	9	5
6	9	7
8	9	8
found		

Number of comparisons = 3

3. Searching for $x = 42$

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

Number of comparisons = 4

4. Searching for $x = -14$

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	3	2	3	4	1	3	2	3	4

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding $25/9$ or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x .

If $x < a[1]$, $a[1] < x < a[2]$, $a[2] < x < a[3]$, $a[5] < x < a[6]$, $a[6] < x < a[7]$ or $a[7] < x < a[8]$ the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

$$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$$

The time complexity for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$.

Successful searches
 $\Theta(1)$, $\Theta(\log n)$, $\Theta(\log n)$
 Best average worst

un-successful searches
 $\Theta(\log n)$
 best, average and worst

Analysis for worst case

Let $T(n)$ be the time complexity of Binary search

The algorithm sets mid to $\lceil (n+1) / 2 \rceil$

Therefore,

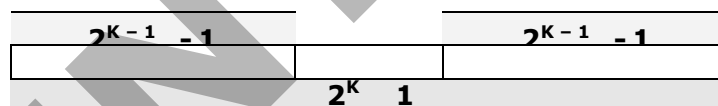
$$T(0) = 0$$

$$T(n) = 1 \quad \text{if } x = a[\text{mid}]$$

$$= 1 + T(\lceil (n+1) / 2 \rceil - 1) \quad \text{if } x < a[\text{mid}]$$

$$= 1 + T(n - \lceil (n+1) / 2 \rceil) \quad \text{if } x > a[\text{mid}]$$

Let us restrict 'n' to values of the form $n = 2^k - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{2^k - 1 + 1}{2} \right\rceil = 2^{k-1}$ for $k \geq 1$

Giving,

$$T(0) = 0$$

$$T(2^k - 1) = 1 \quad \text{if } x = a[\text{mid}]$$

$$= 1 + T(2^{k-1} - 1) \quad \text{if } x < a[\text{mid}]$$

$$= 1 + T(2^{k-1} - 1) \quad \text{if } x > a[\text{mid}]$$

In the worst case the test $x = a[\text{mid}]$ always fails, so

$$w(0) = 0$$

$$w(2^k - 1) = 1 + w(2^{k-1} - 1)$$

This is now solved by repeated substitution:

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

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^k - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

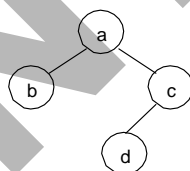
for $n = 2^k - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^k - 1$ weakens the result. In practice this does not matter very much, $w(n)$ is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^k - 1$.

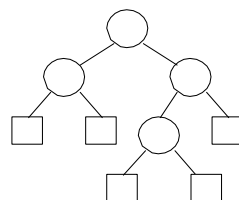
External and Internal path length:

The lines connecting nodes to their non-empty sub trees are called edges. A non-empty binary tree with n nodes has n-1 edges. The size of the tree is the number of nodes it contains.

When drawing binary trees, it is often convenient to represent the empty sub trees explicitly, so that they can be seen. For example:

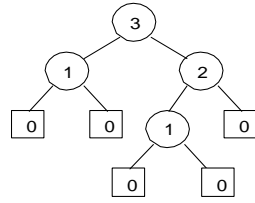


The tree given above in which the empty sub trees appear as square nodes is as follows:

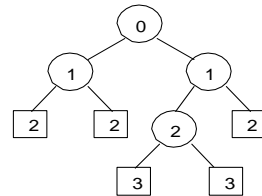


The square nodes are called as external nodes $E(T)$. The square node version is sometimes called an extended binary tree. The round nodes are called internal nodes $I(T)$. A binary tree with n internal nodes has n+1 external nodes.

The height $h(x)$ of node 'x' is the number of edges on the longest path leading down from 'x' in the extended tree. For example, the following tree has heights written inside its nodes:



The depth $d(x)$ of node ' x ' is the number of edges on path from the root to ' x '. It is the number of internal nodes on this path, excluding ' x ' itself. For example, the following tree has depths written inside its nodes:



The internal path length $I(T)$ is the sum of the depths of the internal nodes of ' T ':

$$I(T) = \sum_{x \in I(T)} d(x)$$

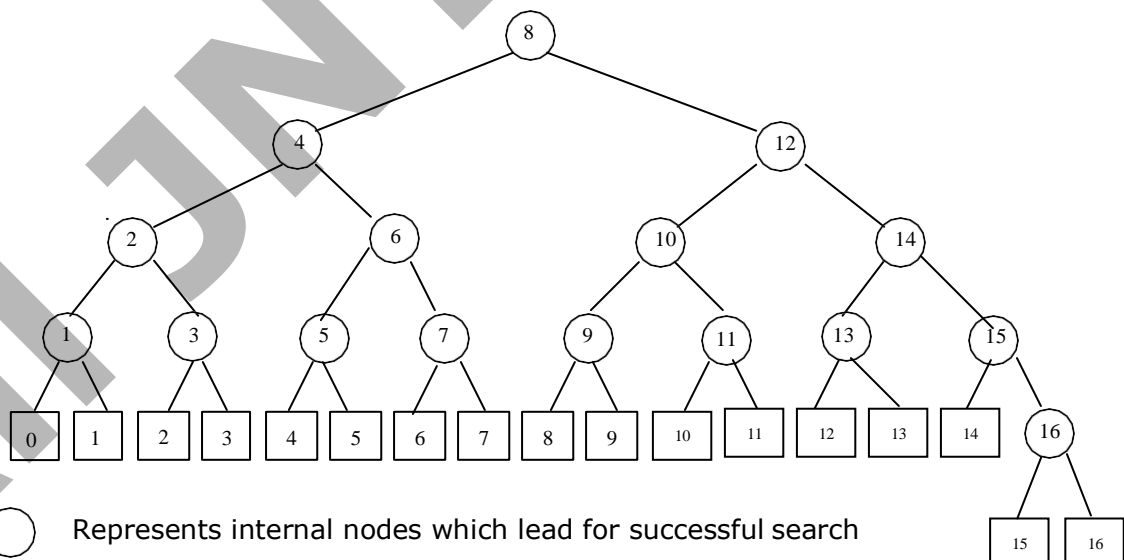
The external path length $E(T)$ is the sum of the depths of the external nodes:

$$E(T) = \sum_{x \in E(T)} d(x)$$

For example, the tree above has $I(T) = 4$ and $E(T) = 12$.

A binary tree T with ' n ' internal nodes, will have $I(T) + 2n = E(T)$ external nodes.

A binary tree corresponding to binary search when $n = 16$ is



○ Represents internal nodes which lead for successful search

□ External square nodes, which lead for unsuccessful search.

Let C_N be the average number of comparisons in a successful search.

C'_N be the average number of comparison in an un successful search.

Then we have,

$$C_N = 1 + \frac{\text{internal path length of tree}}{N}$$

$$C'_N = \frac{\text{External path length of tree}}{N + 1}$$

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

External path length is always $2N$ more than the internal path length.

Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge sort in the *best case*, *worst case* and *average case* is $O(n \log n)$ and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr*, *b ptr* and *c ptr*, which are initially set to the beginning of their respective arrays. The smaller of $a[a \text{ ptr}]$ and $b[b \text{ ptr}]$ is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to 'c'.

To illustrate how merge process works. For example, let us consider the array 'a' containing 1, 13, 24, 26 and 'b' containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
<i>h</i>			
<i>ptr</i>			

5	6	7	8
2	15	27	28
<i>j</i>			
<i>ptr</i>			

1	2	3	4	5	6	7	8
1							
<i>i</i>							
<i>ptr</i>							

and then 2 and 13 are compared. 2 is added to 'c'. Increment *b ptr* and *c ptr*.

1	2	3	4
1	13	24	26
	<i>h</i>		
	<i>ptr</i>		

5	6	7	8
2	15	27	28
<i>j</i>			
<i>ptr</i>			

1	2	3	4	5	6	7	8
1	2						
	<i>i</i>						
	<i>ptr</i>						

then 13 and 15 are compared. 13 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
	<i>h</i> <i>ptr</i>		

5	6	7	8
2	15	27	28
	<i>j</i> <i>ptr</i>		

1	2	3	4	5	6	7	8
1	2	13					
		<i>i</i> <i>ptr</i>					

24 and 15 are compared. 15 is added to 'c'. Increment *b ptr* and *c ptr*.

1	2	3	4
1	13	24	26
		<i>h</i> <i>ptr</i>	

5	6	7	8
2	15	27	28
	<i>j</i> <i>ptr</i>		

1	2	3	4	5	6	7	8
1	2	13	15				
			<i>i</i> <i>ptr</i>				

24 and 27 are compared. 24 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
		<i>h</i> <i>ptr</i>	

5	6	7	8
2	15	27	28
		<i>j</i> <i>ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24			
			<i>i</i> <i>ptr</i>				

26 and 27 are compared. 26 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
			<i>h</i> <i>ptr</i>

5	6	7	8
2	15	27	28
		<i>j</i> <i>ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24	26		
					<i>i</i> <i>ptr</i>		

As one of the lists is exhausted. The remainder of the b array is then copied to 'c'.

1	2	3	4
1	13	24	26
			<i>h</i> <i>ptr</i>

5	6	7	8
2	15	27	28
		<i>j</i> <i>ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24	26	27	28
							<i>i</i> <i>ptr</i>

Algorithm

Algorithm MERGESORT (low, high)

// a (low : high) is a global array to be sorted.

```
{
    if (low < high)
    {
        mid := |(low + high)/2|           //finds where to split the set
        MERGESORT(low, mid)              //sort one subset
        MERGESORT(mid+1, high)           //sort the other subset
        MERGE(low, mid, high)            // combine the results
    }
}
```

Algorithm MERGE (low, mid, high)

// a (low : high) is a global array containing two sorted subsets

// in a (low : mid) and in a (mid + 1 : high).

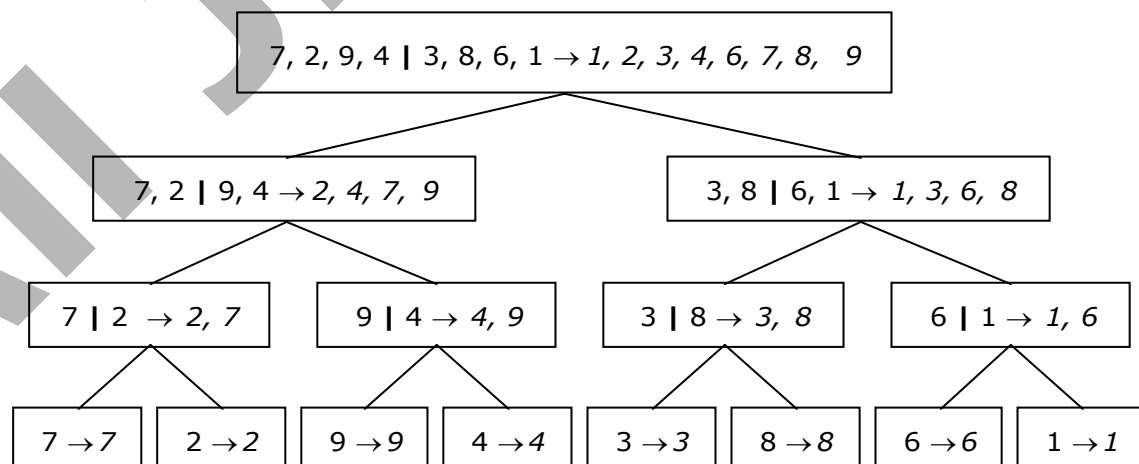
// The objective is to merge these sorted sets into single sorted

// set residing in a (low : high). An auxiliary array B is used.

```
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h]; h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do
        a[k] := b[k];
}
```

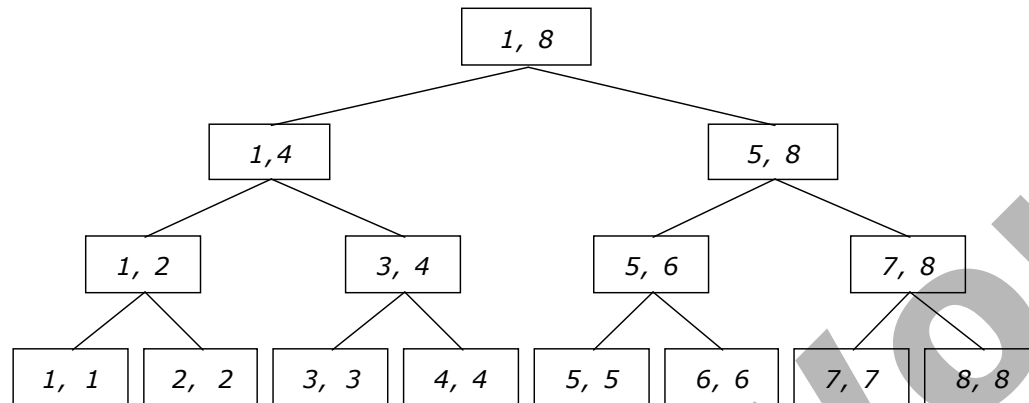
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



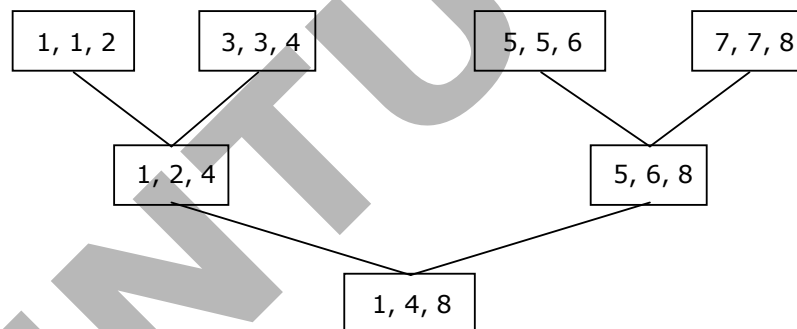
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For $n = 1$, the time to merge sort is constant, which we will denote by 1. Otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size $n/2$, plus the time to merge, which is linear. The equation says this exactly:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2 T(n/2) + n \end{aligned}$$

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right-hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute $n/2$ into this main equation

$$\begin{aligned} 2 T(n/2) &= 2 (2 (T(n/4)) + n/2) \\ &= 4 T(n/4) + n \end{aligned}$$

We have,

$$\begin{aligned} T(n/2) &= 2 T(n/4) + n \\ T(n) &= 4 T(n/4) + 2n \end{aligned}$$

Again, by substituting $n/4$ into the main equation, we see that

$$\begin{aligned} 4T(n/4) &= 4 (2T(n/8)) + n/4 \\ &= 8 T(n/8) + n \end{aligned}$$

So we have,

$$\begin{aligned} T(n/4) &= 2 T(n/8) + n \\ T(n) &= 8 T(n/8) + 3n \end{aligned}$$

Continuing in this manner, we obtain:

$$T(n) = 2^k T(n/2^k) + K \cdot n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

$$\begin{aligned} T(n) &= 2^{\log_2 n} T\left(\frac{2^k}{2^k}\right) + \log_2 n \cdot n \\ &= n T(1) + n \log n \\ &= n \log n + n \end{aligned}$$

Representing this in O notation:

$$T(n) = O(n \log n)$$

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is $O(n \log n)$, it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is $O(n \log n)$.*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two $n \times n$ matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
  for j := 1 to n do
    c[i, j] := 0;
    for K: = 1 to n do
      c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us consider three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

This leads to a divide-and-conquer algorithm, which performs $n \times n$ matrix multiplication by partitioning the matrices into quarters and performing eight $(n/2) \times (n/2)$ matrix multiplications and four $(n/2) \times (n/2)$ matrix additions.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 8 T(n/2) \end{aligned}$$

Which leads to $T(n) = O(n^3)$, where n is the power of 2.

Strassen's insight was to find an alternative method for calculating the C_{ij} , requiring seven $(n/2) \times (n/2)$ matrix multiplications and eighteen $(n/2) \times (n/2)$ matrix additions and subtractions:

$$P = (A_{11} + A_{22}) (B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) B_{11}$$

$$R = A_{11} (B_{12} - B_{22})$$

$$S = A_{22} (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) B_{22}$$

$$U = (A_{21} - A_{11}) (B_{11} + B_{12})$$

$$V = (A_{12} - A_{22}) (B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U.$$

This method is used recursively to perform the seven $(n/2) \times (n/2)$ matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 7 T(n/2) \end{aligned}$$

Solving this for the case of $n = 2^k$ is easy:

$$\begin{aligned} T(2^k) &= 7 T(2^{k-1}) \\ &= 7^2 T(2^{k-2}) \\ &= \dots \\ &= 7^i T(2^{k-i}) \end{aligned}$$

$$\begin{aligned} \text{Put } i &= k \\ &= 7^k T(1) \\ &= 7^k \end{aligned}$$

$$\begin{aligned} \text{That is, } T(n) &= 7^{\log_2 n} \\ &= n^{\log_2 7} \\ &= O(n^{\log_2 7}) = O(n^{2.81}) \end{aligned}$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence w_1, w_2, \dots, w_n take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare has devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is $O(n \log n)$.

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function `partition()` makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until $a[i] \geq \text{pivot}$.
- Repeatedly decrease the pointer 'j' until $a[j] \leq \text{pivot}$.

- If $j > i$, interchange $a[j]$ with $a[i]$
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition $low \geq high$ is satisfied. This condition will be satisfied only when the array is completely sorted.
- Here we choose the first element as the 'pivot'. So, $pivot = x[low]$. Now it calls the partition function to find the proper position j of the element $x[low]$ i.e. pivot. Then we will have two sub-arrays $x[low], x[low+1], \dots, \dots x[j-1]$ and $x[j+1], x[j+2], \dots x[high]$.
- It calls itself recursively to sort the left sub-array $x[low], x[low+1], \dots, \dots x[j-1]$ between positions low and $j-1$ (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array $x[j+1], x[j+2], \dots, \dots x[high]$ between positions $j+1$ and $high$.

Algorithm Algorithm

QUICKSORT(low, high)

/* sorts the elements $a(low), \dots, a(high)$ which reside in the global array $A(1 : n)$ into ascending order $a(n+1)$ is considered to be defined and must be greater than all elements in $a(1 : n)$; $A(n+1) = +\infty$ */

```
{
    if low < high then
    {
        j := PARTITION(a, low, high+1);
        // J is the position of the partitioning element
        QUICKSORT(low, j - 1);
        QUICKSORT(j + 1, high);
    }
}
```

Algorithm PARTITION(a, m, p)

```
{
    V ← a(m); i ← m; j ← p; // A (m) is the partition element
    do
    {
        loop i := i + 1 until a(i) ≥ V // i moves left to right
        loop j := j - 1 until a(j) ≤ V // p moves right to left
        if (i < j) then INTERCHANGE(a, i, j)
    } while (i ≥ j);
    a[m] := a[j]; a[j] := V; // the partition element belongs at position P
    return j;
}
```

Algorithm INTERCHANGE(a, i, j)

```

{
    P:=a[i];
    a[i] := a[j];
    a[j] := p;
}

```

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						
(24	08	16	06	04	02)	38	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	24								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	08	(16)									swap pivot & j
	pivot, j	i											
	(04)	06											swap pivot & j
	04												
	pivot, j, i												
				16									
				pivot, j, i									
(02	04	06	08	16	24)	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	56	(58	79	70	57)	swap pivot & j
							45 pivot, j, i						swap pivot & j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	58	(70	79)	swap pivot & j
									57 pivot, j, i				
											(70	79)	
										pivot, j	i		swap pivot & j
										70			
												79 pivot, j, i	
							(45	56	57	58	70	79)	
02	04	06	08	16	24	38	45	56	57	58	70	79	

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take $T(0) = T(1) = 1$, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + Cn \quad - \quad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1 .

Worst Case Analysis

The pivot is the smallest element, all the time. Then $i=0$ and if we ignore $T(0)=1$, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + Cn \quad n > 1 \quad - \quad (2)$$

Using equation – (1) repeatedly, thus

$$T(n-1) = T(n-2) + C(n-1)$$

$$T(n-2) = T(n-3) + C(n-2)$$

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

$$\begin{aligned} T(n) &= T(1) + \sum_{i=2}^n i \\ &= O(n^2) \end{aligned} \quad (3)$$

Best Case Analysis

In the best case, the pivot is in the middle. To simplify the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big - oh answer.

$$T(n) = 2T(n/2) + Cn \quad (4)$$

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \quad (5)$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \quad (6)$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \quad (7)$$

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \quad (8)$$

We add all the equations from 4 to 8 and note that there are log n of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \quad (9)$$

$$\text{Which yields, } T(n) = Cn \log n + n = O(n \log n) \quad (10)$$

This is exactly the same analysis as merge sort, hence we get the same answer.

Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made $n-k+1$ comparisons. So, first call on partition makes $n+1$ comparisons. The average case complexity of quicksort is

$$T(n) = \text{comparisons for first call on quicksort} \\ + \\ \{ \sum_{1 \leq n_{\text{left}}, n_{\text{right}} \leq n} [T(n_{\text{left}}) + T(n_{\text{right}})] \} n = (n+1) + 2 [T(0) + T(1) + T(2) + \dots + T(n-1)]/n$$

$$nT(n) = n(n+1) + 2 [T(0) + T(1) + T(2) + \dots + T(n-2) + T(n-1)]$$

$$(n-1)T(n-1) = (n-1)n + 2 [T(0) + T(1) + T(2) + \dots + T(n-2)] \setminus$$

Subtracting both sides:

$$nT(n) - (n-1)T(n-1) = [n(n+1) - (n-1)n] + 2T(n-1) = 2n + 2T(n-1)$$

$$nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)$$

$$T(n) = 2 + (n+1)T(n-1)/n$$

The recurrence relation obtained is:

$$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$$

Using the method of substitution:

$$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$$

$$T(n-1)/n = 2/n + T(n-2)/(n-1)$$

$$T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)$$

$$T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)$$

.

.

$$T(3)/4 = 2/4 + T(2)/3$$

$$T(2)/3 = 2/3 + T(1)/2 \quad T(1)/2 = 2/2 + T(0)$$

Adding both sides:

$$T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + \dots + T(2)/3 + T(1)/2]$$

$$= [T(n-1)/n + T(n-2)/(n-1) + \dots + T(2)/3 + T(1)/2] + T(0) +$$

$$[2/(n+1) + 2/n + 2/(n-1) + \dots + 2/4 + 2/3]$$

Cancelling the common terms:

$$T(n)/(n+1) = 2[1/2 + 1/3 + 1/4 + \dots + 1/n + 1/(n+1)]$$

$$T(n) = (n+1)2 \left[\sum_{2 \leq k \leq n+1} 1/k \right]$$

$$= 2(n+1) [\quad - \quad]$$

$$= 2(n+1) [\log(n+1) - \log 2]$$

$$= 2n \log(n+1) + \log(n+1) - 2n \log 2 - \log 2$$

$$T(n) = O(n \log n)$$

3.8. Straight insertion sort:

Straight insertion sort is used to create a sorted list (initially list is empty) and at each iteration the top number on the sorted list is removed and put into its proper

place in the sorted list. This is done by moving along the sorted list, from the smallest to the largest number, until the correct place for the new number is located i.e. until all sorted numbers with smaller values comes before it and all those with larger values comes after it. For example, let us consider the following 8 elements for sorting:

Index	1	2	3	4	5	6	7	8
Elements	27	412	71	81	59	14	273	87

Solution:

Iteration 0:	unsorted Sorted	412 27	71	81	59	14	273	87
Iteration 1:	unsorted Sorted	412 27	71 412	81	59	14	273	87
Iteration 2:	unsorted Sorted	71 27	81 71	59 412	14	273	87	
Iteration 3:	unsorted Sorted	81 27	39 71	14 81	273 412	87		
Iteration 4:	unsorted Sorted	59 274	14 59	273 71	87 81	412		
Iteration 5:	unsorted Sorted	14 14	273 27	87 59	71	81	412	
Iteration 6:	unsorted Sorted	273 14	87 27	59	71	81	273	412
Iteration 7:	unsorted Sorted	87 14	27	59	71	81	87	273 412

Chapter 4

Greedy Method

GENERAL METHOD

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm*.

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm*.

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{
    solution :=  $\phi$ ;           // initialize the solution to empty
    for i:=1 to n do
    {
        x := select (a);
        if feasible (solution, x) then
            solution := Union (Solution, x);
    }
    return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from 'a', removes it and assigns its value to 'x'. Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector. The function Union combines 'x' with solution and updates the objective function.

KNAPSACK PROBLEM

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction x_i , $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'. The problem is stated as:

$$\begin{aligned} &\text{maximize } \sum_{i=1}^n p_i x_i \\ &\text{subject to } \sum_{i=1}^n w_i x_i \leq M \quad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n \end{aligned}$$

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of $p[i] / w[i]$ then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

// P[1 : n] and w[1 : n] contain the profits and weights respectively of

// Objects ordered so that $p[i] / w[i] > p[i + 1] / w[i + 1]$.

// m is the knapsack size and x[1: n] is the solution vector.

```
{
    for i := 1 to n do x[i] := 0.0           // initialize x
    U := m;
    for i := 1 to n do
    {
        if (w(i) > U) then break;
        x[i] := 1.0; U := U - w[i];
    }
    if (i ≤ n) then x[i] := U / w[i];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only $O(n)$ time.

Example:

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

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1/2	1/3	1/4	$18 \times 1/2 + 15 \times 1/3 + 10 \times 1/4 = 16.5$	$25 \times 1/2 + 24 \times 1/3 + 15 \times 1/4 = 24.25$

2. Select the object with the maximum profit first ($p = 25$). So, $x_1 = 1$ and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit ($p = 24$). So, $x_2 = 2/15$

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	2/15	0	$18 \times 1 + 15 \times 2/15 = 20$	$25 \times 1 + 24 \times 2/15 = 28.2$

3. Considering the objects in the order of non-decreasing weights w_i .

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	2/3	1	$15 \times 2/3 + 10 \times 1 = 20$	$24 \times 2/3 + 15 \times 1 = 31$

4. Considered the objects in the order of the ratio p_i / w_i .

p_1/w_1	p_2/w_2	p_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / w_i . Select the object with the maximum p_i / w_i ratio, so, $x_2 = 1$ and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / w_i ratio, so $x_3 = 1/2$ and the profit earned is 7.5.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	1	1/2	$15 \times 1 + 10 \times 1/2 = 20$	$24 \times 1 + 15 \times 1/2 = 31.5$

This solution is the optimal solution.

4.4. OPTIMAL STORAGE ON TAPES

There are 'n' programs that are to be stored on a computer tape of length 'L'. Each program 'i' is of length l_i , $1 \leq i \leq n$. All the programs can be stored on the tape if and only if the sum of the lengths of the programs is at most 'L'.

We shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. If the programs are stored in the order $i = i_1, i_2, \dots, i_n$, the time t_j needed to retrieve program i_j is proportional to

$$\sum_{1 \leq k \leq j} l_{i_k}$$

If all the programs are retrieved equally often then the expected or mean retrieval time (MRT) is:

$$\frac{1}{n} \cdot \sum_{1 \leq j \leq n} t_j$$

For the optimal storage on tape problem, we are required to find the permutation for the 'n' programs so that when they are stored on the tape in this order the MRT is minimized.

$$d(I) = \sum_{j=1}^n \sum_{K=1}^j l_{i_k}$$

Example

Let $n = 3$, $(l_1, l_2, l_3) = (5, 10, 3)$. Then find the optimal ordering?

Solution:

There are $n! = 6$ possible orderings. They are:

Ordering I	$d(I)$		
1, 2, 3	$5 + (5 + 10) + (5 + 10 + 3)$	=	38
1, 3, 2	$5 + (5 + 3) + (5 + 3 + 10)$	=	31
2, 1, 3	$10 + (10 + 5) + (10 + 5 + 3)$	=	43
2, 3, 1	$10 + (10 + 3) + (10 + 3 + 5)$	=	41
3, 1, 2	$3 + (3 + 5) + (3 + 5 + 10)$	=	29
3, 2, 1	$3 + (3 + 10) + (3 + 10 + 5)$	=	34

From the above, it simply requires to store the programs in non-decreasing order (increasing order) of their lengths. This can be carried out by using a efficient sorting algorithm (Heap sort). This ordering can be carried out in $O(n \log n)$ time using heap sort algorithm.

The tape storage problem can be extended to several tapes. If there are $m > 1$ tapes, T_0, \dots, T_{m-1} , then the programs are to be distributed over these tapes.

The total retrieval time (RT) is $\sum_{j=0}^{m-1} d(I_j)$

The objective is to store the programs in such a way as to minimize RT.

The programs are to be sorted in non decreasing order of their lengths l_i 's, $l_1 \leq l_2 \leq \dots \leq l_n$.

The first 'm' programs will be assigned to tapes T_0, \dots, T_{m-1} respectively. The next 'm' programs will be assigned to T_0, \dots, T_{m-1} respectively. The general rule is that program i is stored on tape $T_{i \bmod m}$.

Algorithm:

The algorithm for assigning programs to tapes is as follows:

Algorithm Store (n, m)

// n is the number of programs and m the number of tapes

```
{
    j := 0;                                // next tape to store on
    for i := 1 to n do
    {
        Print ('append program', i, 'to permutation for tape', j);
        j := (j + 1) mod m;
    }
}
```

On any given tape, the programs are stored in non-decreasing order of their lengths.

JOB SEQUENCING WITH DEADLINES

When we are given a set of 'n' jobs. Associated with each Job i, deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit p_i is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \leq r \leq k$ are the jobs in 'j' and $d(j[1]) \leq d(j[2]) \leq \dots \leq d(j[k])$. To test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \leq r$, $1 \leq r \leq k+1$.

Example:

Let $n = 4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	OPTIMAL
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

Algorithm GreedyJob (d, J, n)

// J is a set of jobs that can be completed by their deadlines.

```
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J U {i} can be completed by their dead lines)
        then J := J U {i};
    }
}
```

OPTIMAL MERGE PATTERNS

Given ' n ' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge ' n ' sorted files together. This type of merging is called as 2-way merge patterns. To merge an n -record file and an m -record file requires possibly $n + m$ record moves, the obvious choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
```

```
{
    treenode * lchild;
    treenode * rchild;
};
```

Algorithm TREE (n)

// list is a global of n single node binary trees

```
{
    for i := 1 to n - 1 do
    {
        pt ← new treenode
        (pt → lchild) ← least (list);          // merge two trees with smallest
        lengths
        (pt → rchild) ← least (list);
        (pt → weight) ← ((pt → lchild) → weight) + ((pt → rchild) → weight);
        insert (list, pt);
    }
    return least (list);                        // The tree left in list is the merge
tree
}
```


Example 1:

Suppose we are having three sorted files X_1 , X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

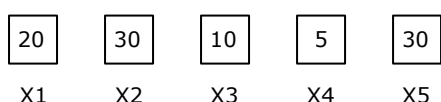
S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	$X_1 \& X_2 = T_1$	50	$T_1 \& X_3$	60	$50 + 60 = 110$
2.	$X_2 \& X_3 = T_1$	30	$T_1 \& X_1$	60	$30 + 60 = 90$

The Second case is optimal.

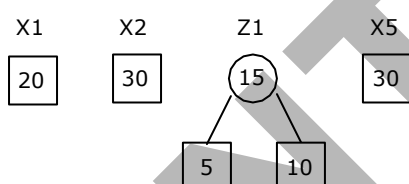
Example 2:

Given five files (X_1, X_2, X_3, X_4, X_5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.

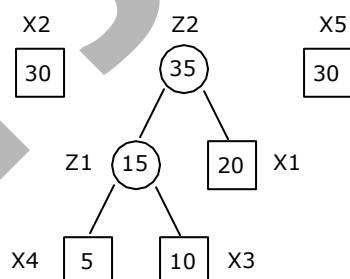
Solution:



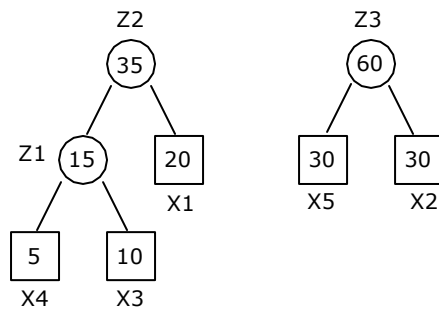
Merge X_4 and X_3 to get 15 record moves. Call this Z_1 .



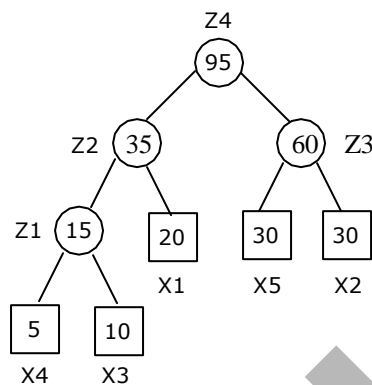
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2 .



Merge X_2 and X_5 to get 60 record moves. Call this Z_3 .



Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4 .



Therefore the total number of record moves is $15 + 35 + 60 + 95 = 205$. This is an optimal merge pattern for the given problem.

Huffman Codes

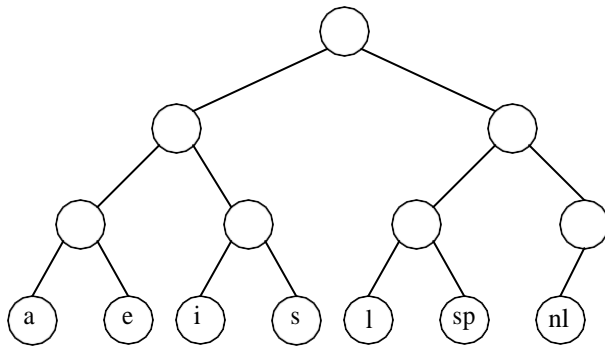
Another application of Greedy Algorithm is file compression.

Suppose that we have a file only with characters a, e, i, s, t, spaces and new lines, the frequency of appearance of a's is 10, e's fifteen, twelve i's, three s's, four t's, thirteen banks and one newline.

Using a standard coding scheme, for 58 characters using 3 bits for each character, the file requires 174 bits to represent. This is shown in table below.

<u>Character</u>	<u>Code</u>	<u>Frequency</u>	<u>Total bits</u>
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
Space	101	13	39
New line	110	1	3

Representing by a binary tree, the binary code for the alphabets are as follows:

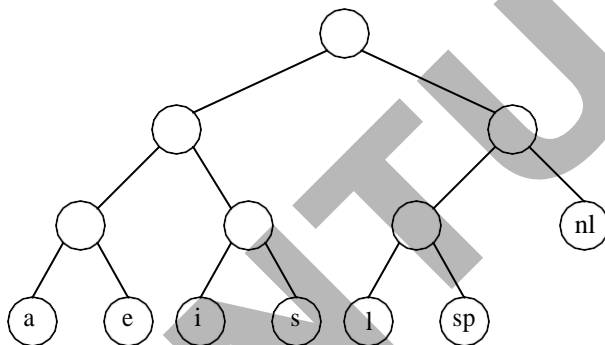


The representation of each character can be found by starting at the root and recording the path. Use a 0 to indicate the left branch and a 1 to indicate the right branch.

If the character c_i is at depth d_i and occurs f_i times, the cost of the code is equal to $\sum d_i f_i$

With this representation the total number of bits is $3 \times 10 + 3 \times 15 + 3 \times 12 + 3 \times 3 + 3 \times 4 + 3 \times 13 + 3 \times 1 = 174$

A better code can be obtained by with the following representation.



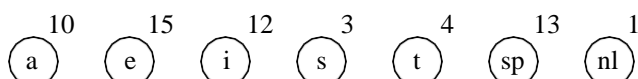
The basic problem is to find the full binary tree of minimal total cost. This can be done by using Huffman coding (1952).

Huffman's Algorithm:

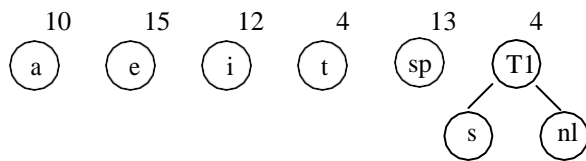
Huffman's algorithm can be described as follows: We maintain a forest of trees. The weights of a tree is equal to the sum of the frequencies of its leaves. If the number of characters is 'c'. $c - 1$ times, select the two trees T_1 and T_2 , of smallest weight, and form a new tree with sub-trees T_1 and T_2 . Repeating the process we will get an optimal Huffman coding tree.

Example:

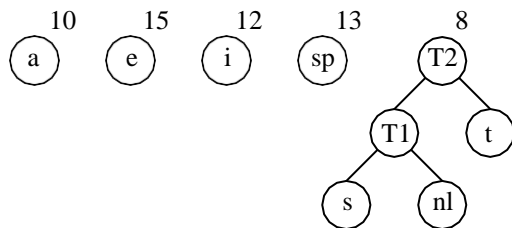
The initial forest with the weight of each tree is as follows:



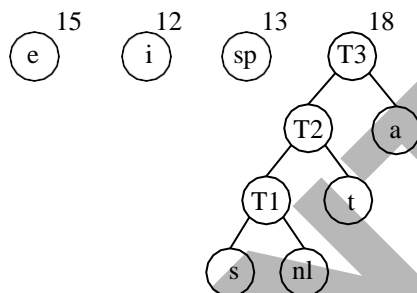
The two trees with the lowest weight are merged together, creating the forest, the Huffman algorithm after the first merge with new root T_1 is as follows: The total weight of the new tree is the sum of the weights of the old trees.



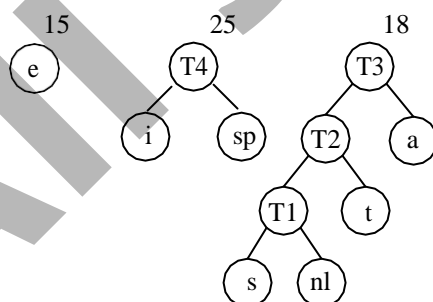
We again select the two trees of smallest weight. This happens to be T_1 and t , which are merged into a new tree with root T_2 and weight 8.



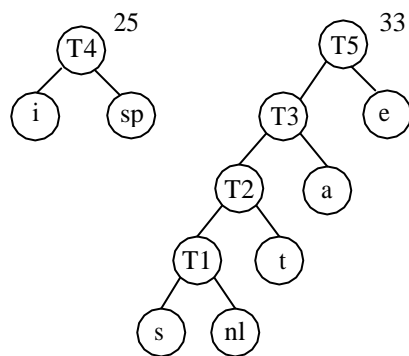
In next step we merge T_2 and a creating T_3 , with weight $10+8=18$. The result of this operation is



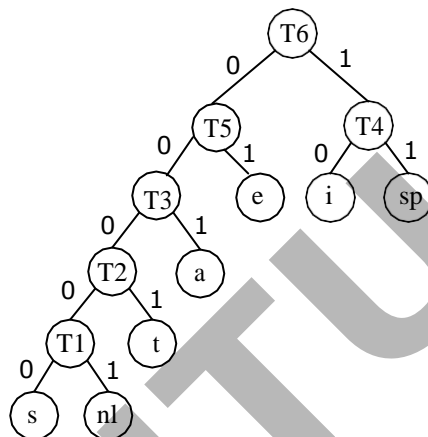
After third merge, the two trees of lowest weight are the single node trees representing i and the blank space. These trees merged into the new tree with root T_4 .



The fifth step is to merge the trees with roots e and T₃. The results of this step is



Finally, the optimal tree is obtained by merging the two remaining trees. The optimal trees with root T₆ is:



The full binary tree of minimal total cost, where all characters are obtained in the leaves, uses only 146 bits.

Character	Code	Frequency	Total bits (Code bits X frequency)
A	001	10	30
E	01	15	30
I	10	12	24
S	00000	3	15
T	0001	4	16
Space	11	13	26
New line	00001	1	5
		Total :	146

GRAPH ALGORITHMS

Basic Definitions:

- **Graph G** is a pair (V, E) , where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote $n := |V|$, $m := |E|$.
- Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If $(u, v) \in E$, then vertices u , and v are adjacent.
- We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called **weighted**.
- **Degree** of a vertex v is the number of vertices u for which $(u, v) \in E$ (denote $\deg(v)$). The number of **incoming edges** to a vertex v is called **in-degree** of the vertex (denote $\text{indeg}(v)$). The number of **outgoing edges** from a vertex is called **out-degree** (denote $\text{outdeg}(v)$).

Representation of Graphs:

Consider graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$.

Adjacency matrix represents the graph as an $n \times n$ matrix $A = (a_{ij})$, where

$$a_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

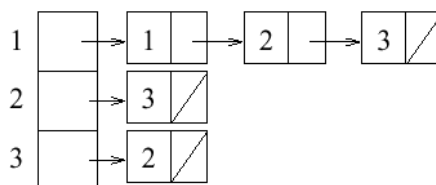
$$a_{i,j} = \begin{cases} w(v_i, v_j), & \text{if } (v_i, v_j) \in E, \\ \text{default}, & \text{otherwise,} \end{cases}$$

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be ∞ , meaning value larger than any other value).

Adjacency List: An array $\text{Adj}[1 \dots n]$ of pointers where for $1 \leq v \leq n$, $\text{Adj}[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

Adjacency matrix



Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices $(v_1, v_2, \dots, v_k, v_{k+1} = v_1)$, where for all i , $(v_i, v_{i+1}) \in E$ and all vertices in the cycle are distinct except pair v_1, v_{k+1} .

Subgraphs and Spanning Trees:

Subgraphs: A graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v . If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

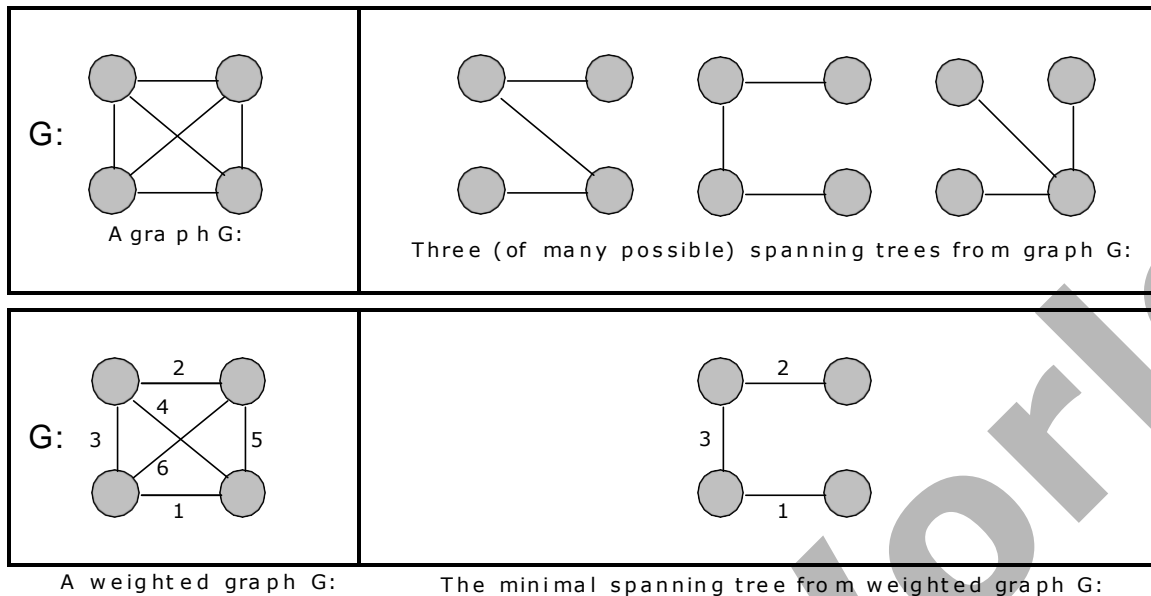
Lemma 1: Let T be a spanning tree of a graph G . Then

1. Any two vertices in T are connected by a unique simple path.
2. If any edge is removed from T , then T becomes disconnected.
3. If we add any edge into T , then the new graph will contain a cycle.
4. Number of edges in T is $n-1$.

Minimum Spanning Trees (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.



Here are some examples:

To explain further upon the Minimum Spanning Tree, and what it applies to, let's consider a couple of real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until $(n - 1)$ edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

Algorithm Kruskal (E, cost, n, t)

```
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
    Construct a heap out of the edge costs using heapify;
    for i := 1 to n do parent [i] := -1;           // Each vertex is in a different set.

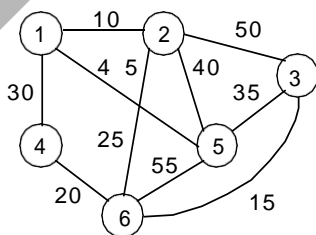
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap and
        re-heapify using Adjust;
        j := Find (u); k := Find (v);
        if (j ≠ k) then
        {
            i := i + 1;
            t [i, 1] := u; t [i, 2] := v;
            mincost := mincost + cost [u, v];
            Union (j, k);
        }
    }
    if (i ≠ n-1) then write ("no spanning tree");
    else return mincost;
}
```

Running time:

- The number of finds is at most $2e$, and the number of unions at most $n-1$. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than $O(n + e)$.
- We can add at most $n-1$ edges to tree T . So, the total time for operations on T is $O(n)$.

Summing up the various components of the computing times, we get $O(n + e \log e)$ as asymptotic complexity



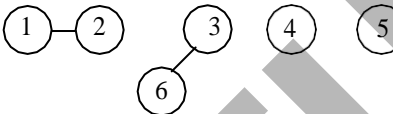
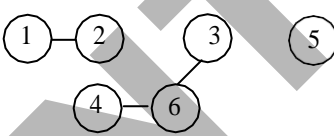
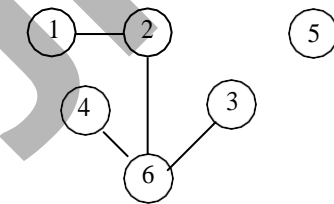
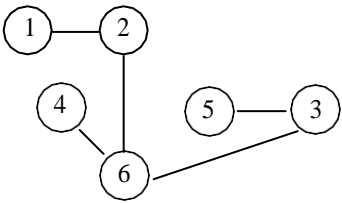
Example 1:



Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Finds** on the vertex sets suffice. When an edge is included in T , two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$	
(1, 2)	10		$\{1, 2\}, \{3\}, \{4\}, \{5\}, \{6\}$	The vertices 1 and 2 are in different sets, so the edge is combined
(3, 6)	15		$\{1, 2\}, \{3, 6\}, \{4\}, \{5\}$	The vertices 3 and 6 are in different sets, so the edge is combined
(4, 6)	20		$\{1, 2\}, \{3, 4, 6\}, \{5\}$	The vertices 4 and 6 are in different sets, so the edge is combined
(2, 6)	25		$\{1, 2, 3, 4, 6\}, \{5\}$	The vertices 2 and 6 are in different sets, so the edge is combined
(1, 4)	30	Reject		The vertices 1 and 4 are in the same set, so the edge is rejected
(3, 5)	35		$\{1, 2, 3, 4, 5, 6\}$	The vertices 3 and 5 are in the same set, so the edge is combined

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

Algorithm Algorithm Prim

(E, cost, n, t)

```
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or  $\infty$  if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost [k, l];
    t [1, 1] := k; t [1, 2] := l;
    for i := 1 to n do // Initialize near
        if (cost [i, l] < cost [i, k]) then near [i] := l;
        else near [i] := k;
    near [k] := near [l] := 0;
    for i := 2 to n - 1 do // Find n - 2 additional edges for t.
    {
        Let j be an index such that near [j]  $\neq$  0 and
        cost [j, near [j]] is minimum;
        t [i, 1] := j; t [i, 2] := near [j];
        mincost := mincost + cost [j, near [j]];
        near [j] := 0
        for k := 1 to n do // Update near[].
            if ((near [k]  $\neq$  0) and (cost [k, near [k]] > cost [k, j]))
                then near [k] := j;
    }
    return mincost;
}
```

Running time:

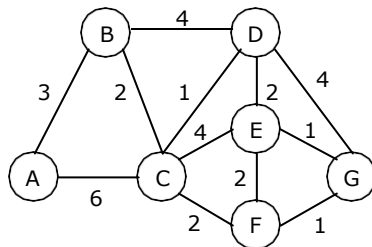
We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get $O(n^2)$ time when we implement dist with array, $O(n + |E| \log n)$ when we implement it with a heap.

For each vertex u in the graph we dequeue it and check all its neighbors in $\Theta(1 + \deg(u))$ time. Therefore the running time is:

$$\Theta\left(\sum_{v \in V} 1 + \deg(v)\right) = \Theta\left(n + \sum_{v \in V} \deg(v)\right) = \Theta(n + m)$$

EXAMPLE 1:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



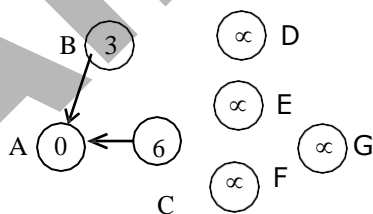
SOLUTION:

The cost adjacency matrix is

$$\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$$

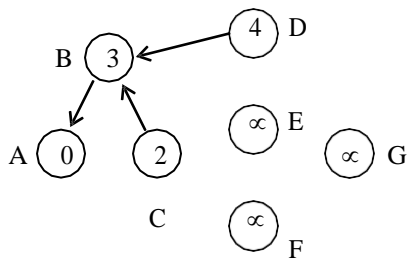
The stepwise progress of the prim's algorithm is as follows:

Step 1:



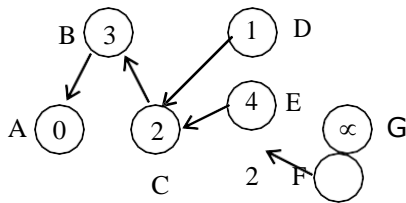
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∞	∞	∞	∞
Next	*	A	A	A	A	A	A

Step 2:



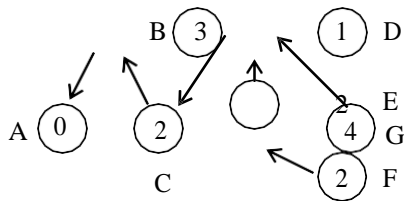
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∞	∞	∞
Next	*	A	B	B	A	A	A

Step 3:



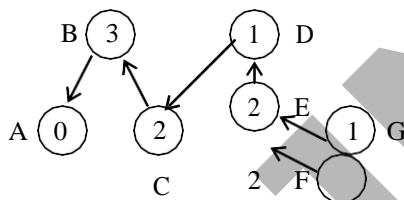
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∞
Next	*	A	B	C	C	C	A

Step 4:



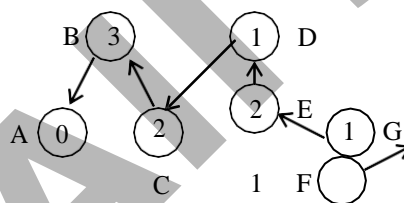
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



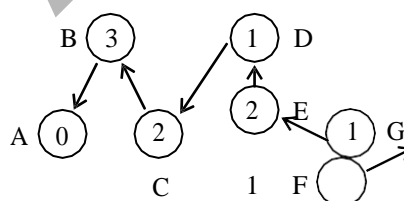
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

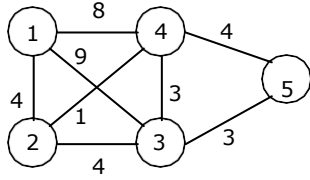
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

EXAMPLE 2:

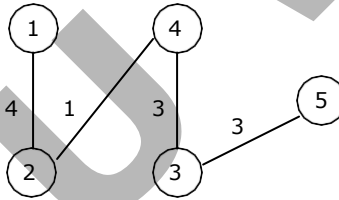
Considering the following graph, find the minimal spanning tree using prim's algorithm.



The cost adjacent matrix is
$$\begin{pmatrix} \infty & 4 & 9 & 8 & \infty \\ 4 & \infty & 1 & 4 & \infty \\ 9 & 4 & \infty & 3 & 3 \\ 8 & 1 & 3 & \infty & 4 \\ \infty & \infty & 3 & 4 & \infty \end{pmatrix}$$

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2



The cost of Minimal spanning tree = 11.

The steps as per the algorithm are as follows:

Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

K = 2, l = 4

Min cost = cost (2, 4) = 1

T [1, 1] = 2

T [1, 2] = 4

<p>for i = 1 to 5</p> <p>Begin</p> <p>i = 1</p> <p>is cost (1, 4) < cost (1, 2)</p> <p>8 < 4, No</p> <p>Than near (1) = 2</p> <p>i = 2</p> <p>is cost (2, 4) < cost (2, 2)</p> <p>1 < ∞, Yes</p> <p>So near [2] = 4</p> <p>i = 3</p> <p>is cost (3, 4) < cost (3, 2)</p> <p>1 < 4, Yes</p> <p>So near [3] = 4</p> <p>i = 4</p> <p>is cost (4, 4) < cost (4, 2)</p> <p>∞ < 1, no</p> <p>So near [4] = 2</p> <p>i = 5</p> <p>is cost (5, 4) < cost (5, 2)</p> <p>4 < ∞, yes</p> <p>So near [5] = 4</p> <p>end</p> <p>near [k] = near [l] = 0</p> <p>near [2] = near[4] = 0</p>	<p>Near matrix</p> <table><tr><td>2</td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> <table><tr><td>2</td><td>4</td><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> <table><tr><td>2</td><td>4</td><td>4</td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> <table><tr><td>2</td><td>4</td><td>4</td><td>2</td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> <table><tr><td>2</td><td>4</td><td>4</td><td>2</td><td>4</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table> <table><tr><td>2</td><td>0</td><td>4</td><td>0</td><td>4</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2					1	2	3	4	5	2	4				1	2	3	4	5	2	4	4			1	2	3	4	5	2	4	4	2		1	2	3	4	5	2	4	4	2	4	1	2	3	4	5	2	0	4	0	4	1	2	3	4	5	<p>Edges added to min spanning tree:</p> <p>T [1, 1] = 2</p> <p>T [1, 2] = 4</p>
2																																																														
1	2	3	4	5																																																										
2	4																																																													
1	2	3	4	5																																																										
2	4	4																																																												
1	2	3	4	5																																																										
2	4	4	2																																																											
1	2	3	4	5																																																										
2	4	4	2	4																																																										
1	2	3	4	5																																																										
2	0	4	0	4																																																										
1	2	3	4	5																																																										
<p>for i = 2 to n-1 (4) do</p> <p>i = 2</p> <p>for j = 1 to 5</p> <p>j = 1</p> <p>near(1)≠0 and cost(1, near(1))</p> <p>2 ≠ 0 and cost (1, 2) = 4</p> <p>j = 2</p> <p>near (2) = 0</p> <p>j = 3</p> <p>is near (3) ≠ 0</p> <p>4 ≠ 0 and cost (3, 4) = 3</p>																																																														

$j = 4$
 $\text{near}(4) = 0$

$J = 5$
 $\text{Is near}(5) \neq 0$
 $4 \neq 0$ and $\text{cost}(4, 5) = 4$

select the min cost from the
 above obtained costs, which is
 3 and corresponding $J = 3$

$\text{min cost} = 1 + \text{cost}(3, 4)$
 $= 1 + 3 = 4$

$T(2, 1) = 3$
 $T(2, 2) = 4$

$\text{Near}[j] = 0$
 i.e. $\text{near}(3) = 0$

for ($k = 1$ to n)

$K = 1$
 $\text{is near}(1) \neq 0$, yes
 $2 \neq 0$
 and $\text{cost}(1, 2) > \text{cost}(1, 3)$
 $4 > 9$, No

$K = 2$
 $\text{Is near}(2) \neq 0$, No

$K = 3$
 $\text{Is near}(3) \neq 0$, No

$K = 4$
 $\text{Is near}(4) \neq 0$, No

$K = 5$
 $\text{Is near}(5) \neq 0$
 $4 \neq 0$, yes
 and $\text{cost}(5, 4) > \text{cost}(5, 3)$
 $4 > 3$, yes
 than $\text{near}(5) = 3$

$i = 3$

for ($j = 1$ to 5)

$J = 1$
 $\text{is near}(1) \neq 0$
 $2 \neq 0$
 $\text{cost}(1, 2) = 4$

$J = 2$
 $\text{Is near}(2) \neq 0$, No

2	0	0	0	4
1	2	3	4	5

$T(2, 1) = 3$
 $T(2, 2) = 4$

2	0	0	0	3
1	2	3	4	5

$J = 3$
 Is near (3) $\neq 0$, no
 Near (3) = 0

 $J = 4$
 Is near (4) $\neq 0$, no
 Near (4) = 0

 $J = 5$
 Is near (5) $\neq 0$
 Near (5) = 3 $\rightarrow 3 \neq 0$, yes
 And cost (5, 3) = 3

 Choosing the min cost from
 the above obtaining costs
 which is 3 and corresponding J
 = 5

 Min cost = 4 + cost (5, 3)
 = 4 + 3 = 7

$T(3, 1) = 5$
 $T(3, 2) = 3$

Near (J) = 0 \rightarrow near (5) = 0
for (k=1 to 5)

2	0	0	0	0
1	2	3	4	5

$k = 1$
 is near (1) $\neq 0$, yes
 and cost(1,2) > cost(1,5)
 $4 > \infty$, No

$K = 2$
 Is near (2) $\neq 0$ no

$K = 3$
 Is near (3) $\neq 0$ no

$K = 4$
 Is near (4) $\neq 0$ no

$K = 5$
 Is near (5) $\neq 0$ no

i = 4

for J = 1 to 5
 $J = 1$
 Is near (1) $\neq 0$
 $2 \neq 0$, yes
 cost (1, 2) = 4

$j = 2$
 is near (2) $\neq 0$, No

$T(3, 1) = 5$
 $T(3, 2) = 3$

$J = 3$
 Is near (3) $\neq 0$, No
 Near (3) = 0

 $J = 4$
 Is near (4) $\neq 0$, No
 Near (4) = 0

 $J = 5$
 Is near (5) $\neq 0$, No
 Near (5) = 0

 Choosing min cost from the above it is only '4' and corresponding $J = 1$

 Min cost = 7 + cost (1,2)
 = 7+4 = 11

 $T(4, 1) = 1$
 $T(4, 2) = 2$

 Near (J) = 0 \rightarrow Near (1) = 0

for (k = 1 to 5)

 $K = 1$
 Is near (1) $\neq 0$, No

 $K = 2$
 Is near (2) $\neq 0$, No

 $K = 3$
 Is near (3) $\neq 0$, No

 $K = 4$
 Is near (4) $\neq 0$, No

 $K = 5$
 Is near (5) $\neq 0$, No

 End.

0	0	0	0	0
1	2	3	4	5

$T(4, 1) = 1$
 $T(4, 2) = 2$

4.8.7. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

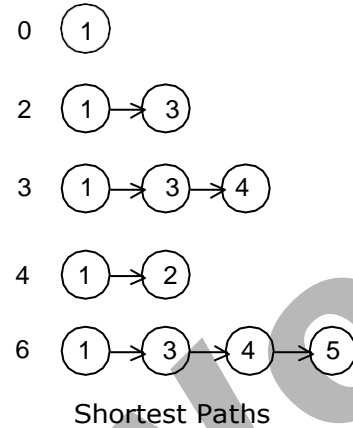
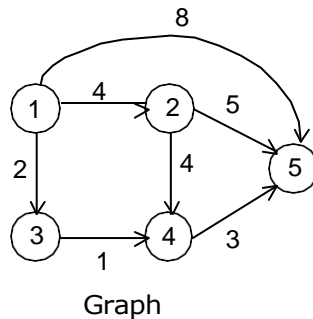
In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the

shortest path between them (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Algorithm:

Algorithm Shortest-Paths (v , $cost$, $dist$, n)

```
// dist [j],  $1 \leq j \leq n$ , is set to the length of the shortest path
// from vertex  $v$  to vertex  $j$  in the digraph  $G$  with  $n$  vertices.
// dist [ $v$ ] is set to zero.  $G$  is represented by its
// cost adjacency matrix  $cost [1:n, 1:n]$ .
{
    for  $i := 1$  to  $n$  do
    {
         $S[i] := false$ ; // Initialize  $S$ .
         $dist[i] := cost[v, i]$ ;
    }
     $S[v] := true$ ;  $dist[v] := 0.0$ ; // Put  $v$  in  $S$ .
    for  $num := 2$  to  $n - 1$  do
    {
        Determine  $n - 1$  paths from  $v$ .
        Choose  $u$  from among those vertices not in  $S$  such that  $dist[u]$  is minimum;
         $S[u] := true$ ; // Put  $u$  in  $S$ .
        for (each  $w$  adjacent to  $u$  with  $S[w] = false$ ) do
            if ( $dist[w] > (dist[u] + cost[u, w])$ ) then // Update distances
                 $dist[w] := dist[u] + cost[u, w]$ ;
    }
}
```

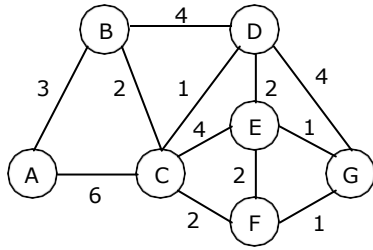
Running time:

Depends on implementation of data structures for $dist$.

- Build a structure with n elements A
- at most $m = |E|$ times decrease the value of an item mB
- 'n' times select the smallest value nC
- For array $A = O(n)$; $B = O(1)$; $C = O(n)$ which gives $O(n^2)$ total.
- For heap $A = O(n)$; $B = O(\log n)$; $C = O(\log n)$ which gives $O(n + m \log n)$ total.

Example 1:

Use Dijkstra's algorithm to find the shortest path from A to each of the other six vertices in the graph:



Solution:

The cost adjacency matrix is

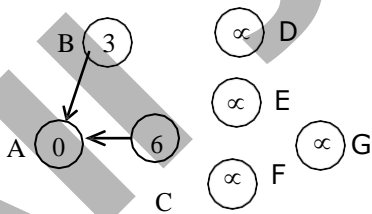
$$\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$$

The problem is solved by considering the following information:

- Status[v] will be either '0', meaning that the shortest path from v to v_0 has definitely been found; or '1', meaning that it hasn't.
- Dist[v] will be a number, representing the length of the shortest path from v to v_0 found so far.
- Next[v] will be the first vertex on the way to v_0 along the shortest path found so far from v to v_0 .

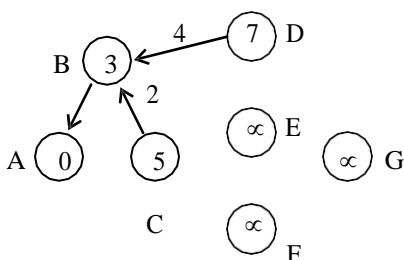
The progress of Dijkstra's algorithm on the graph shown above is as follows:

Step 1:



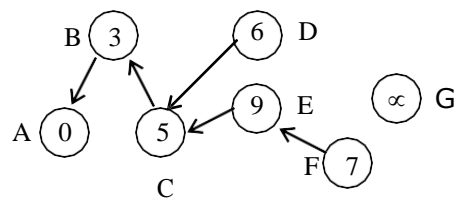
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∞	∞	∞	∞
Next	*	A	A	A	A	A	A

Step 2:



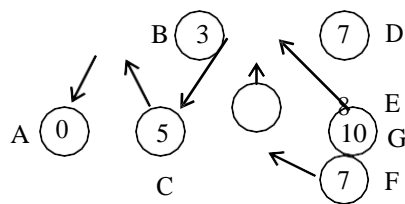
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	∞	∞	∞
Next	*	A	B	B	A	A	A

Step 3:



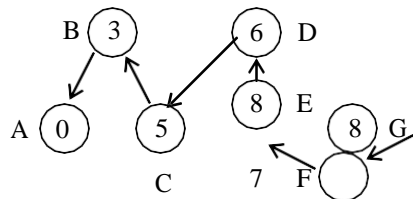
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	∞
Next	*	A	B	C	C	C	A

Step 4:



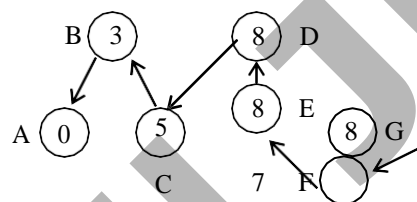
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step 5:



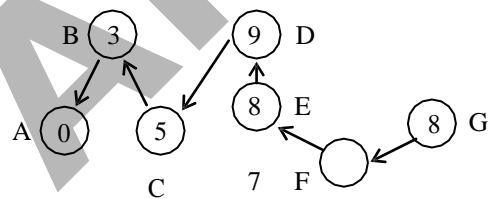
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Chapter

5

Dynamic Programming

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations*, that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seem to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

5.1 MULTI STAGE GRAPHS

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i , $1 \leq i \leq k$. In addition, if $\langle u, v \rangle$ is an edge in E , then $u \in V_i$ and $v \in V_{i+1}$ for some i , $1 \leq i < k$.

Let the vertex 's' be the source, and 't' the sink. Let $c(i, j)$ be the cost of edge $\langle i, j \rangle$. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set V_i defines a stage in the graph. Because of the constraints on E , every path from 's' to 't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k .

A dynamic programming formulation for a k -stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of $k - 2$ decisions. The i^{th} decision involves determining which vertex in V_{i+1} , $1 \leq i \leq k - 2$, is to be on the path. Let $c(i, j)$ be the cost of the path from source to destination. Then using the forward approach, we obtain:

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

ALGORITHM:

Algorithm Fgraph (G, k, n, p)

// The input is a k -stage graph $G = (V, E)$ with n vertices

// indexed in order or stages. E is a set of edges and $c[i, j]$

// is the cost of $\langle i, j \rangle$. $p[1 : k]$ is a minimum cost path.

```
{
    cost[n] := 0.0;
    for j := n - 1 to 1 step - 1 do
    {
        // compute cost [j]
        let r be a vertex such that (j, r) is an edge
        of G and c[j, r] + cost[r] is minimum;
        cost[j] := c[j, r] + cost[r];
        d[j] := r;
    }
    p[1] := 1; p[k] := n; // Find a minimum cost path.
    for j := 2 to k - 1 do p[j] := d[p[j - 1]];
}
```

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

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

Algorithm Bgraph (G, k, n, p)

// Same function as Fgraph

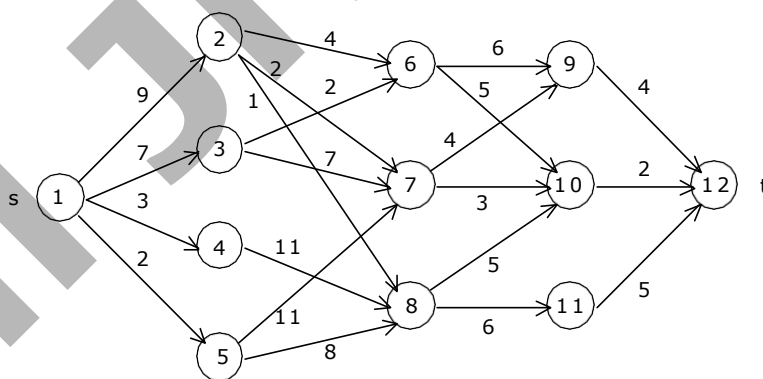
```
{
    Bcost [1] := 0.0;
    for j := 2 to n do
    {
        // Compute Bcost [j].
        Let r be such that (r, j) is an edge of
        G and Bcost [r] + c [r, j] is minimum;
        Bcost [j] := Bcost [r] + c [r, j];
        D [j] := r;
    }
    //find a minimum cost path
    p [1] := 1; p [k] := n;
    for j:= k - 1 to 2 do p [j] := d [p [j + 1]];
}
```

Complexity Analysis:

The complexity analysis of the algorithm is fairly straightforward. Here, if G has $|E|$ edges, then the time for the first for loop is $\Phi(|V| + |E|)$.

EXAMPLE 1:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.



FORWARD APPROACH:

We use the following equation to find the minimum cost path from s to t :

$$cost(i, j) = \min_{l \in V_{i+1}} \{ c(j, l) + cost(i+1, l) \}$$

$$\begin{aligned} \text{cost}(1, 1) &= \min_{\langle j, l \rangle \in E} \{c(1, 2) + \text{cost}(2, 2), c(1, 3) + \text{cost}(2, 3), c(1, 4) + \text{cost}(2, 4), \\ &\quad c(1, 5) + \text{cost}(2, 5)\} \\ &= \min \{9 + \text{cost}(2, 2), 7 + \text{cost}(2, 3), 3 + \text{cost}(2, 4), 2 + \text{cost}(2, 5)\} \end{aligned}$$

Now first starting with,

$$\begin{aligned} \text{cost}(2, 2) &= \min \{c(2, 6) + \text{cost}(3, 6), c(2, 7) + \text{cost}(3, 7), c(2, 8) + \text{cost}(3, 8)\} \\ &= \min \{4 + \text{cost}(3, 6), 2 + \text{cost}(3, 7), 1 + \text{cost}(3, 8)\} \end{aligned}$$

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

$$\text{cost}(4, 9) = \min \{c(9, 12) + \text{cost}(5, 12)\} = \min \{4 + 0\} = 4$$

$$\text{cost}(4, 10) = \min \{c(10, 12) + \text{cost}(5, 12)\} = 2$$

$$\text{Therefore, cost}(3, 6) = \min \{6 + 4, 5 + 2\} = 7$$

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

$$\text{cost}(4, 9) = \min \{c(9, 12) + \text{cost}(5, 12)\} = \min \{4 + 0\} = 4$$

$$\text{Cost}(4, 10) = \min \{c(10, 12) + \text{cost}(5, 12)\} = \min \{2 + 0\} = 2$$

$$\text{Therefore, cost}(3, 7) = \min \{4 + 4, 3 + 2\} = \min \{8, 5\} = 5$$

$$\begin{aligned} \text{cost}(3, 8) &= \min \{c(8, 10) + \text{cost}(4, 10), c(8, 11) + \text{cost}(4, 11)\} \\ &= \min \{5 + \text{cost}(4, 10), 6 + \text{cost}(4, 11)\} \end{aligned}$$

$$\text{cost}(4, 11) = \min \{c(11, 12) + \text{cost}(5, 12)\} = 5$$

$$\text{Therefore, cost}(3, 8) = \min \{5 + 2, 6 + 5\} = \min \{7, 11\} = 7$$

$$\text{Therefore, cost}(2, 2) = \min \{4 + 7, 2 + 5, 1 + 7\} = \min \{11, 7, 8\} = 7$$

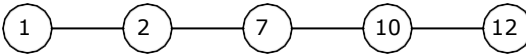
$$\begin{aligned} \text{Therefore, cost}(2, 3) &= \min \{c(3, 6) + \text{cost}(3, 6), c(3, 7) + \text{cost}(3, 7)\} \\ &= \min \{2 + \text{cost}(3, 6), 7 + \text{cost}(3, 7)\} \\ &= \min \{2 + 7, 7 + 5\} = \min \{9, 12\} = 9 \end{aligned}$$

$$\text{cost}(2, 4) = \min \{c(4, 8) + \text{cost}(3, 8)\} = \min \{11 + 7\} = 18$$

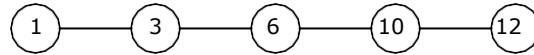
$$\begin{aligned} \text{cost}(2, 5) &= \min \{c(5, 7) + \text{cost}(3, 7), c(5, 8) + \text{cost}(3, 8)\} \\ &= \min \{11 + 5, 8 + 7\} = \min \{16, 15\} = 15 \end{aligned}$$

$$\begin{aligned} \text{Therefore, cost}(1, 1) &= \min \{9 + 7, 7 + 9, 3 + 18, 2 + 15\} \\ &= \min \{16, 16, 21, 17\} = 16 \end{aligned}$$

The minimum cost path is 16.

The path is 

or



BACKWARD APPROACH:

We use the following equation to find the minimum cost path from t to s :

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

$$\begin{aligned} Bcost(5, 12) &= \min \{Bcost(4, 9) + c(9, 12), Bcost(4, 10) + c(10, 12), \\ &\quad Bcost(4, 11) + c(11, 12)\} \\ &= \min \{Bcost(4, 9) + 4, Bcost(4, 10) + 2, Bcost(4, 11) + 5\} \end{aligned}$$

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

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

$$Bcost(2, 2) = \min \{Bcost(1, 1) + c(1, 2)\} = \min \{0 + 9\} = 9$$

$$Bcost(2, 3) = \min \{Bcost(1, 1) + c(1, 3)\} = \min \{0 + 7\} = 7$$

$$Bcost(3, 6) = \min \{9 + 4, 7 + 2\} = \min \{13, 9\} = 9$$

$$\begin{aligned} Bcost(3, 7) &= \min \{Bcost(2, 2) + c(2, 7), Bcost(2, 3) + c(3, 7), \\ &\quad Bcost(2, 5) + c(5, 7)\} \end{aligned}$$

$$Bcost(2, 5) = \min \{Bcost(1, 1) + c(1, 5)\} = 2$$

$$Bcost(3, 7) = \min \{9 + 2, 7 + 7, 2 + 11\} = \min \{11, 14, 13\} = 11$$

$$Bcost(4, 9) = \min \{9 + 6, 11 + 4\} = \min \{15, 15\} = 15$$

$$\begin{aligned} Bcost(4, 10) &= \min \{Bcost(3, 6) + c(6, 10), Bcost(3, 7) + c(7, 10), \\ &\quad Bcost(3, 8) + c(8, 10)\} \end{aligned}$$

$$\begin{aligned} Bcost(3, 8) &= \min \{Bcost(2, 2) + c(2, 8), Bcost(2, 4) + c(4, 8), \\ &\quad Bcost(2, 5) + c(5, 8)\} \end{aligned}$$

$$Bcost(2, 4) = \min \{Bcost(1, 1) + c(1, 4)\} = 3$$

$$Bcost(3, 8) = \min \{9 + 1, 3 + 11, 2 + 8\} = \min \{10, 14, 10\} = 10$$

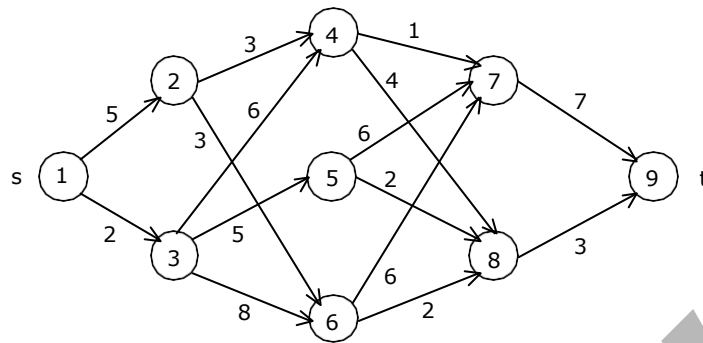
$$Bcost(4, 10) = \min \{9 + 5, 11 + 3, 10 + 5\} = \min \{14, 14, 15\} = 14$$

$$\begin{aligned} Bcost(4, 11) &= \min \{Bcost(3, 8) + c(8, 11)\} = \min \{Bcost(3, 8) + 6\} \\ &= \min \{10 + 6\} = 16 \end{aligned}$$

$$\text{Bcost}(5, 12) = \min \{15 + 4, 14 + 2, 16 + 5\} = \min \{19, 16, 21\} = 16.$$

EXAMPLE 2:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.



SOLUTION:

FORWARD APPROACH:

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

$$\begin{aligned} \text{cost}(1, 1) &= \min \{c(1, 2) + \text{cost}(2, 2), c(1, 3) + \text{cost}(2, 3)\} \\ &= \min \{5 + \text{cost}(2, 2), 2 + \text{cost}(2, 3)\} \end{aligned}$$

$$\begin{aligned} \text{cost}(2, 2) &= \min \{c(2, 4) + \text{cost}(3, 4), c(2, 6) + \text{cost}(3, 6)\} \\ &= \min \{3 + \text{cost}(3, 4), 8 + \text{cost}(3, 6)\} \end{aligned}$$

$$\begin{aligned} \text{cost}(3, 4) &= \min \{c(3, 7) + \text{cost}(4, 7), c(3, 8) + \text{cost}(4, 8)\} \\ &= \min \{1 + \text{cost}(4, 7), 4 + \text{cost}(4, 8)\} \end{aligned}$$

$$\text{cost}(4, 7) = \min \{c(7, 9) + \text{cost}(5, 9)\} = \min \{7 + 0\} = 7$$

$$\text{cost}(4, 8) = \min \{c(8, 9) + \text{cost}(5, 9)\} = 3$$

$$\text{Therefore, cost}(3, 4) = \min \{8, 7\} = 7$$

$$\begin{aligned} \text{cost}(3, 6) &= \min \{c(6, 7) + \text{cost}(4, 7), c(6, 8) + \text{cost}(4, 8)\} \\ &= \min \{6 + \text{cost}(4, 7), 2 + \text{cost}(4, 8)\} = \min \{6 + 7, 2 + 3\} = 5 \end{aligned}$$

$$\text{Therefore, cost}(2, 2) = \min \{10, 8\} = 8$$

$$\text{cost}(2, 3) = \min \{c(3, 4) + \text{cost}(3, 4), c(3, 5) + \text{cost}(3, 5), c(3, 6) + \text{cost}(3, 6)\}$$

$$\begin{aligned} \text{cost}(3, 5) &= \min \{c(5, 7) + \text{cost}(4, 7), c(5, 8) + \text{cost}(4, 8)\} = \min \{6 + 7, 2 + 3\} \\ &= 5 \end{aligned}$$

$$\text{Therefore, cost}(2, 3) = \min \{13, 10, 13\} = 10$$

$$\text{cost}(1, 1) = \min \{5 + 8, 2 + 10\} = \min \{13, 12\} = 12$$

BACKWARD APPROACH:

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

$$\begin{aligned} \text{Bcost}(5, 9) &= \min \{ \text{Bcost}(4, 7) + c(7, 9), \text{Bcost}(4, 8) + c(8, 9) \} \\ &= \min \{ \text{Bcost}(4, 7) + 7, \text{Bcost}(4, 8) + 3 \} \end{aligned}$$

$$\begin{aligned} \text{Bcost}(4, 7) &= \min \{ \text{Bcost}(3, 4) + c(4, 7), \text{Bcost}(3, 5) + c(5, 7), \\ &\quad \text{Bcost}(3, 6) + c(6, 7) \} \\ &= \min \{ \text{Bcost}(3, 4) + 1, \text{Bcost}(3, 5) + 6, \text{Bcost}(3, 6) + 6 \} \end{aligned}$$

$$\begin{aligned} \text{Bcost}(3, 4) &= \min \{ \text{Bcost}(2, 2) + c(2, 4), \text{Bcost}(2, 3) + c(3, 4) \} \\ &= \min \{ \text{Bcost}(2, 2) + 3, \text{Bcost}(2, 3) + 6 \} \end{aligned}$$

$$\text{Bcost}(2, 2) = \min \{ \text{Bcost}(1, 1) + c(1, 2) \} = \min \{ 0 + 5 \} = 5$$

$$\text{Bcost}(2, 3) = \min \{ \text{Bcost}(1, 1) + c(1, 3) \} = \min \{ 0 + 2 \} = 2$$

$$\text{Therefore, Bcost}(3, 4) = \min \{ 5 + 3, 2 + 6 \} = \min \{ 8, 8 \} = 8$$

$$\text{Bcost}(3, 5) = \min \{ \text{Bcost}(2, 3) + c(3, 5) \} = \min \{ 2 + 5 \} = 7$$

$$\begin{aligned} \text{Bcost}(3, 6) &= \min \{ \text{Bcost}(2, 2) + c(2, 6), \text{Bcost}(2, 3) + c(3, 6) \} \\ &= \min \{ 5 + 5, 2 + 8 \} = 10 \end{aligned}$$

$$\text{Therefore, Bcost}(4, 7) = \min \{ 8 + 1, 7 + 6, 10 + 6 \} = 9$$

$$\begin{aligned} \text{Bcost}(4, 8) &= \min \{ \text{Bcost}(3, 4) + c(4, 8), \text{Bcost}(3, 5) + c(5, 8), \\ &\quad \text{Bcost}(3, 6) + c(6, 8) \} \\ &= \min \{ 8 + 4, 7 + 2, 10 + 2 \} = 9 \end{aligned}$$

$$\text{Therefore, Bcost}(5, 9) = \min \{ 9 + 7, 9 + 3 \} = 12$$

All pairs shortest paths

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G . That is, for every pair of vertices (i, j) , we are to find a shortest path from i to j as well as one from j to i . These two paths are the same when G is undirected.

When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.

The all pairs shortest path problem is to determine a matrix A such that $A(i, j)$ is the length of a shortest path from i to j . The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires $O(n^2)$ time, the matrix A can be obtained in $O(n^3)$ time.

The dynamic programming solution, called Floyd's algorithm, runs in $O(n^3)$ time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

The shortest i to j path in G , $i \neq j$ originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j . If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j , respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let $A^k(i, j)$ represent the length of a shortest path from i to j going through no vertex of index greater than k , we obtain:

$$A^k(i, j) = \min_{1 \leq k \leq n} \{ \min \{ A^{k-1}(i, k) + A^{k-1}(k, j) \}, c(i, j) \}$$

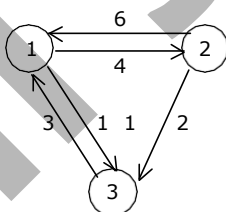
Algorithm All Paths (Cost, A, n)

```
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for  $1 \leq i \leq n$ .
{
    for i := 1 to n do
        for j := 1 to n do
            A [i, j] := cost [i, j];           // copy cost into A.
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to n do
                    A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
            }
}
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating $n+1$ matrices, each of size $n \times n$. Therefore, the algorithm has a complexity of $O(n^3)$.

Example 1:

Given a weighted digraph $G = (V, E)$ with weight. Determine the length of the shortest path between all pairs of vertices in G . Here we assume that there are no cycles with zero or negative cost.



$$\text{Cost adjacency matrix } (A^0) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

General formula: $\min_{1 \leq k \leq n} \{ A^{k-1}(i, k) + A^{k-1}(k, j) \}, c(i, j) \}$

Solve the problem for different values of $k = 1, 2$ and 3

Step 1: Solving the equation for, $k = 1$;

$$\begin{aligned}
A^1(1, 1) &= \min \{(A^0(1, 1) + A^0(1, 1)), c(1, 1)\} = \min \{0 + 0, 0\} = 0 \\
A^1(1, 2) &= \min \{(A^0(1, 1) + A^0(1, 2)), c(1, 2)\} = \min \{(0 + 4), 4\} = 4 \\
A^1(1, 3) &= \min \{(A^0(1, 1) + A^0(1, 3)), c(1, 3)\} = \min \{(0 + 11), 11\} = 11 \\
A^1(2, 1) &= \min \{(A^0(2, 1) + A^0(1, 1)), c(2, 1)\} = \min \{(6 + 0), 6\} = 6 \\
A^1(2, 2) &= \min \{(A^0(2, 1) + A^0(1, 2)), c(2, 2)\} = \min \{(6 + 4), 0\} = 0 \\
A^1(2, 3) &= \min \{(A^0(2, 1) + A^0(1, 3)), c(2, 3)\} = \min \{(6 + 11), 2\} = 2 \\
A^1(3, 1) &= \min \{(A^0(3, 1) + A^0(1, 1)), c(3, 1)\} = \min \{(3 + 0), 3\} = 3 \\
A^1(3, 2) &= \min \{(A^0(3, 1) + A^0(1, 2)), c(3, 2)\} = \min \{(3 + 4), \infty\} = 7 \\
A^1(3, 3) &= \min \{(A^0(3, 1) + A^0(1, 3)), c(3, 3)\} = \min \{(3 + 11), 0\} = 0
\end{aligned}$$

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 2: Solving the equation for, $K = 2$;

$$\begin{aligned}
A^2(1, 1) &= \min \{(A^1(1, 2) + A^1(2, 1), c(1, 1)\} = \min \{(4 + 6), 0\} = 0 \\
A^2(1, 2) &= \min \{(A^1(1, 2) + A^1(2, 2), c(1, 2)\} = \min \{(4 + 0), 4\} = 4 \\
A^2(1, 3) &= \min \{(A^1(1, 2) + A^1(2, 3), c(1, 3)\} = \min \{(4 + 2), 11\} = 6 \\
A^2(2, 1) &= \min \{(A^1(2, 2) + A^1(2, 1), c(2, 1)\} = \min \{(0 + 6), 6\} = 6 \\
A^2(2, 2) &= \min \{(A^1(2, 2) + A^1(2, 2), c(2, 2)\} = \min \{(0 + 0), 0\} = 0 \\
A^2(2, 3) &= \min \{(A^1(2, 2) + A^1(2, 3), c(2, 3)\} = \min \{(0 + 2), 2\} = 2 \\
A^2(3, 1) &= \min \{(A^1(3, 2) + A^1(2, 1), c(3, 1)\} = \min \{(7 + 6), 3\} = 3 \\
A^2(3, 2) &= \min \{(A^1(3, 2) + A^1(2, 2), c(3, 2)\} = \min \{(7 + 0), 7\} = 7 \\
A^2(3, 3) &= \min \{(A^1(3, 2) + A^1(2, 3), c(3, 3)\} = \min \{(7 + 2), 0\} = 0
\end{aligned}$$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 3: Solving the equation for, $k = 3$;

$$\begin{aligned}
A^3(1, 1) &= \min \{A^2(1, 3) + A^2(3, 1), c(1, 1)\} = \min \{(6 + 3), 0\} = 0 \\
A^3(1, 2) &= \min \{A^2(1, 3) + A^2(3, 2), c(1, 2)\} = \min \{(6 + 7), 4\} = 4 \\
A^3(1, 3) &= \min \{A^2(1, 3) + A^2(3, 3), c(1, 3)\} = \min \{(6 + 0), 6\} = 6 \\
A^3(2, 1) &= \min \{A^2(2, 3) + A^2(3, 1), c(2, 1)\} = \min \{(2 + 3), 6\} = 5 \\
A^3(2, 2) &= \min \{A^2(2, 3) + A^2(3, 2), c(2, 2)\} = \min \{(2 + 7), 0\} = 0 \\
A^3(2, 3) &= \min \{A^2(2, 3) + A^2(3, 3), c(2, 3)\} = \min \{(2 + 0), 2\} = 2 \\
A^3(3, 1) &= \min \{A^2(3, 3) + A^2(3, 1), c(3, 1)\} = \min \{(0 + 3), 3\} = 3 \\
A^3(3, 2) &= \min \{A^2(3, 3) + A^2(3, 2), c(3, 2)\} = \min \{(0 + 7), 7\} = 7
\end{aligned}$$

$$A^3(3, 3) = \min \{A^2(3, 3) + A^2(3, 3), c(3, 3)\} = \min \{(0 + 0), 0\} = 0$$

$$A^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

TRAVELLING SALESPERSON PROBLEM

Let $G = (V, E)$ be a directed graph with edge costs C_{ij} . The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \alpha$ if $\langle i, j \rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V . The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.

Let $g(i, S)$ be the length of shortest path starting at vertex i , going through all vertices in S , and terminating at vertex 1. The function $g(1, V - \{1\})$ is the length of an optimal salesperson tour. From the principal of optimality it follows that:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{--} \quad 1$$

Generalizing equation 1, we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(i, S - \{j\})\} \quad \text{--} \quad 2$$

The Equation can be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all choices of k .

Complexity Analysis:

For each value of $|S|$ there are $n - 1$ choices for i . The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$.

Hence, the total number of $g(i, S)$'s to be computed before computing $g(1, V - \{1\})$ is:

$$\sum_{k=0}^{n-1} (n-1) \binom{n-2}{k}$$

To calculate this sum, we use the binominal theorem:

$$(n-1) \left[\binom{n-2}{0} + \binom{n-2}{1} + \binom{n-2}{2} + \dots + \binom{n-2}{n-2} \right]$$

According to the binominal theorem:

$$\left[\binom{n-2}{0} + \binom{n-2}{1} + \binom{n-2}{2} + \dots + \binom{n-2}{n-2} \right] = 2^{n-2}$$

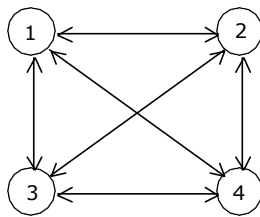
Therefore,

$$\sum_{k=0}^{n-1} (n-1) \binom{n-2}{k} = (n-1) 2^{n-2}$$

This is $\Phi(n 2^{n-2})$, so there are exponential number of calculate. Calculating one $g(i, S)$ require finding the minimum of at most n quantities. Therefore, the entire algorithm is $\Phi(n^2 2^{n-2})$. This is better than enumerating all $n!$ different tours to find the best one. So, we have traded on exponential growth for a much smaller exponential growth. The most serious drawback of this dynamic programming solution is the space needed, which is $O(n 2^n)$. This is too large even for modest values of n .

Example 1:

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix =

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad - \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(j, s - \{j\})\} \quad - \quad (2)$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \leq i \leq n$. So,

$$g(2, \Phi) = C_{21} = 5$$

$$g(3, \Phi) = C_{31} = 6$$

$$g(4, \Phi) = C_{41} = 8$$

Using equation - (2) we obtain:

$$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$$

$$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} \\ = \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$$

$$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$$

$$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$$

Therefore, $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

$g(3, \{2, 4\}) = \min \{(c_{32} + g(2, \{4\})), (c_{34} + g(4, \{2\}))\}$

$g(2, \{4\}) = \min \{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{41, 25\} = 25$

$g(4, \{2, 3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$g(2, \{3\}) = \min \{c_{23} + g(3, \Phi)\} = 9 + 6 = 15$

$g(3, \{2\}) = \min \{c_{32} + g(2, \Phi)\} = 13 + 5 = 18$

Therefore, $g(4, \{2, 3\}) = \min \{8 + 15, 9 + 18\} = \min \{23, 27\} = 23$

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$
 $= \min \{10 + 25, 15 + 25, 20 + 23\} = \min \{35, 40, 43\} = 35$

The optimal tour for the graph has length = 35

The optimal tour is: 1, 2, 4, 3, 1.

OPTIMAL BINARY SEARCH TREE

Let us assume that the given set of identifiers is $\{a_1, \dots, a_n\}$ with $a_1 < a_2 < \dots < a_n$. Let $p(i)$ be the probability with which we search for a_i . Let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$ (assume $a_0 = -\infty$ and $a_{n+1} = +\infty$). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.

In a binary search tree, the number of comparisons needed to access an element at depth ' d ' is $d + 1$, so if ' a_i ' is placed at depth ' d_i ', then we want to minimize:

$$\sum_{i=1}^n P_i (1 + d_i) .$$

Let $P(i)$ be the probability with which we shall be searching for ' a_i '. Let $Q(i)$ be the probability of an un-successful search. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

The expected cost contribution for the internal node for ' a_i ' is:

$$P(i) * \text{level}(a_i) .$$

Unsuccessful search terminate with $I = 0$ (i.e at an external node). Hence the cost contribution for this node is:

$$Q(i) * \text{level}((E_i) - 1)$$

The expected cost of binary search tree is:

$$\sum_{i=1}^n P(i) * \text{level}(a_i) + \sum_{i=0}^n Q(i) * \text{level}((E_i) - 1)$$

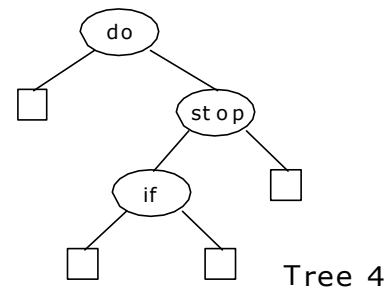
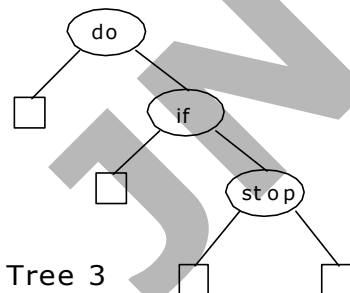
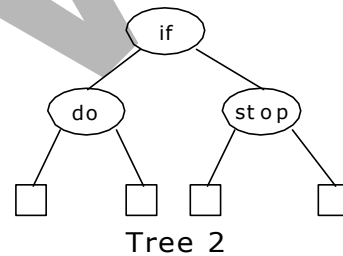
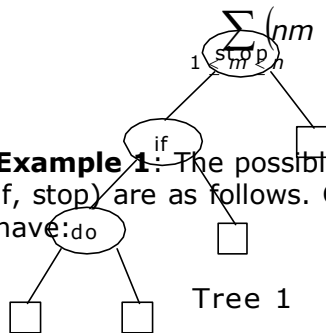
Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

The computation of each of these $c(i, j)$'s requires us to find the minimum of m quantities. Hence, each such $c(i, j)$ can be computed in time $O(m)$. The total time for all $c(i, j)$'s with $j - i = m$ is therefore $O(nm - m^2)$.

The total time to evaluate all the $c(i, j)$'s and $r(i, j)$'s is therefore:

$$\sum_{m=1}^n (nm - m^2) = O(n^3)$$

Example 1: The possible binary search trees for the identifier set $(a_1, a_2, a_3) = (\text{do}, \text{if}, \text{stop})$ are as follows. Given the equal probabilities $p(i) = Q(i) = 1/7$ for all i , we have:



$$\begin{aligned} \text{Cost (tree \# 1)} &= \left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3 \right) + \left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3 + \frac{1}{7} \times 3 \right) \\ &= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7} \end{aligned}$$

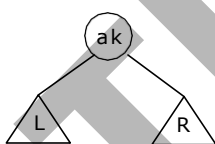
$$\begin{aligned} \text{Cost (tree \# 2)} &= \left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 2 \right) + \left(\frac{1}{7} \times 2 + \frac{1}{7} \times 2 + \frac{1}{7} \times 2 + \frac{1}{7} \times 2 \right) \\ &= \frac{1+2+2}{7} + \frac{2+2+2+2}{7} = \frac{5+8}{7} = \frac{13}{7} \end{aligned}$$

$$\begin{aligned}\text{Cost (tree \# 3)} &= \left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3 \right) + \left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3 + \frac{1}{7} \times 3 \right) \\ &= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}\end{aligned}$$

$$\begin{aligned}\text{Cost (tree \# 4)} &= \left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3 \right) + \left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3 + \frac{1}{7} \times 3 \right) \\ &= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}\end{aligned}$$

Huffman coding tree solved by a greedy algorithm has a limitation of having the data only at the leaves and it must not preserve the property that all nodes to the left of the root have keys, which are less etc. Construction of an optimal binary search tree is harder, because the data is not constrained to appear only at the leaves, and also because the tree must satisfy the binary search tree property and it must preserve the property that all nodes to the left of the root have keys, which are less.

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality. A possible approach to this is to make a decision as which of the a_i 's be arraigned to the root node at 'T'. If we choose ' a_k ' then is clear that the internal nodes for a_1, a_2, \dots, a_{k-1} as well as the external nodes for the classes E_0, E_1, \dots, E_{k-1} will lie in the left sub tree, L, of the root. The remaining nodes will be in the right subtree, R. The structure of an optimal binary search tree is:



$$\text{Cost (L)} = \sum_{i=1}^K P(i) * \text{level}(a_i) + \sum_{i=0}^K Q(i) * (\text{level}(E_i) - 1)$$

$$\text{Cost (R)} = \sum_{i=K}^n P(i) * \text{level}(a_i) + \sum_{i=K}^n Q(i) * (\text{level}(E_i) - 1)$$

The $C(i, J)$ can be computed as:

$$\begin{aligned}C(i, J) &= \min_{i < k \leq J} \{C(i, k-1) + C(k, J) + P(K) + w(i, K-1) + w(K, J)\} \\ &= \min_{i < k \leq J} \{C(i, K-1) + C(K, J)\} + w(i, J) \quad \text{--} \quad (1)\end{aligned}$$

$$\text{Where } W(i, J) = P(J) + Q(J) + w(i, J-1) \quad \text{--} \quad (2)$$

Initially $C(i, i) = 0$ and $w(i, i) = Q(i)$ for $0 \leq i \leq n$.

Equation (1) may be solved for $C(0, n)$ by first computing all $C(i, J)$ such that $J - i = 1$

Next, we can compute all $C(i, J)$ such that $J - i = 2$, Then all $C(i, J)$ with $J - i = 3$ and so on.

$C(i, j)$ is the cost of the optimal binary search tree ' T_{ij} ' during computation we record the root $R(i, j)$ of each tree ' T_{ij} '. Then an optimal binary search tree may be constructed from these $R(i, j)$. $R(i, j)$ is the value of 'K' that minimizes equation (1).

We solve the problem by knowing $W(i, i+1)$, $C(i, i+1)$ and $R(i, i+1)$, $0 \leq i < 4$; Knowing $W(i, i+2)$, $C(i, i+2)$ and $R(i, i+2)$, $0 \leq i < 3$ and repeating until $W(0, n)$, $C(0, n)$ and $R(0, n)$ are obtained.

The results are tabulated to recover the actual tree.

Example 1:

Let $n = 4$, and $(a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{need}, \text{while})$ Let $P(1:4) = (3, 3, 1, 1)$ and $Q(0:4) = (2, 3, 1, 1, 1)$

Solution:

Table for recording $W(i, j)$, $C(i, j)$ and $R(i, j)$:

Column Row	0	1	2	3	4
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	14, 25, 2	11, 19, 2			
4	16, 32, 2				

This computation is carried out row-wise from row 0 to row 4. Initially, $W(i, i) = Q(i)$ and $C(i, i) = 0$ and $R(i, i) = 0$, $0 \leq i < 4$.

Solving for $C(0, n)$:

First, computing all $C(i, j)$ such that $j - i = 1$; $j = i + 1$ and as $0 \leq i < 4$; $i = 0, 1, 2$ and 3; $i < k \leq j$. Start with $i = 0$; so $j = 1$; as $i < k \leq j$, so the possible value for $k = 1$

$$W(0, 1) = P(1) + Q(1) + W(0, 0) = 3 + 3 + 2 = 8$$

$$C(0, 1) = W(0, 1) + \min \{C(0, 0) + C(1, 1)\} = 8$$

$$R(0, 1) = 1 \text{ (value of 'K' that is minimum in the above equation).}$$

Next with $i = 1$; so $j = 2$; as $i < k \leq j$, so the possible value for $k = 2$

$$W(1, 2) = P(2) + Q(2) + W(1, 1) = 3 + 1 + 3 = 7$$

$$C(1, 2) = W(1, 2) + \min \{C(1, 1) + C(2, 2)\} = 7$$

$$R(1, 2) = 2$$

Next with $i = 2$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 3$

$$\begin{aligned}
W(2, 3) &= P(3) + Q(3) + W(2, 2) = 1 + 1 + 1 = 3 \\
C(2, 3) &= W(2, 3) + \min \{C(2, 2) + C(3, 3)\} = 3 + [(0 + 0)] = 3 \\
R(2, 3) &= 3
\end{aligned}$$

Next with $i = 3$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 4$

$$\begin{aligned}
W(3, 4) &= P(4) + Q(4) + W(3, 3) = 1 + 1 + 1 = 3 \\
C(3, 4) &= W(3, 4) + \min \{[C(3, 3) + C(4, 4)]\} = 3 + [(0 + 0)] = 3 \\
R(3, 4) &= 4
\end{aligned}$$

Second, Computing all $C(i, j)$ such that $j - i = 2$; $j = i + 2$ and as $0 \leq i < 3$; $i = 0, 1, 2$; $i < k \leq J$. Start with $i = 0$; so $j = 2$; as $i < k \leq J$, so the possible values for $k = 1$ and 2 .

$$\begin{aligned}
W(0, 2) &= P(2) + Q(2) + W(0, 1) = 3 + 1 + 8 = 12 \\
C(0, 2) &= W(0, 2) + \min \{(C(0, 0) + C(1, 2)), (C(0, 1) + C(2, 2))\} \\
&= 12 + \min \{(0 + 7, 8 + 0)\} = 19 \\
R(0, 2) &= 1
\end{aligned}$$

Next, with $i = 1$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 2$ and 3 .

$$\begin{aligned}
W(1, 3) &= P(3) + Q(3) + W(1, 2) = 1 + 1 + 7 = 9 \\
C(1, 3) &= W(1, 3) + \min \{[C(1, 1) + C(2, 3)], [C(1, 2) + C(3, 3)]\} \\
&= W(1, 3) + \min \{(0 + 3), (7 + 0)\} = 9 + 3 = 12 \\
R(1, 3) &= 2
\end{aligned}$$

Next, with $i = 2$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 3$ and 4 .

$$\begin{aligned}
W(2, 4) &= P(4) + Q(4) + W(2, 3) = 1 + 1 + 3 = 5 \\
C(2, 4) &= W(2, 4) + \min \{[C(2, 2) + C(3, 4)], [C(2, 3) + C(4, 4)]\} \\
&= 5 + \min \{(0 + 3), (3 + 0)\} = 5 + 3 = 8 \\
R(2, 4) &= 3
\end{aligned}$$

Third, Computing all $C(i, j)$ such that $J - i = 3$; $j = i + 3$ and as $0 \leq i < 2$; $i = 0, 1$; $i < k \leq J$. Start with $i = 0$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 1, 2$ and 3 .

$$\begin{aligned}
W(0, 3) &= P(3) + Q(3) + W(0, 2) = 1 + 1 + 12 = 14 \\
C(0, 3) &= W(0, 3) + \min \{[C(0, 0) + C(1, 3)], [C(0, 1) + C(2, 3)], \\
&\quad [C(0, 2) + C(3, 3)]\} \\
&= 14 + \min \{(0 + 12), (8 + 3), (19 + 0)\} = 14 + 11 = 25 \\
R(0, 3) &= 2
\end{aligned}$$

Start with $i = 1$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and 4 .

$$\begin{aligned}
W(1, 4) &= P(4) + Q(4) + W(1, 3) = 1 + 1 + 9 = 11 \\
C(1, 4) &= W(1, 4) + \min \{[C(1, 1) + C(2, 4)], [C(1, 2) + C(3, 4)], \\
&\quad [C(1, 3) + C(4, 4)]\} \\
&= 11 + \min \{(0 + 8), (7 + 3), (12 + 0)\} = 11 + 8 = 19 \\
R(1, 4) &= 2
\end{aligned}$$

Fourth, Computing all $C(i, j)$ such that $j - i = 4$; $j = i + 4$ and as $0 \leq i < 1$; $i = 0$; $i < k \leq J$.

Start with $i = 0$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and 4 .

$$\begin{aligned}
 W(0, 4) &= P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16 \\
 C(0, 4) &= W(0, 4) + \min \{ [C(0, 0) + C(1, 4)], [C(0, 1) + C(2, 4)], \\
 &\quad [C(0, 2) + C(3, 4)], [C(0, 3) + C(4, 4)] \} \\
 &= 16 + \min [0 + 19, 8 + 8, 19 + 3, 25 + 0] = 16 + 16 = 32 \\
 R(0, 4) &= 2
 \end{aligned}$$

From the table we see that $C(0, 4) = 32$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4) . The root of the tree 'T₀₄' is 'a₂'.

Hence the left sub tree is 'T₀₁' and right sub tree is T₂₄. The root of 'T₀₁' is 'a₁' and the root of 'T₂₄' is a₃.

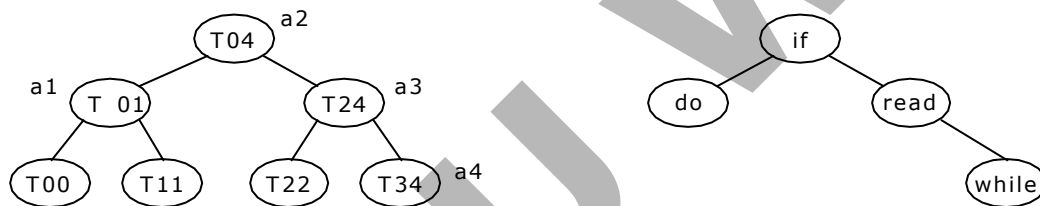
The left and right sub trees for 'T₀₁' are 'T₀₀' and 'T₁₁' respectively. The root of T₀₁ is 'a₁'

The left and right sub trees for T₂₄ are T₂₂ and T₃₄ respectively.

The root of T₂₄ is 'a₃'.

The root of T₂₂ is null

The root of T₃₄ is a₄.



Example 2:

Consider four elements a_1, a_2, a_3 and a_4 with $Q_0 = 1/8, Q_1 = 3/16, Q_2 = Q_3 = Q_4 = 1/16$ and $p_1 = 1/4, p_2 = 1/8, p_3 = p_4 = 1/16$. Construct an optimal binary search tree. Solving for $C(0, n)$:

First, computing all $C(i, j)$ such that $j - i = 1; j = i + 1$ and as $0 \leq i < 4; i = 0, 1, 2$ and $3; i < k \leq j$. Start with $i = 0$; so $j = 1$; as $i < k \leq j$, so the possible value for $k = 1$

$$\begin{aligned}
 W(0, 1) &= P(1) + Q(1) + W(0, 0) = 4 + 3 + 2 = 9 \\
 C(0, 1) &= W(0, 1) + \min \{ C(0, 0) + C(1, 1) \} = 9 + [(0 + 0)] = 9 \\
 R(0, 1) &= 1 \text{ (value of 'K' that is minimum in the above equation).}
 \end{aligned}$$

Next with $i = 1$; so $j = 2$; as $i < k \leq j$, so the possible value for $k = 2$

$$\begin{aligned}
 W(1, 2) &= P(2) + Q(2) + W(1, 1) = 2 + 1 + 3 = 6 \\
 C(1, 2) &= W(1, 2) + \min \{ C(1, 1) + C(2, 2) \} = 6 + [(0 + 0)] = 6 \\
 R(1, 2) &= 2
 \end{aligned}$$

Next with $i = 2$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 3$

$$\begin{aligned}
 W(2, 3) &= P(3) + Q(3) + W(2, 2) = 1 + 1 + 1 = 3 \\
 C(2, 3) &= W(2, 3) + \min \{ C(2, 2) + C(3, 3) \} = 3 + [(0 + 0)] = 3
 \end{aligned}$$

$$R(2, 3) = 3$$

Next with $i = 3$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 4$

$$\begin{aligned} W(3, 4) &= P(4) + Q(4) + W(3, 3) = 1 + 1 + 1 = 3 \\ C(3, 4) &= W(3, 4) + \min \{[C(3, 3) + C(4, 4)]\} = 3 + [(0 + 0)] = 3 \\ R(3, 4) &= 4 \end{aligned}$$

Second, Computing all $C(i, j)$ such that $j - i = 2$; $j = i + 2$ and as $0 \leq i < 3$; $i = 0, 1, 2$; $i < k \leq j$

Start with $i = 0$; so $j = 2$; as $i < k \leq j$, so the possible values for $k = 1$ and 2 .

$$\begin{aligned} W(0, 2) &= P(2) + Q(2) + W(0, 1) = 2 + 1 + 9 = 12 \\ C(0, 2) &= W(0, 2) + \min \{(C(0, 0) + C(1, 2)), (C(0, 1) + C(2, 2))\} \\ &= 12 + \min \{(0 + 6, 9 + 0)\} = 12 + 6 = 18 \\ R(0, 2) &= 1 \end{aligned}$$

Next, with $i = 1$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 2$ and 3 .

$$\begin{aligned} W(1, 3) &= P(3) + Q(3) + W(1, 2) = 1 + 1 + 6 = 8 \\ C(1, 3) &= W(1, 3) + \min \{[C(1, 1) + C(2, 3)], [C(1, 2) + C(3, 3)]\} \\ &= W(1, 3) + \min \{(0 + 3), (6 + 0)\} = 8 + 3 = 11 \\ R(1, 3) &= 2 \end{aligned}$$

Next, with $i = 2$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 3$ and 4 .

$$\begin{aligned} W(2, 4) &= P(4) + Q(4) + W(2, 3) = 1 + 1 + 3 = 5 \\ C(2, 4) &= W(2, 4) + \min \{[C(2, 2) + C(3, 4)], [C(2, 3) + C(4, 4)]\} \\ &= 5 + \min \{(0 + 3), (3 + 0)\} = 5 + 3 = 8 \\ R(2, 4) &= 3 \end{aligned}$$

Third, Computing all $C(i, j)$ such that $j - i = 3$; $j = i + 3$ and as $0 \leq i < 2$; $i = 0, 1$; $i < k \leq j$. Start with $i = 0$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 1, 2$ and 3 .

$$\begin{aligned} W(0, 3) &= P(3) + Q(3) + W(0, 2) = 1 + 1 + 12 = 14 \\ C(0, 3) &= W(0, 3) + \min \{[C(0, 0) + C(1, 3)], [C(0, 1) + C(2, 3)], \\ &\quad [C(0, 2) + C(3, 3)]\} \\ &= 14 + \min \{(0 + 11), (9 + 3), (18 + 0)\} = 14 + 11 = 25 \\ R(0, 3) &= 1 \end{aligned}$$

Start with $i = 1$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and 4 .

$$\begin{aligned} W(1, 4) &= P(4) + Q(4) + W(1, 3) = 1 + 1 + 8 = 10 \\ C(1, 4) &= W(1, 4) + \min \{[C(1, 1) + C(2, 4)], [C(1, 2) + C(3, 4)], \\ &\quad [C(1, 3) + C(4, 4)]\} \\ &= 10 + \min \{(0 + 8), (6 + 3), (11 + 0)\} = 10 + 8 = 18 \\ R(1, 4) &= 2 \end{aligned}$$

Fourth, Computing all $C(i, j)$ such that $j - i = 4$; $j = i + 4$ and as $0 \leq i < 1$; $i = 0$; $i < k \leq j$. Start with $i = 0$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and 4 .

$$\begin{aligned} W(0, 4) &= P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16 \\ C(0, 4) &= W(0, 4) + \min \{[C(0, 0) + C(1, 4)], [C(0, 1) + C(2, 4)], \\ &\quad [C(0, 2) + C(3, 4)], [C(0, 3) + C(4, 4)]\} \end{aligned}$$

$$= 16 + \min [0 + 18, 9 + 8, 18 + 3, 25 + 0] = 16 + 17 = 33$$

$$R(0, 4) = 2$$

Table for recording $W(i, j)$, $C(i, j)$ and $R(i, j)$

Column Row	0	1	2	3	4
0	2, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0
1	9, 9, 1	6, 6, 2	3, 3, 3	3, 3, 4	
2	12, 18, 1	8, 11, 2	5, 8, 3		
3	14, 25, 2	11, 18, 2			
4	16, 33, 2				

From the table we see that $C(0, 4) = 33$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4)

The root of the tree ' T_{04} ' is ' a_2 '.

Hence the left sub tree is ' T_{01} ' and right sub tree is T_{24} . The root of ' T_{01} ' is ' a_1 ' and the root of ' T_{24} ' is a_3 .

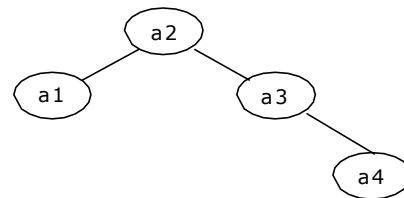
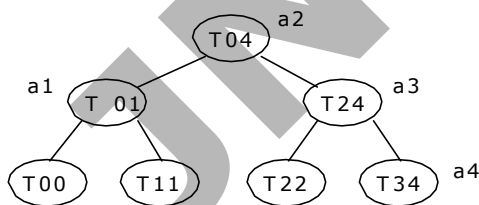
The left and right sub trees for ' T_{01} ' are ' T_{00} ' and ' T_{11} ' respectively. The root of T_{01} is ' a_1 '

The left and right sub trees for T_{24} are T_{22} and T_{34} respectively.

The root of T_{24} is ' a_3 '.

The root of T_{22} is null.

The root of T_{34} is a_4 .



Example 3:

<u>WORD</u>	<u>PROBABILITY</u>
A	4
B	2
C	1
D	3
E	5
F	2
G	1

and all other elements have zero probability.

Solving c(0,n):

First computing all $c(i, j)$ such that $j - i = 1$; $j = i + 1$ and as $0 \leq i < 7$; $i = 0, 1, 2, 3, 4, 5$ and 6 ; $i < k \leq j$. Start with $i = 0$; so $j = 1$; as $i < k \leq j$, so the possible value for $k = 1$

$$\begin{aligned} W(0, 1) &= P(1) + Q(1) + W(0, 0) = 4 + 0 + 0 = 4 \\ C(0, 1) &= W(0, 1) + \min \{C(0, 0) + C(1, 1)\} = 4 + [(0 + 0)] = 4 \\ R(0, 1) &= 1 \end{aligned}$$

next with $i = 1$; so $j = 2$; as $i < k \leq j$, so the possible value for $k = 2$

$$\begin{aligned} W(1, 2) &= P(2) + Q(2) + W(1, 1) = 2 + 0 + 0 = 2 \\ C(1, 2) &= W(1, 2) + \min \{C(1, 1) + C(2, 2)\} = 2 + [(0 + 0)] = 2 \\ R(1, 2) &= 2 \end{aligned}$$

next with $i = 2$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 3$

$$\begin{aligned} W(2, 3) &= P(3) + Q(3) + W(2, 2) = 1 + 0 + 0 = 1 \\ C(2, 3) &= W(2, 3) + \min \{C(2, 2) + C(3, 3)\} = 1 + [(0 + 0)] = 1 \\ R(2, 3) &= 3 \end{aligned}$$

next with $i = 3$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 4$

$$\begin{aligned} W(3, 4) &= P(4) + Q(4) + W(3, 3) = 3 + 0 + 0 = 3 \\ C(3, 4) &= W(3, 4) + \min \{C(3, 4) + C(4, 4)\} = 3 + [(0 + 0)] = 3 \\ R(3, 4) &= 4 \end{aligned}$$

next with $i = 4$; so $j = 5$; as $i < k \leq j$, so the possible value for $k = 5$

$$\begin{aligned} W(4, 5) &= P(5) + Q(5) + W(4, 4) = 5 + 0 + 0 = 5 \\ C(4, 5) &= W(4, 5) + \min \{C(4, 4) + C(5, 5)\} = 5 + [(0 + 0)] = 5 \\ R(4, 5) &= 5 \end{aligned}$$

next with $i = 5$; so $j = 6$; as $i < k \leq j$, so the possible value for $k = 6$

$$\begin{aligned} W(5, 6) &= P(6) + Q(6) + W(5, 5) = 2 + 0 + 0 = 2 \\ C(5, 6) &= W(5, 6) + \min \{C(5, 5) + C(6, 6)\} = 2 + [(0 + 0)] = 2 \\ R(5, 6) &= 6 \end{aligned}$$

next with $i = 6$; so $j = 7$; as $i < k \leq j$, so the possible value for $k = 7$

$$\begin{aligned} W(6, 7) &= P(7) + Q(7) + W(6, 6) = 1 + 0 + 0 = 1 \\ C(6, 7) &= W(6, 7) + \min \{C(6, 6) + C(7, 7)\} = 1 + [(0 + 0)] = 1 \\ R(6, 7) &= 7 \end{aligned}$$

Second, computing all $c(i, j)$ such that $j - i = 2$; $j = i + 2$ and as $0 \leq i < 6$; $i = 0, 1, 2, 3, 4$ and 5 ; $i < k \leq j$; Start with $i = 0$; so $j = 2$; as $i < k \leq j$, so the possible values for $k = 1$ and 2 .

$$\begin{aligned} W(0, 2) &= P(2) + Q(2) + W(0, 1) = 2 + 0 + 4 = 6 \\ C(0, 2) &= W(0, 2) + \min \{C(0, 0) + C(1, 2), C(0, 1) + C(2, 2)\} \\ &= 6 + \min\{0 + 2, 4 + 0\} = 8 \\ R(0, 2) &= 1 \end{aligned}$$

next with $i = 1$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 2$ and 3 .

$$\begin{aligned} W(1, 3) &= P(3) + Q(3) + W(1, 2) = 1 + 0 + 2 = 3 \\ C(1, 3) &= W(1, 3) + \min \{C(1, 1) + C(2, 3), C(1, 2) + C(3, 3)\} \\ &= 3 + \min\{0 + 1, 2 + 0\} = 4 \\ R(1, 3) &= 2 \end{aligned}$$

next with $i = 2$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 3$ and 4 .

$$\begin{aligned} W(2, 4) &= P(4) + Q(4) + W(2, 3) = 3 + 0 + 1 = 4 \\ C(2, 4) &= W(2, 4) + \min \{C(2, 2) + C(3, 4), C(2, 3) + C(4, 4)\} \\ &= 4 + \min\{0 + 3, 1 + 0\} = 5 \\ R(2, 4) &= 4 \end{aligned}$$

next with $i = 3$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 4$ and 5 .

$$\begin{aligned} W(3, 5) &= P(5) + Q(5) + W(3, 4) = 5 + 0 + 3 = 8 \\ C(3, 5) &= W(3, 5) + \min \{C(3, 3) + C(4, 5), C(3, 4) + C(5, 5)\} \\ &= 8 + \min\{0 + 5, 3 + 0\} = 11 \\ R(3, 5) &= 5 \end{aligned}$$

next with $i = 4$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 5$ and 6 .

$$\begin{aligned} W(4, 6) &= P(6) + Q(6) + W(4, 5) = 2 + 0 + 5 = 7 \\ C(4, 6) &= W(4, 6) + \min \{C(4, 4) + C(5, 6), C(4, 5) + C(6, 6)\} \\ &= 7 + \min\{0 + 2, 5 + 0\} = 9 \\ R(4, 6) &= 5 \end{aligned}$$

next with $i = 5$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 6$ and 7 .

$$\begin{aligned} W(5, 7) &= P(7) + Q(7) + W(5, 6) = 1 + 0 + 2 = 3 \\ C(5, 7) &= W(5, 7) + \min \{C(5, 5) + C(6, 7), C(5, 6) + C(7, 7)\} \\ &= 3 + \min\{0 + 1, 2 + 0\} = 4 \\ R(5, 7) &= 6 \end{aligned}$$

Third, computing all $c(i, j)$ such that $j - i = 3$; $j = i + 3$ and as $0 \leq i < 5$; $i = 0, 1, 2, 3, 4$ and $i < k \leq j$.

Start with $i = 0$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 1, 2$ and 3 .

$$\begin{aligned} W(0, 3) &= P(3) + Q(3) + W(0, 2) = 1 + 0 + 6 = 7 \\ C(0, 3) &= W(0, 3) + \min \{C(0, 0) + C(1, 3), C(0, 1) + C(2, 3), C(0, 2) + C(3, 3)\} \\ &= 7 + \min\{0 + 4, 4 + 1, 8 + 0\} = 7 \\ R(0, 3) &= 1 \end{aligned}$$

next with $i = 1$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and 4 .

$$\begin{aligned} W(1, 4) &= P(4) + Q(4) + W(1, 3) = 3 + 0 + 3 = 6 \\ C(1, 4) &= W(1, 4) + \min \{C(1, 1) + C(2, 4), C(1, 2) + C(3, 4), C(1, 3) + C(4, 4)\} \\ &= 6 + \min\{0 + 5, 2 + 3, 4 + 0\} = 10 \\ R(1, 4) &= 4 \end{aligned}$$

next with $i = 2$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 3, 4$ and 5 .

$$\begin{aligned} W(2, 5) &= P(5) + Q(5) + W(2, 4) = 5 + 0 + 4 = 9 \\ C(2, 5) &= W(2, 5) + \min \{C(2, 2) + C(3, 5), C(2, 3) + C(4, 5), C(2, 4) + C(5, 5)\} \\ &= 9 + \min\{0 + 11, 1 + 5, 5 + 0\} = 14 \\ R(2, 5) &= 5 \end{aligned}$$

next with $i = 3$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 4, 5$ and 6 .

$$\begin{aligned} W(3, 6) &= P(6) + Q(6) + W(3, 5) = 2 + 0 + 8 = 10 \\ C(3, 6) &= W(3, 6) + \min \{C(3, 3) + C(4, 6), C(3, 4) + C(5, 6), C(3, 5) + C(6, 6)\} \\ &= 10 + \min\{0 + 9, 3 + 2, 11 + 0\} = 15 \\ R(3, 6) &= 5 \end{aligned}$$

next with $i = 4$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 5, 6$ and 7 .

$$\begin{aligned} W(4, 7) &= P(7) + Q(7) + W(4, 6) = 1 + 0 + 7 = 8 \\ C(4, 7) &= W(4, 7) + \min \{C(4, 4) + C(5, 7), C(4, 5) + C(6, 7), C(4, 6) + C(7, 7)\} \\ &= 8 + \min\{0 + 4, 5 + 1, 9 + 0\} = 12 \\ R(4, 7) &= 5 \end{aligned}$$

Fourth, computing all $c(i, j)$ such that $j - i = 4$; $j = i + 4$ and as $0 \leq i < 4$; $i = 0, 1, 2, 3$ for $i < k \leq j$. Start with $i = 0$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and 4 .

$$\begin{aligned} W(0, 4) &= P(4) + Q(4) + W(0, 3) = 3 + 0 + 7 = 10 \\ C(0, 4) &= W(0, 4) + \min \{C(0, 0) + C(1, 4), C(0, 1) + C(2, 4), C(0, 2) + C(3, 4), \\ &\quad C(0, 3) + C(4, 4)\} \\ &= 10 + \min\{0 + 10, 4 + 5, 8 + 3, 11 + 0\} = 19 \\ R(0, 4) &= 2 \end{aligned}$$

next with $i = 1$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 2, 3, 4$ and 5 .

$$\begin{aligned} W(1, 5) &= P(5) + Q(5) + W(1, 4) = 5 + 0 + 6 = 11 \\ C(1, 5) &= W(1, 5) + \min \{C(1, 1) + C(2, 5), C(1, 2) + C(3, 5), C(1, 3) + C(4, 5), \\ &\quad C(1, 4) + C(5, 5)\} \\ &= 11 + \min\{0 + 14, 2 + 11, 4 + 5, 10 + 0\} = 20 \\ R(1, 5) &= 4 \end{aligned}$$

next with $i = 2$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 3, 4, 5$ and 6 .

$$\begin{aligned} W(2, 6) &= P(6) + Q(6) + W(2, 5) = 2 + 0 + 9 = 11 \\ C(2, 6) &= W(2, 6) + \min \{C(2, 2) + C(3, 6), C(2, 3) + C(4, 6), C(2, 4) + C(5, 6), \\ &\quad C(2, 5) + C(6, 6)\} = 11 + \min\{0 + 15, 1 + 9, 5 + 2, 14 + 0\} = 18 \\ R(2, 6) &= 5 \end{aligned}$$

next with $i = 3$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 4, 5, 6$ and 7 .

$$\begin{aligned} W(3, 7) &= P(7) + Q(7) + W(3, 6) = 1 + 0 + 11 = 12 \\ C(3, 7) &= W(3, 7) + \min \{C(3, 3) + C(4, 7), C(3, 4) + C(5, 7), C(3, 5) + C(6, 7), \\ &\quad C(3, 6) + C(7, 7)\} = 12 + \min\{0 + 12, 3 + 4, 11 + 1, 15 + 0\} = 19 \\ R(3, 7) &= 5 \end{aligned}$$

Fifth, computing all $c(i, j)$ such that $j - i = 5$; $j = i + 5$ and as $0 \leq i < 3$; $i = 0, 1, 2$, $i < k \leq j$. Start with $i = 0$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3, 4$ and 5 .

$$\begin{aligned} W(0, 5) &= P(5) + Q(5) + W(0, 4) = 5 + 0 + 10 = 15 \\ C(0, 5) &= W(0, 5) + \min \{C(0, 0) + C(1, 5), C(0, 1) + C(2, 5), C(0, 2) + C(3, 5), \\ &\quad C(0, 3) + C(4, 5), C(0, 4) + C(5, 5)\} \\ &= 10 + \min\{0 + 20, 4 + 14, 8 + 11, 19 + 0\} = 28 \end{aligned}$$

$$R(0, 5) = 2$$

next with $i = 1$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 2, 3, 4, 5$ & 6 .

$$\begin{aligned} W(1, 6) &= P(6) + Q(6) + W(1, 5) = 2 + 0 + 11 = 13 \\ C(1, 6) &= W(1, 6) + \min \{C(1, 1) + C(2, 6), C(1, 2) + C(3, 6), C(1, 3) + C(4, 6), \\ &\quad C(1, 4) + C(5, 6), C(1, 5) + C(6, 6)\} \\ &= 13 + \min\{0 + 18, 2 + 15, 4 + 9, 10 + 2, 20 + 0\} = 25 \\ R(1, 6) &= 5 \end{aligned}$$

next with $i = 2$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 3, 4, 5, 6$ and 7 .

$$\begin{aligned} W(2, 7) &= P(7) + Q(7) + W(2, 6) = 1 + 0 + 11 = 12 \\ C(2, 7) &= W(2, 7) + \min \{C(2, 2) + C(3, 7), C(2, 3) + C(4, 7), C(2, 4) + C(5, 7), \\ &\quad C(2, 5) + C(6, 7), C(2, 6) + C(7, 7)\} \\ &= 12 + \min\{0 + 18, 1 + 12, 5 + 4, 14 + 1, 18 + 0\} = 21 \\ R(2, 7) &= 5 \end{aligned}$$

Sixth, computing all $c(i, j)$ such that $j - i = 6$; $j = i + 6$ and as $0 \leq i < 2$; $i = 0, 1$ $i < k \leq j$. Start with $i = 0$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3, 4, 5$ & 6 .

$$\begin{aligned} W(0, 6) &= P(6) + Q(6) + W(0, 5) = 2 + 0 + 15 = 17 \\ C(0, 6) &= W(0, 6) + \min \{C(0, 0) + C(1, 6), C(0, 1) + C(2, 6), C(0, 2) + C(3, 6), \\ &\quad C(0, 3) + C(4, 6), C(0, 4) + C(5, 6), C(0, 5) + C(6, 6)\} \\ &= 17 + \min\{0 + 25, 4 + 18, 8 + 15, 19 + 2, 31 + 0\} = 37 \\ R(0, 6) &= 4 \end{aligned}$$

next with $i = 1$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 2, 3, 4, 5, 6$ and 7 .

$$\begin{aligned} W(1, 7) &= P(7) + Q(7) + W(1, 6) = 1 + 0 + 13 = 14 \\ C(1, 7) &= W(1, 7) + \min \{C(1, 1) + C(2, 7), C(1, 2) + C(3, 7), C(1, 3) + C(4, 7), \\ &\quad C(1, 4) + C(5, 7), C(1, 5) + C(6, 7), C(1, 6) + C(7, 7)\} \\ &= 14 + \min\{0 + 21, 2 + 18, 4 + 12, 10 + 4, 20 + 1, 21 + 0\} = 28 \\ R(1, 7) &= 5 \end{aligned}$$

Seventh, computing all $c(i, j)$ such that $j - i = 7$; $j = i + 7$ and as $0 \leq i < 1$; $i = 0$ $i < k \leq j$. Start with $i = 0$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3, 4, 5, 6$ and 7 .

$$\begin{aligned} W(0, 7) &= P(7) + Q(7) + W(0, 6) = 1 + 0 + 17 = 18 \\ C(0, 7) &= W(0, 7) + \min \{C(0, 0) + C(1, 7), C(0, 1) + C(2, 7), C(0, 2) + C(3, 7), \\ &\quad C(0, 3) + C(4, 7), C(0, 4) + C(5, 6), C(0, 5) + C(6, 7), C(0, 6) + C(7, 7)\} \\ &= 18 + \min\{0 + 28, 4 + 21, 8 + 18, 19 + 4, 31 + 1, 37 + 0\} = 41 \\ R(0, 7) &= 4 \end{aligned}$$

0/1 – KNAPSACK

We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value V_i . The knapsack can carry a weight not exceeding W . Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x_1, x_2, \dots, x_n . A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that

decisions on the x_i are made in the order x_n, x_{n-1}, \dots, x_1 . Following a decision on x_n , we may be in one of two possible states: the capacity remaining in $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions x_{n-1}, \dots, x_1 must be optimal with respect to the problem state resulting from the decision on x_n . Otherwise, x_n, \dots, x_1 will not be optimal. Hence, the principal of optimality holds.

$$F_n(m) = \max \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \quad \text{--} \quad 1$$

For arbitrary $f_i(y)$, $i > 0$, this equation generalizes to:

$$F_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad \text{--} \quad 2$$

Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y) = 0$ for all y and $f_i(y) = -\infty$, $y < 0$. Then f_1, f_2, \dots, f_n can be successively computed using equation-2.

When the w_i 's are integer, we need to compute $f_i(y)$ for integer y , $0 \leq y \leq m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly. Since each f_i can be computed from f_{i-1} in $\Theta(m)$ time, it takes $\Theta(mn)$ time to compute f_n . When the w_i 's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i 's are integer, the explicit $\Theta(mn)$ computation of f_n may not be the most efficient computation. So, we explore **an alternative method for both cases**.

The $f_i(y)$ is an ascending step function; i.e., there are a finite number of y 's, $0 = y_1 < y_2 < \dots < y_k$, such that $f_i(y_1) < f_i(y_2) < \dots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f_i(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y \leq y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, W) , where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S^i by first computing:

$$S^i_1 = \{(P, W) \mid (P - p_i, W - w_i) \in S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S^i_1 together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j) .

Example 1:

Consider the knapsack instance $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$ and $M = 6$.

Solution:

Initially, $f_0(x) = 0$, for all x and $f_i(x) = -\infty$ if $x < 0$.

$$F_n(M) = \max \{f_{n-1}(M), f_{n-1}(M - w_n) + p_n\}$$

$$F_3(6) = \max \{f_2(6), f_2(6 - 4) + 5\} = \max \{f_2(6), f_2(2) + 5\}$$

$$F_2(6) = \max \{f_1(6), f_1(6 - 3) + 2\} = \max \{f_1(6), f_1(3) + 2\}$$

$$F_1(6) = \max \{f_0(6), f_0(6-2) + 1\} = \max \{0, 0 + 1\} = 1$$

$$F_1(3) = \max \{f_0(3), f_0(3-2) + 1\} = \max \{0, 0 + 1\} = 1$$

$$\text{Therefore, } F_2(6) = \max \{1, 1 + 2\} = 3$$

$$F_2(2) = \max \{f_1(2), f_1(2-3) + 2\} = \max \{f_1(2), -\infty + 2\}$$

$$F_1(2) = \max \{f_0(2), f_0(2-2) + 1\} = \max \{0, 0 + 1\} = 1$$

$$F_2(2) = \max \{1, -\infty + 2\} = 1$$

$$\text{Finally, } f_3(6) = \max \{3, 1 + 5\} = 6$$

Other Solution:

For the given data we have:

$$S^0 = \{(0, 0)\}; \quad S^0_1 = \{(1, 2)\}$$

$$S^1 = (S^0 \cup S^0_1) = \{(0, 0), (1, 2)\}$$

$$\begin{array}{ll} X - 2 = 0 \Rightarrow x = 2. & y - 3 = 0 \Rightarrow y = 3 \\ X - 2 = 1 \Rightarrow x = 3. & y - 3 = 2 \Rightarrow y = 5 \end{array}$$

$$S^1_1 = \{(2, 3), (3, 5)\}$$

$$S^2 = (S^1 \cup S^1_1) = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$$

$$\begin{array}{ll} X - 5 = 0 \Rightarrow x = 5. & y - 4 = 0 \Rightarrow y = 4 \\ X - 5 = 1 \Rightarrow x = 6. & y - 4 = 2 \Rightarrow y = 6 \\ X - 5 = 2 \Rightarrow x = 7. & y - 4 = 3 \Rightarrow y = 7 \\ X - 5 = 3 \Rightarrow x = 8. & y - 4 = 5 \Rightarrow y = 9 \end{array}$$

$$S^2_1 = \{(5, 4), (6, 6), (7, 7), (8, 9)\}$$

$$S^3 = (S^2 \cup S^2_1) = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)\}$$

By applying Dominance rule,

$$S^3 = (S^2 \cup S^2_1) = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)\}$$

From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$

Reliability Design

The problem is to design a system that is composed of several devices connected in series. Let r_i be the reliability of device D_i (that is r_i is the probability that device i will function properly) then the reliability of the entire system is $\prod r_i$. Even if the individual devices are very reliable (the r_i 's are very close to one), the reliability of the system may not be very good. For example, if $n = 10$ and $r_i = 0.99$, $i \leq i \leq 10$, then $\prod r_i = .904$. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.

If stage i contains m_i copies of device D_i . Then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$.

The reliability of stage ' i ' is given by a function $\phi_i(m_i)$.

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed.

We wish to solve:

$$\begin{aligned} &\text{Maximize } \prod_{1 \leq i \leq n} \phi_i(m_i) \\ &\text{Subject to } \sum_{1 \leq i \leq n} C_i m_i < C \\ &m_i \geq 1 \text{ and integer, } 1 \leq i \leq n \end{aligned}$$

Assume each $C_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor \left(C + C_i - \sum_{j=1}^n C_j \right) / C_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \geq 1$

An optimal solution m_1, m_2, \dots, m_n is the result of a sequence of decisions, one decision for each m_i .

Let $f_i(x)$ represent the maximum value of $\prod_{1 \leq j \leq i} \phi_j(m_j)$

Subject to the constraints:

$$\sum_{1 \leq j \leq i} C_j m_j \leq x \quad \text{and} \quad 1 \leq m_j \leq u_j, \quad 1 \leq j \leq i$$

The last decision made requires one to choose m_n from $\{1, 2, 3, \dots, u_n\}$

Once a value of m_n has been chosen, the remaining decisions must be such as to use the remaining funds $C - C_n m_n$ in an optimal way.

The principle of optimality holds on

$$f_n(C) = \max_{1 \leq m_n \leq u_n} \{ \phi_n(m_n) f_{n-1}(C - C_n m_n) \}$$

for any $f_i(x_i)$, $i > 1$, this equation generalizes to

$$f_n(x) = \max_{1 \leq m_i \leq u_i} \{ \phi_i(m_i) f_{i-1}(x - C_i m_i) \}$$

clearly, $f_0(x) = 1$ for all x , $0 \leq x \leq C$ and $f(x) = -\infty$ for all $x < 0$.

Let S^i consist of tuples of the form (f, x) , where $f = f_i(x)$.

There is at most one tuple for each different 'x', that result from a sequence of decisions on m_1, m_2, \dots, m_n . The dominance rule (f_1, x_1) dominate (f_2, x_2) if $f_1 \geq f_2$ and $x_1 \leq x_2$. Hence, dominated tuples can be discarded from S^i .

Example 1:

Design a three stage system with device types D_1, D_2 and D_3 . The costs are \$30, \$15 and \$20 respectively. The Cost of the system is to be no more than \$105. The reliability of each device is 0.9, 0.8 and 0.5 respectively.

Solution:

We assume that if stage I has m_i devices of type i in parallel, then $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$

Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$. Where:

$$u_i = \left\lfloor \left(C + C_i - \sum_{j=1}^n C_j \right) / C_i \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3 .

$$u_1 = \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3$$

$$u_3 = \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3$$

We use $S_j^i \rightarrow i$: stage number and J : no. of devices in stage $i = m_i$

$$S^0 = \{f_0(x), x\} \quad \text{initially } f_0(x) = 1 \text{ and } x = 0, \text{ so, } S^0 = \{1, 0\}$$

Compute S^1, S^2 and S^3 as follows:

S^1 = depends on u_1 value, as $u_1 = 2$, so

$$S^1 = \{S_1^1, S_2^1\}$$

S^2 = depends on u_2 value, as $u_2 = 3$, so

$$S^2 = \{S_1^2, S_2^2, S_3^2\}$$

S^3 = depends on u_3 value, as $u_3 = 3$, so

$$S^3 = \{S_1^3, S_2^3, S_3^3\}$$

Now find $S_1^1 = \{(f_1(x), x)\}$

$f_1(x) = \{\phi_1(1)f_0(\cdot), \phi_1(2)f_0(\cdot)\}$ With devices $m_1 = 1$ and $m_2 = 2$

Compute $\phi_1(1)$ and $\phi_1(2)$ using the formula: $\phi_1(mi) = 1 - (1 - r_i)^{m_i}$

$$\phi_1(1) = 1 - (1 - r_1)^{m_1} = 1 - (1 - 0.9)^1 = 0.9$$

$$\phi_1(2) = 1 - (1 - 0.9)^2 = 0.99$$

$$S_1^1 = \{f_1(x), x\} = (0.9, 30)$$

$$S_2^1 = \{0.99, 30 + 30\} = (0.99, 60)$$

Therefore, $S^1 = \{(0.9, 30), (0.99, 60)\}$

Next find $S_1^2 = \{(f_2(x), x)\}$

$f_2(x) = \{\phi_2(1) * f_1(\cdot), \phi_2(2) * f_1(\cdot), \phi_2(3) * f_1(\cdot)\}$

$$\phi_2(1) = 1 - (1 - r_{1mi}) = 1 - (1 - 0.8) = 1 - 0.2 = 0.8$$

$$\phi_2(2) = 1 - (1 - 0.8)^2 = 0.96$$

$$\phi_2(3) = 1 - (1 - 0.8)^3 = 0.992$$

$$S_1^2 = \{(0.8(0.9), 30 + 15), (0.8(0.99), 60 + 15)\} = \{(0.72, 45), (0.792, 75)\}$$

$$S_2^2 = \{(0.96(0.9), 30 + 15 + 15), (0.96(0.99), 60 + 15 + 15)\} \\ = \{(0.864, 60), (0.9504, 90)\}$$

$$S_3^2 = \{(0.992(0.9), 30 + 15 + 15 + 15), (0.992(0.99), 60 + 15 + 15 + 15)\} \\ = \{(0.8928, 75), (0.98208, 105)\}$$

$$S^2 = \{S_1^2, S_2^2, S_3^2\}$$

By applying Dominance rule to S^2 :

Therefore, $S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

Dominance Rule:

If S^i contains two pairs (f_1, x_1) and (f_2, x_2) with the property that $f_1 \geq f_2$ and $x_1 \leq x_2$, then (f_1, x_1) dominates (f_2, x_2) , hence by dominance rule (f_2, x_2) can be discarded. Discarding or pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^i and Dominated tuples has to be discarded from S^i .

Case 1: if $f_1 \leq f_2$ and $x_1 > x_2$ then discard (f_1, x_1)

Case 2: if $f_1 \geq f_2$ and $x_1 < x_2$ the discard (f_2, x_2)

Case 3: otherwise simply write (f_1, x_1)

$$S_2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$$

$$\phi_3(1) = 1 - (1 - r_I)^{m_1} = 1 - (1 - 0.5)^1 = 1 - 0.5 = 0.5$$

$$\phi_3(2) = 1 - (1 - 0.5)^2 = 0.75$$

$$\phi_3(3) = 1 - (1 - 0.5)^3 = 0.875$$

$$S_1^3 = \{(0.5(0.72), 45 + 20), (0.5(0.864), 60 + 20), (0.5(0.8928), 75 + 20)\}$$

$$S_1^3 = \{(0.36, 65), (0.437, 80), (0.4464, 95)\}$$

$$S_2^3 = \{(0.75(0.72), 45 + 20 + 20), (0.75(0.864), 60 + 20 + 20), (0.75(0.8928), 75 + 20 + 20)\}$$

$$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$$

$$S_3^3 = \{(0.875(0.72), 45 + 20 + 20 + 20), (0.875(0.864), 60 + 20 + 20 + 20), (0.875(0.8928), 75 + 20 + 20 + 20)\}$$

$$S_3^3 = \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$$

If cost exceeds 105, remove that tuples

$$S^3 = \{(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)\}$$

The best design has a reliability of 0.648 and a cost of 100. Tracing back for the solution through S^i 's we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

Other Solution:

According to the principle of optimality:

$$f_n(C) = \max_{1 \leq m_n \leq u_n} \{\phi_n(m_n) \cdot f_{n-1}(C - C_n m_n) \text{ with } f_0(x) = 1 \text{ and } 0 \leq x \leq C;\}$$

Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$. Where:

$$u_i = \left\lfloor \frac{C + C_i - \sum_{j=1}^n C_j}{C_i} \right\rfloor$$

Using the above equation compute u_1 , u_2 and u_3 .

$$u_1 = \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3$$

$$u_3 = \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3$$

$$\begin{aligned} f_3(105) &= \max_{1 \leq m_3 \leq u_3} \{ \phi_3(m_3) \cdot f_2(105 - 20m_3) \} \\ &= \max \{ \phi_3(1) f_2(105 - 20), \phi_3(2) f_2(105 - 20 \times 2), \phi_3(3) f_2(105 - 20 \times 3) \} \\ &= \max \{ 0.5 f_2(85), 0.75 f_2(65), 0.875 f_2(45) \} \\ &= \max \{ 0.5 \times 0.8928, 0.75 \times 0.864, 0.875 \times 0.72 \} = 0.648. \end{aligned}$$

$$\begin{aligned} f_2(85) &= \max_{1 \leq m_2 \leq u_2} \{ \phi_2(m_2) \cdot f_1(85 - 15m_2) \} \\ &= \max \{ \phi_2(1) \cdot f_1(85 - 15), \phi_2(2) \cdot f_1(85 - 15 \times 2), \phi_2(3) \cdot f_1(85 - 15 \times 3) \} \\ &= \max \{ 0.8 f_1(70), 0.96 f_1(55), 0.992 f_1(40) \} \\ &= \max \{ 0.8 \times 0.99, 0.96 \times 0.9, 0.99 \times 0.9 \} = 0.8928 \end{aligned}$$

$$\begin{aligned} f_1(70) &= \max_{1 \leq m_1 \leq u_1} \{ \phi_1(m_1) \cdot f_0(70 - 30m_1) \} \\ &= \max \{ \phi_1(1) f_0(70 - 30), \phi_1(2) f_0(70 - 30 \times 2) \} \\ &= \max \{ \phi_1(1) \times 1, \phi_1(2) \times 1 \} = \max \{ 0.9, 0.99 \} = 0.99 \end{aligned}$$

$$\begin{aligned} f_1(55) &= \max_{1 \leq m_1 \leq u_1} \{ \phi_1(m_1) \cdot f_0(55 - 30m_1) \} \\ &= \max \{ \phi_1(1) f_0(55 - 30), \phi_1(2) f_0(55 - 30 \times 2) \} \\ &= \max \{ \phi_1(1) \times 1, \phi_1(2) \times -\infty \} = \max \{ 0.9, -\infty \} = 0.9 \end{aligned}$$

$$\begin{aligned} f_1(40) &= \max_{1 \leq m_1 \leq u_1} \{ \phi_1(m_1) \cdot f_0(40 - 30m_1) \} \\ &= \max \{ \phi_1(1) f_0(40 - 30), \phi_1(2) f_0(40 - 30 \times 2) \} \\ &= \max \{ \phi_1(1) \times 1, \phi_1(2) \times -\infty \} = \max \{ 0.9, -\infty \} = 0.9 \end{aligned}$$

$$\begin{aligned}
f_2(65) &= \max_{1 \leq m_2 \leq u_2} \{\phi_2(m_2) \cdot f_1(65 - 15m_2)\} \\
&= \max \{\phi_2(1) f_1(65 - 15), \phi_2(2) f_1(65 - 15 \times 2), \phi_2(3) f_1(65 - 15 \times 3)\} \\
&= \max \{0.8 f_1(50), 0.96 f_1(35), 0.992 f_1(20)\} \\
&= \max \{0.8 \times 0.9, 0.96 \times 0.9, -\infty\} = 0.864
\end{aligned}$$

$$\begin{aligned}
f_1(50) &= \max_{1 \leq m_1 \leq u_1} \{\phi_1(m_1) \cdot f_0(50 - 30m_1)\} \\
&= \max \{\phi_1(1) f_0(50 - 30), \phi_1(2) f_0(50 - 30 \times 2)\} \\
&= \max \{\phi_1(1) \times 1, \phi_1(2) \times -\infty\} = \max\{0.9, -\infty\} = 0.9
\end{aligned}$$

$$\begin{aligned}
f_1(35) &= \max_{1 \leq m_1 \leq u_1} \{\phi_1(m_1) \cdot f_0(35 - 30m_1)\} \\
&= \max \{\phi_1(1) \cdot f_0(35 - 30), \phi_1(2) \cdot f_0(35 - 30 \times 2)\} \\
&= \max \{\phi_1(1) \times 1, \phi_1(2) \times -\infty\} = \max\{0.9, -\infty\} = 0.9
\end{aligned}$$

$$\begin{aligned}
f_1(20) &= \max_{1 \leq m_1 \leq u_1} \{\phi_1(m_1) \cdot f_0(20 - 30m_1)\} \\
&= \max \{\phi_1(1) f_0(20 - 30), \phi_1(2) f_0(20 - 30 \times 2)\} \\
&= \max \{\phi_1(1) \times -\infty, \phi_1(2) \times -\infty\} = \max\{-\infty, -\infty\} = -\infty
\end{aligned}$$

$$\begin{aligned}
f_2(45) &= \max_{1 \leq m_2 \leq u_2} \{\phi_2(m_2) \cdot f_1(45 - 15m_2)\} \\
&= \max \{\phi_2(1) f_1(45 - 15), \phi_2(2) f_1(45 - 15 \times 2), \phi_2(3) f_1(45 - 15 \times 3)\} \\
&= \max \{0.8 f_1(30), 0.96 f_1(15), 0.992 f_1(0)\} \\
&= \max \{0.8 \times 0.9, 0.96 \times -\infty, 0.99 \times -\infty\} = 0.72
\end{aligned}$$

$$\begin{aligned}
f_1(30) &= \max_{1 \leq m_1 \leq u_1} \{\phi_1(m_1) \cdot f_0(30 - 30m_1)\} \\
&= \max \{\phi_1(1) f_0(30 - 30), \phi_1(2) f_0(30 - 30 \times 2)\} \\
&= \max \{\phi_1(1) \times 1, \phi_1(2) \times -\infty\} = \max\{0.9, -\infty\} = 0.9
\end{aligned}$$

Similarly, $f_1(15) = -\infty$, $f_1(0) = -\infty$.

The best design has a reliability = 0.648 and

Cost = $30 \times 1 + 15 \times 2 + 20 \times 2 = 100$.

Tracing back for the solution through S^i 's we can determine that:

$m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

Chapter

6

BASIC TRAVERSAL AND SEARCH TECHNIQUES

Search means finding a path or traversal between a start node and one of a set of goal nodes. Search is a study of states and their transitions.

Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal.

Techniques for Traversal of a Binary Tree:

A binary tree is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and remaining elements (if any) are partitioned into two binary trees, which are called the left and right subtrees.

There are three common ways to traverse a binary tree:

1. Preorder
2. Inorder
3. postorder

In all the three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among the three orders comes from the difference in the time at which a node is visited.

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for preorder traversal is as follows:

treenode = record

```
{
    Type data;
    Treenode *lchild; treenode *rchild;
}
```

//Type is the data type of data.

algorithm inorder (t)

// t is a binary tree. Each node of t has three fields: lchild, data, and rchild.

```
{
    if t ≠ 0 then
    {
        inorder (t → lchild);
        visit (t);
        inorder (t → rchild);
    }
}
```

Preorder Traversal:

In a preorder traversal, each node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

Algorithm Preorder (t)

// t is a binary tree. Each node of t has three fields; lchild, data, and rchild.

```
{
    if t ≠ 0 then
    {
        visit (t);
        Preorder (t → lchild);
        Preorder (t → rchild);
    }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for preorder traversal is as follows:

Algorithm Postorder (t)

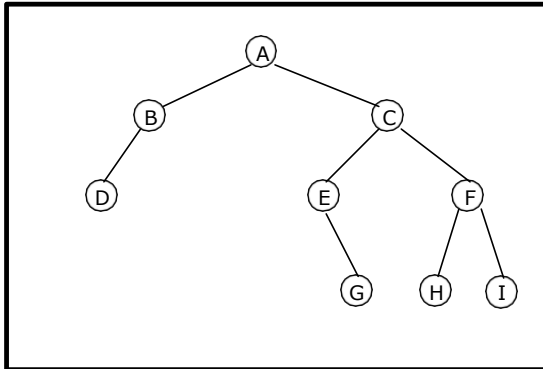
// t is a binary tree. Each node of t has three fields : lchild, data, and rchild.

```
{
    if t ≠ 0 then
    {
        Postorder (t → lchild);
        Postorder (t → rchild);
        visit(t);
    }
}
```

Examples for binary tree traversal/search technique:

Example 1:

Traverse the following binary tree in pre, post and in-order.



Bin a ry T re e

Preordering of the vertices:
A, B, D, C, E, G, F, H, I.

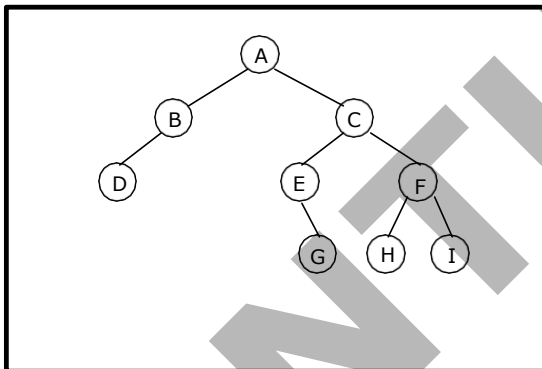
Postordering of the vertices:
D, B, G, E, H, I, F, C, A.

Inordering of the vertices:
D, B, A, E, G, C, H, F, I

Pre, Post and In-order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



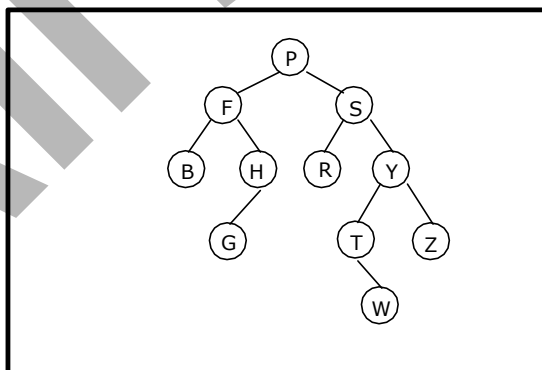
Bin a ry T re e

- Preorder traversal yields:
A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
D, B, A, E, G, C, H, F, I
- Level order traversal yields:
A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



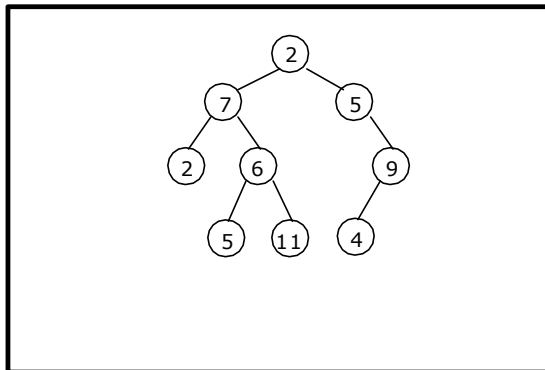
Bin a ry T re e

- Preorder traversal yields:
P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



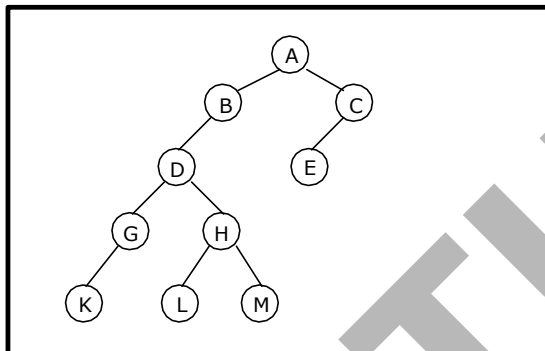
Bin a ry T re e

- Preorder traversal yields:
2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder traversal yields:
2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 5:

Traverse the following binary tree in pre, post, inorder and level order.



Bin a ry T re e

- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C
- Level order traversal yields:
A, B, C, D, E, G, H, K, L, M

Pre, Post, Inorder and level order Traversing

Non Recursive Binary Tree Traversal Algorithms:

At first glance, it appears we would always want to use the flat traversal functions since they use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

The algorithm for inorder Non Recursive traversal is as follows:

Algorithm **inorder()**

```
{
    stack[1] = 0
    vertex = root
top: while(vertex ≠ 0)
    {
        push the vertex into the stack
        vertex = leftson(vertex)
    }

    pop the element from the stack and make it as vertex

    while(vertex ≠ 0)
    {
        print the vertex node
        if(rightson(vertex) ≠ 0)
        {
            vertex = rightson(vertex)
            goto top
        }
        pop the element from the stack and made it as vertex
    }
}
```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

The algorithm for preorder Non Recursive traversal is as follows:

Algorithm **preorder()**

```
{
    stack[1]: = 0
    vertex := root.
    while(vertex ≠ 0)
    {
        print vertex node
        if(rightson(vertex) ≠ 0)
            push the right son of vertex into the stack.
        if(leftson(vertex) ≠ 0)
            vertex := leftson(vertex)
        else
            pop the element from the stack and made it as vertex
    }
}
```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push $-(\text{right son of vertex})$ onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

The algorithm for postorder Non Recursive traversal is as follows:

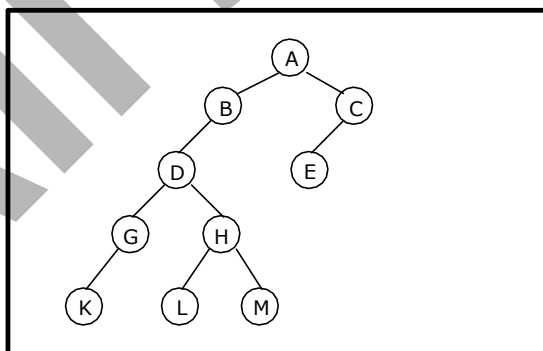
Algorithm **postorder**()

```
{
    stack[1] := 0
    vertex := root

    top: while(vertex ≠ 0)
    {
        push vertex onto stack
        if(rightson(vertex) ≠ 0)
            push -(vertex) onto stack
        vertex := leftson(vertex)
    }
    pop from stack and make it as vertex
    while(vertex > 0)
    {
        print the vertex node
        pop from stack and make it as vertex
    }
    if(vertex < 0)
    {
        vertex := -(vertex)
        goto top
    }
}
```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Bin ary T re e

- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
H	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A -C B D -H G K		PUSH the left most path of A with a -ve for right sons
	0 A -C B D -H	K G	POP all +ve nodes K and G
H	0 A -C B D	K G	Pop H
	0 A -C B D H -M L	K G	PUSH the left most path of H with a -ve for right sons
	0 A -C B D H -M	K G L	POP all +ve nodes L
M	0 A -C B D H	K G L	Pop M
	0 A -C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A -C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0		Stop since stack is empty

Preorder Traversal:

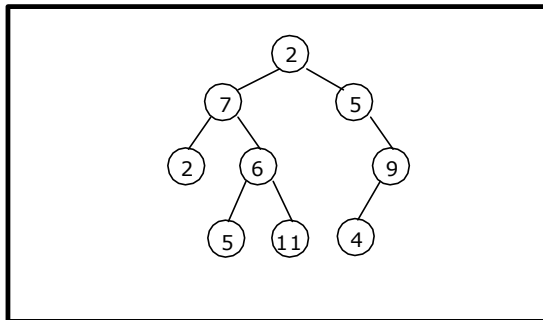
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Bin a ry T re e

- Preorder traversal yields:
2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder traversal yields:
2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
2, 7, 5, 6, 11, 2, 5, 4, 9

Pre, Post and In order Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
11	0 2	2 7 5 6 11	
5	0 5	2 7 5 6 11 2	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	
	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 2 -5 7 -6 2		
2	0 2 -5 7 -6	2	
6	0 2 -5 7	2	
	0 2 -5 7 6 -11 5	2	
5	0 2 -5 7 6 -11	2 5	
11	0 2 -5 7 6 11	2 5	
	0 2 -5	2 5 11 6 7	
5	0 2 5 -9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11	
9	0	2 7 2 6 5 11 5	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty

Techniques for graphs:

Given a graph $G = (V, E)$ and a vertex V in $V(G)$ traversing can be done in two ways.

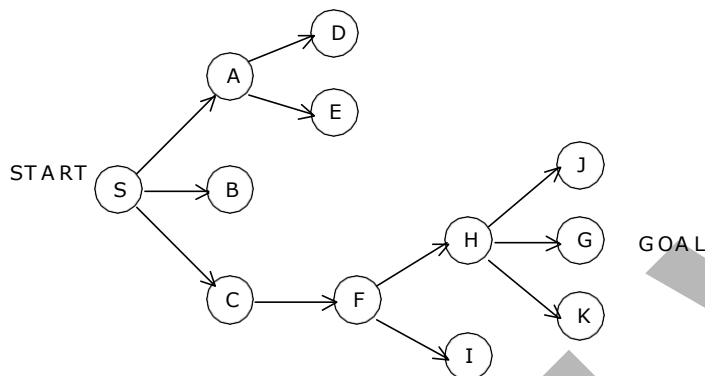
1. Depth first search
2. Breadth first search
3. D-search (Depth Search)

Depth first search:

With depth first search, the start state is chosen to begin, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not yet been found, then it "backs up" that means it goes to the immediately previous state or predecessor formally, the state whose successor was 'S' originally.

For example consider the figure. The circled letters are state and arrows are branches.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state. So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get stuck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.
To eliminate this keep a list of states previously visited, and never permit search to return to any of them.
2. One more problem is that, the state space tree may be of infinite depth, to prevent consideration of paths that are too long, a maximum is often placed on the depth of nodes to be expanded, and any node at that depth is treated as if it had no successors.
3. We cannot come up with shortest solution to the problem.

Time Complexity:

Let $n = |V|$ and $e = |E|$. Observe that the initialization portion requires $\Phi(n)$ time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n assuming each vertex is reachable from the source). As, each vertex is visited at most once. At each vertex visited, we scan its adjacency list once. Thus, each edge is examined at most twice (once at each endpoint). So the total running time is $O(n + e)$.

Alternatively,

If the average branching factor is assumed as 'b' and the depth of the solution as 'd', and maximum depth $m \geq d$.

The worst case time complexity is $O(b^m)$ as we explore b^m nodes. If many solutions exists DFS will be likely to find faster than the BFS.

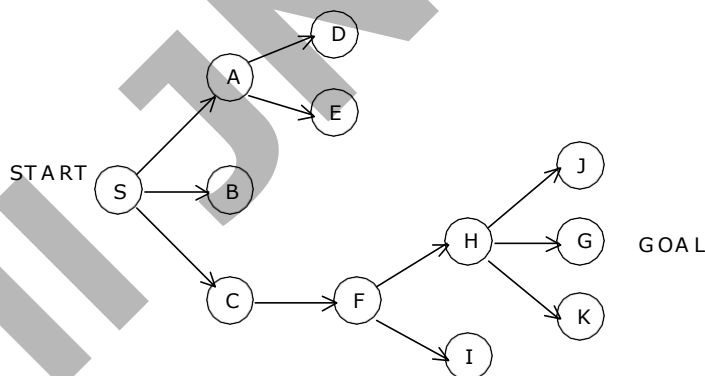
Space Complexity:

We have to store the nodes from root to current leaf and all the unexpanded siblings of each node on path. So, We need to store bm nodes.

Breadth first search:

Given an graph $G = (V, E)$, breadth-first search starts at some source vertex S and "discovers" which vertices are reachable from S . Define the distance between a vertex V and S to be the minimum number of edges on a path from S to V . Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths (where the length of a path = number of edges on the path). Breadth-first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A , B and C are at level 1; D , e and F at level 2; H and I at level 3; and J , G and K at level 4. So breadth first search, will consider in order S , A , B , C , D , E , F , H , I , J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state. Any time we need a new state, we pick it from the front of the queue and any time we find successors, we put them at the end of the queue. That way we are guaranteed to not try (find successors of) any states at level 'N' until all states at level 'N - 1' have been tried.

Time Complexity:

The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. Let $n = |V|$ and $e = |E|$. Observe that the initialization portion requires $\Phi(n)$ time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n , assuming each vertex is reachable from the source). So, Running time is $O(n + e)$ as in DFS. For a directed graph the analysis is essentially the same.

Alternatively,

If the average branching factor is assumed as 'b' and the depth of the solution as 'd'.

In the worst case we will examine $1 + b + b^2 + b^3 + \dots + b^d = (b^{d+1} - 1) / (b - 1) = O(b^d)$.

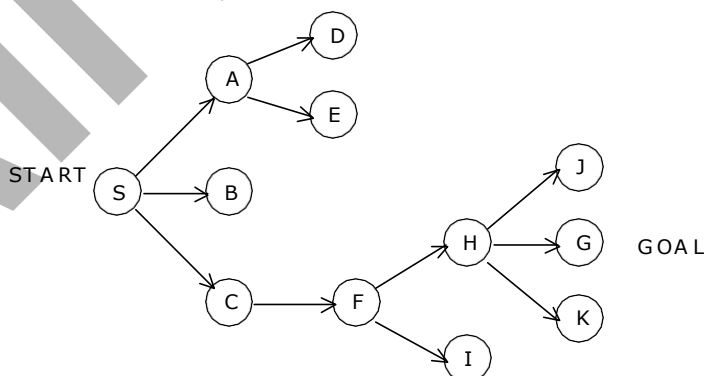
In the average case the last term of the series would be $b^d / 2$. So, the complexity is still $O(b^d)$

Space Complexity:

Before examining any node at depth d, all of its siblings must be expanded and stored. So, space requirement is also $O(b^d)$.

Depth Search (D-Search):

The exploration of a new node cannot begin until the node currently being explored is fully explored. D-search like state space search is called LIFO (Last In First Out) search which uses stack data structure. To illustrate the D-search let us consider the following tree:



The search order for goal node (G) is as follows: S, A, B, C, F, H, I, J, G.

Time Complexity:

The time complexity is same as Breadth first search.

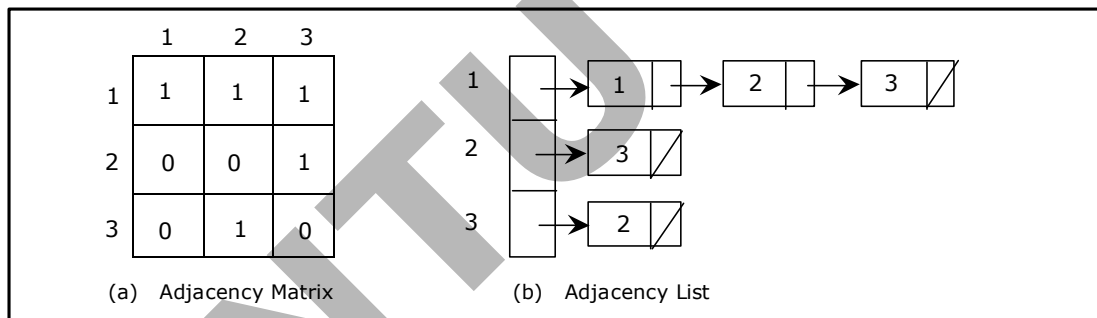
Representation of Graphs and Digraphs by Adjacency List:

We will describe two ways of representing digraphs. We can represent undirected graphs using exactly the same representation, but we will double each edge, representing the undirected edge $\{v, w\}$ by the two oppositely directed edges (v, w) and (w, v) . Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

Let $G = (V, E)$ be a digraph with $n = |V|$ and let $e = |E|$. We will assume that the vertices of G are indexed $\{1, 2, \dots, n\}$.

$$A[v, w] = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency List: An array $\text{Adj}[1 \dots n]$ of pointers where for $1 \leq v \leq n$, $\text{Adj}[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.



Adjacency matrix and adjacency list

An adjacency matrix requires $\Theta(n^2)$ storage and an adjacency list requires $\Theta(n + e)$ storage.

Adjacency matrices allow faster access to edge queries (for example, is $(u, v) \in E$) and adjacency lists allow faster access to enumeration tasks (for example, find all the vertices adjacent to v).

Depth First and Breadth First Spanning Trees:

BFS and DFS impose a tree (the BFS/DFS tree) along with some auxiliary edges (cross edges) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. Trees are much more structured objects than graphs. For example, trees break up nicely into subtrees, upon which subproblems can be solved recursively. For directed graphs the other edges of the graph can be classified as follows:

Back edges: (u, v) where v is a (not necessarily proper) ancestor of u in the tree. (Thus, a self-loop is considered to be a back edge).

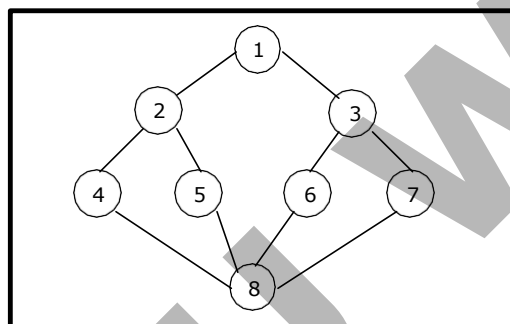
Forward edges: (u, v) where v is a proper descendent of u in the tree.

Cross edges: (u, v) where u and v are not ancestors or descendants of one another (in fact, the edge may go between different trees of the forest).

Depth first search and traversal:

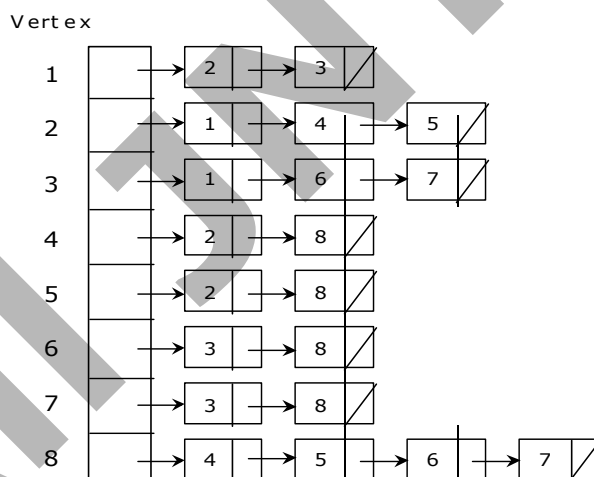
Depth first search of undirected graph proceeds as follows. The start vertex V is visited. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'u' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

Let us consider the following Graph (G):

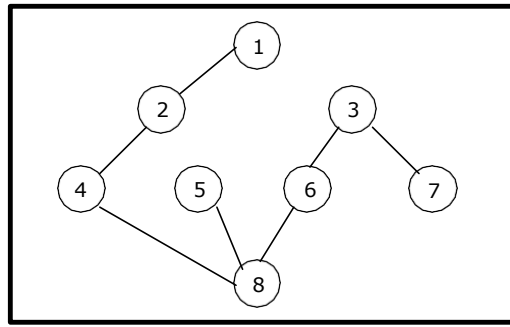


Graph

The adjacency list for G is:



If the depth first is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:

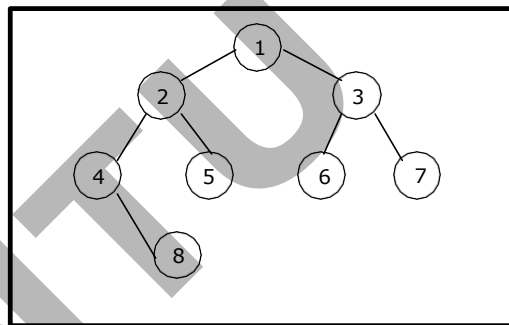


Depth First Spanning Tree

The spanning trees obtained using depth first searches are called depth first spanning trees. The edges rejected in the context of depth first search are called a back edges. Depth first spanning tree has no cross edges.

Breadth first search and traversal:

Starting at vertex 'V' and marking it as visited, BFS differs from DFS in that all unvisited vertices adjacent to V are visited next. Then unvisited vertices adjacent to there vertices are visited and so on. A breadth first search beginning at vertex 1 of the graph would first visit 1 and then 2 and 3.



Breadth First Spanning Tree

Next vertices 4, 5, 6 and 7 will be visited and finally 8. The spanning trees obtained using BFS are called Breadth first spanning trees. The edges that were rejected in the breadth first search are called cross edges.

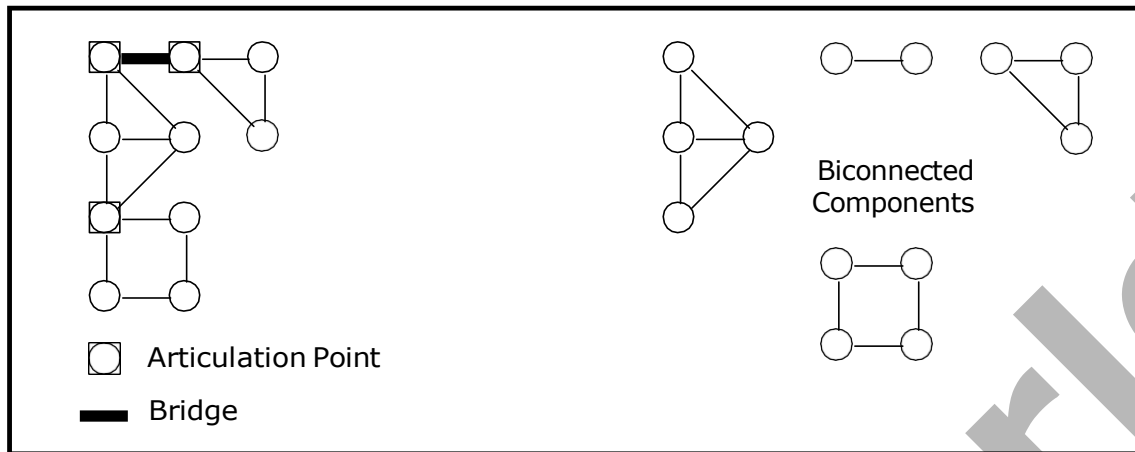
Articulation Points and Biconnected Components:

Let $G = (V, E)$ be a connected undirected graph. Consider the following definitions:

Articulation Point (or Cut Vertex): An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces..

Bridge: Is an edge whose removal results in a disconnected graph.

Biconnected: A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components. This is illustrated in the following figure:



Articulation Points and Bridges

Biconnected graphs and articulation points are of great interest in the design of network algorithms, because these are the "critical" points, whose failure will result in the network becoming disconnected.

Let us consider the typical case of vertex v , where v is not a leaf and v is not the root. Let w_1, w_2, \dots, w_k be the children of v . For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is no back edge going to a proper ancestor of v , then if we remove v , this subtree becomes disconnected from the rest of the graph, and hence v is an articulation point.

On the other hand, if every one of the subtree rooted at the children of v have back edges to proper ancestors of v , then if v is removed, the graph remains connected (the back edges hold everything together). This leads to the following:

Observation 1: An internal vertex v of the DFS tree (other than the root) is an articulation point if and only if there is a subtree rooted at a child of v such that there is no back edge from any vertex in this subtree to a proper ancestor of v .

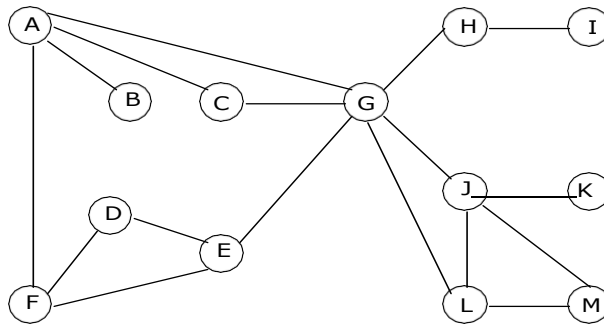
Observation 2: A leaf of the DFS tree is never an articulation point, since a leaf will not have any subtrees in the DFS tree.

Thus, after deletion of a leaf from a tree, the rest of the tree remains connected, thus even ignoring the back edges, the graph is connected after the deletion of a leaf from the DFS tree.

Observation 3: The root of the DFS is an articulation point if and only if it has two or more children. If the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree, and hence cannot disconnect the graph in general.

Articulation Points by Depth First Search:

Determining the articulation turns out to be a simple extension of depth first search. Consider a depth first spanning tree for this graph.

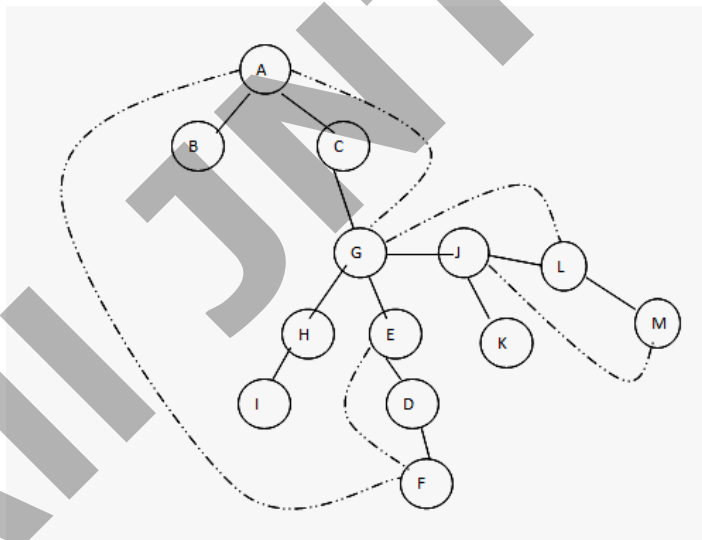


Observations 1, 2, and 3 provide us with a structural characterization of which vertices in the DFS tree are articulation points.

Deleting node E does not disconnect the graph because G and D both have dotted links (back edges) that point above E, giving alternate paths from them to F. On the other hand, deleting G does disconnect the graph because there are no such alternate paths from L or H to E (G's parent).

A vertex 'x' is not an articulation point if every child 'y' has some node lower in the tree connect (via a dotted link) to a node higher in the tree than 'x', thus providing an alternate connection from 'x' to 'y'. This rule will not work for the root node since there are no *nodes higher in the tree*. The root is an articulation point if it has two or more children.

Depth First Spanning Tree for the above graph is:



By using the above observations the articulation points of this graph are:

- A : because it connects B to the rest of the graph.
- H : because it connects I to the rest of the graph.
- J : because it connects K to the rest of the graph.
- G : because the graph would fall into three pieces if G is deleted.

Biconnected components are: {A, C, G, D, E, F}, {G, J, L, M}, B, H, I and K

This observation leads to a simple rule to identify articulation points. For each is define $L(u)$ as follows:

$$L(u) = \min \{ \text{DFN}(u), \min \{ L(w) \mid w \text{ is a child of } u \}, \min \{ \text{DFN}(w) \mid (u, w) \text{ is a back edge} \} \}.$$

$L(u)$ is the lowest depth first number that can be reached from 'u' using a path of descendents followed by at most one back edge. It follows that, If 'u' is not the root then 'u' is an articulation point iff 'u' has a child 'w' such that:

$$L(w) \geq \text{DFN}(u)$$

6.6.2. Algorithm for finding the Articulation points:

Pseudocode to compute DFN and L.

Algorithm Art (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initialized to zero and that // the
// global variable num is initialized to 1. n is the number of vertices in G.
{
    dfn[u] := num; L[u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if (dfn[w] = 0) then
        {
            Art(w, u); // w is unvisited.
            L[u] := min(L[u], L[w]);
        }
        else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
    }
}
```

6.6.1. Algorithm for finding the Biconnected Components:

Algorithm BiComp (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initially zero and that the
// global variable num is initialized to 1. n is the number of vertices in G.
{
    dfn[u] := num; L[u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if ((v ≠ w) and (dfn[w] ≤ dfn[u])) then
            add (u, w) to the top of a stack s;
        if (dfn[w] = 0) then
        {
            if (L[w] ≥ dfn[u]) then
            {
                write ("New bicomponent");
                repeat
                {
                    Delete an edge from the top of stack s;
                    Let this edge be (x, y);
                    Write (x, y);
                } until (((x, y) = (u, w)) or ((x, y) = (w, u)));
            }
        }
    }
}
```

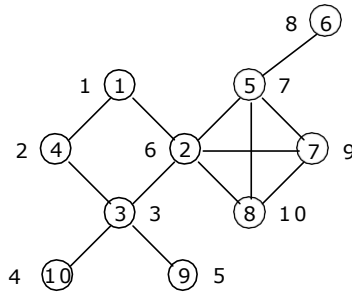
```

        BiComp (w, u);                                // w is unvisited.
        L [u] := min (L [u], L [w]);
    }
    else if (w ≠ v) then L [u] := min (L [u], dfn [w]);
}
}

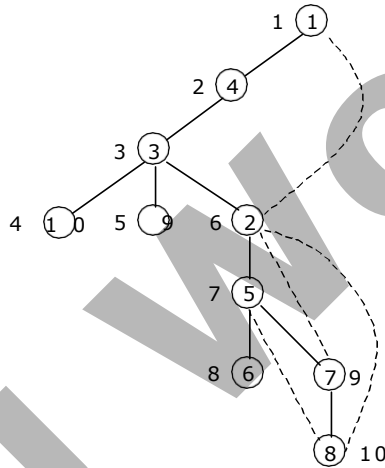
```

6.7.1. Example:

For the following graph identify the articulation points and Biconnected components:



Graph h



Depth First Spanning Tree

To identify the articulation points, we use:

$$L(u) = \min \{ \text{DFN}(u), \min \{ L(w) \mid w \text{ is a child of } u \}, \min \{ \text{DFN}(w) \mid w \text{ is a vertex to which there is back edge from } u \} \}$$

$$L(1) = \min \{ \text{DFN}(1), \min \{ L(4) \} \} = \min \{ 1, L(4) \} = \min \{ 1, 1 \} = 1$$

$$L(4) = \min \{ \text{DFN}(4), \min \{ L(3) \} \} = \min \{ 2, L(3) \} = \min \{ 2, 1 \} = 1$$

$$L(3) = \min \{ \text{DFN}(3), \min \{ L(10), L(9), L(2) \} \} = \min \{ 3, \min \{ L(10), L(9), L(2) \} \} = \min \{ 3, \min \{ 4, 5, 1 \} \} = 1$$

$$L(10) = \min \{ \text{DFN}(10) \} = 4$$

$$L(9) = \min \{ \text{DFN}(9) \} = 5$$

$$L(2) = \min \{ \text{DFN}(2), \min \{ L(5) \}, \min \{ \text{DFN}(1) \} \} = \min \{ 6, \min \{ L(5) \}, 1 \} = \min \{ 6, 6, 1 \} = 1$$

$$L(5) = \min \{ \text{DFN}(5), \min \{ L(6), L(7) \} \} = \min \{ 7, 8, 6 \} = 6$$

$$L(6) = \min \{ \text{DFN}(6) \} = 8$$

$$L(7) = \min \{ \text{DFN}(7), \min \{ L(8), \min \{ \text{DFN}(2) \} \} \} = \min \{ 9, L(8), 6 \} = \min \{ 9, 6, 6 \} = 6$$

$$L(8) = \min \{ \text{DFN}(8), \min \{ \text{DFN}(5), \text{DFN}(2) \} \}$$

$$= \min \{10, \min (7, 6)\} = \min \{10, 6\} = 6$$

Therefore, $L(1: 10) = (1, 1, 1, 1, 6, 8, 6, 6, 5, 4)$

Finding the Articulation Points:

Vertex 1: Vertex 1 is not an articulation point. It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as child 5 has $L(5) = 6$ and $DFN(2) = 6$,
So, the condition $L(5) = DFN(2)$ is true.

Vertex 3: is an articulation point as child 10 has $L(10) = 4$ and $DFN(3) = 3$,
So, the condition $L(10) > DFN(3)$ is true.

Vertex 4: is not an articulation point as child 3 has $L(3) = 1$ and $DFN(4) = 2$,
So, the condition $L(3) \geq DFN(4)$ is false.

Vertex 5: is an articulation point as child 6 has $L(6) = 8$, and $DFN(5) = 7$,
So, the condition $L(6) > DFN(5)$ is true.

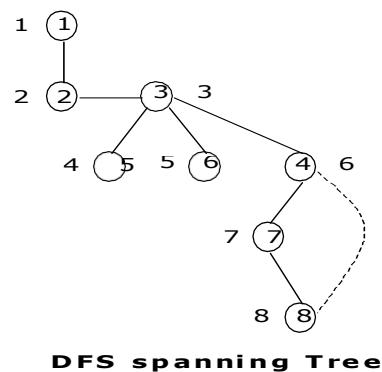
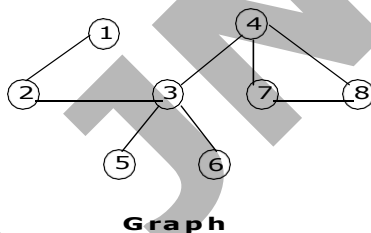
Vertex 7: is not an articulation point as child 8 has $L(8) = 6$, and $DFN(7) = 9$,
So, the condition $L(8) \geq DFN(7)$ is false.

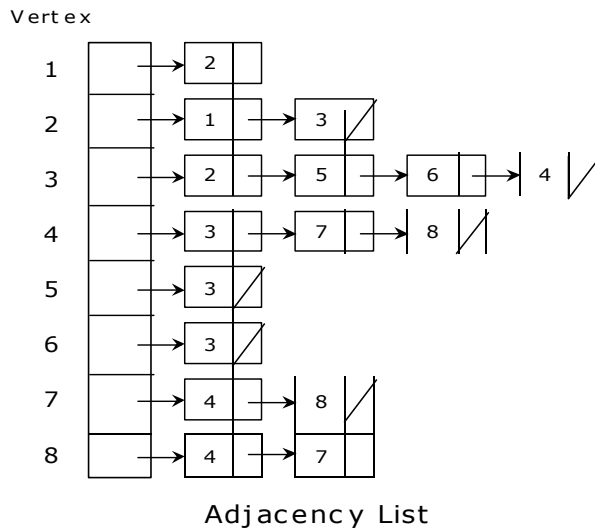
Vertex 6, Vertex 8, Vertex 9 and Vertex 10 are leaf nodes.

Therefore, the articulation points are $\{2, 3, 5\}$.

Example:

For the following graph identify the articulation points and Biconnected components:





$L(u) = \min \{DFN(u), \min \{L(w) \mid w \text{ is a child of } u\}, \min \{DFN(w) \mid w \text{ is a vertex to which there is back edge from } u\}\}$

$L(1) = \min \{DFN(1), \min \{L(2)\}\} = \min \{1, L(2)\} = \min \{1, 2\} = 1$

$L(2) = \min \{DFN(2), \min \{L(3)\}\} = \min \{2, L(3)\} = \min \{2, 3\} = 2$

$L(3) = \min \{DFN(3), \min \{L(4), L(5), L(6)\}\} = \min \{3, \min \{6, 4, 5\}\} = 3$

$L(4) = \min \{DFN(4), \min \{L(7)\}\} = \min \{6, L(7)\} = \min \{6, 6\} = 6$

$L(5) = \min \{DFN(5)\} = 4$

$L(6) = \min \{DFN(6)\} = 5$

$L(7) = \min \{DFN(7), \min \{L(8)\}\} = \min \{7, 6\} = 6$

$L(8) = \min \{DFN(8), \min \{DFN(4)\}\} = \min \{8, 6\} = 6$

Therefore, $L(1:8) = \{1, 2, 3, 6, 4, 5, 6, 6\}$

Finding the Articulation Points:

Check for the condition if $L(w) \geq DFN(u)$ is true, where w is any child of u .

Vertex 1: Vertex 1 is not an articulation point.

It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as $L(3) = 3$ and $DFN(2) = 2$.

So, the condition is true

Vertex 3: is an articulation Point as:

- I. $L(5) = 4$ and $DFN(3) = 3$
- II. $L(6) = 5$ and $DFN(3) = 3$ and
- III. $L(4) = 6$ and $DFN(3) = 3$

So, the condition true in above cases

Vertex 4: is an articulation point as $L(7) = 6$ and $DFN(4) = 6$.
So, the condition is true

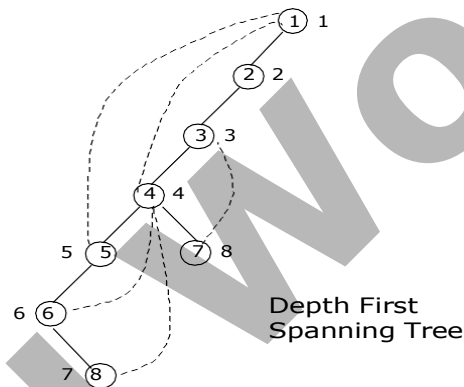
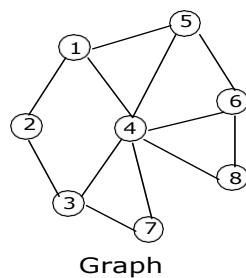
Vertex 7: is not an articulation point as $L(8) = 6$ and $DFN(7) = 7$.
So, the condition is False

Vertex 5, Vertex 6 and Vertex 8 are leaf nodes.

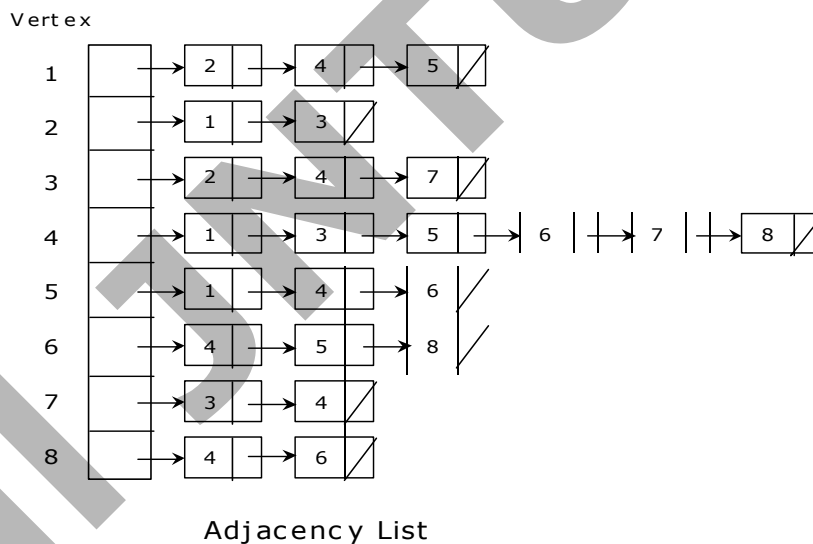
Therefore, the articulation points are $\{2, 3, 4\}$.

Example:

For the following graph identify the articulation points and Biconnected components:



$DFN(1: 8) = \{1, 2, 3, 4, 5, 6, 8, 7\}$



$L(u) = \min \{DFN(u), \min \{L(w) \mid w \text{ is a child of } u\}, \min \{DFN(w) \mid w \text{ is a vertex to which there is back edge from } u\}\}$

$L(1) = \min \{DFN(1), \min \{L(2)\}\}$
 $= \min \{1, L(2)\} = 1$

$L(2) = \min \{DFN(2), \min \{L(3)\}\} = \min \{2, L(3)\} = \min \{2, 1\} = 1$

$L(3) = \min \{DFN(3), \min \{L(4)\}\} = \min \{3, L(4)\} = \min \{3, L(4)\}$
 $= \min \{3, 1\} = 1$

$L(4) = \min \{DFN(4), \min \{L(5), L(7)\}, \min \{DFN(1)\}\}$
 $= \min \{4, \min \{L(5), L(7)\}, 1\} = \min \{4, \min \{1, 3\}, 1\}$
 $= \min \{4, 1, 1\} = 1$

$L(5) = \min \{DFN(5), \min \{L(6)\}, \min \{DFN(1)\}\} = \min \{5, L(6), 1\}$
 $= \min \{5, 4, 1\} = 1$

$L(6) = \min \{DFN(6), \min \{L(8)\}, \min \{DFN(4)\}\} = \min \{6, L(8), 4\}$
 $= \min \{6, 4, 4\} = 4$

$L(7) = \min \{DFN(7), \min \{DFN(3)\}\} = \min \{8, 3\} = 3$

$L(8) = \min \{DFN(8), \min \{DFN(4)\}\} = \min \{7, 4\} = 4$

Therefore, $L(1:8) = \{1, 1, 1, 1, 1, 4, 3, 4\}$

Finding the Articulation Points:

Check for the condition if $L(w) \geq DFN(u)$ is true, where w is any child of u .

Vertex 1: is not an articulation point.

It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is not an articulation point. As $L(3) = 1$ and $DFN(2) = 2$.
So, the condition is False.

Vertex 3: is not an articulation Point as $L(4) = 1$ and $DFN(3) = 3$.
So, the condition is False.

Vertex 4: is not an articulation Point as:
 $L(3) = 1$ and $DFN(2) = 2$ and
 $L(7) = 3$ and $DFN(4) = 4$
 So, the condition fails in both cases.

Vertex 5: is not an Articulation Point as $L(6) = 4$ and $DFN(5) = 6$.
So, the condition is False

Vertex 6: is not an Articulation Point as $L(8) = 4$ and $DFN(6) = 7$.
So, the condition is False

Vertex 7: is a leaf node.

Vertex 8: is a leaf node.

So they are no articulation points.

GAME PLAYING

In game-playing literature, the term play is used for a move. The two major components of game-playing program are plausible move generator, and a static evaluation function generator.

In a game of chess, for every move a player makes, the average branching factor is 35, that is the opponent can make 35 different moves. It may not be possible to examine all the states because the amount of time given for a move is limited and the amount of computational power available for examining various states is also limited. Hence it is essential that only very selected moves or paths are examined. For this purpose only, one has a plausible move generator, that expands or generates only selected moves.

Static Evaluation function generator (SEF) is the most important component of the game-playing program. The process of computing a number that reflects board quality is called static evaluation. The procedure that does the computation is called a static evaluator. A static evaluation function, is defined as the one that estimates the value of the board position without looking any of that position's successors.

The SEF gives a snapshot of a particular move. More the SEF value, more is the probability for a victory. Games can be classified as either a single person playing or multi-person playing. For single person games (for example, Rubik's cube, 8-tile puzzle etc.) search strategies such as Best-first branch and bound algorithm can be used.

On the other hand, in a two-person games like chess, checkers etc., each player tries to outsmart the opponent. Each has their own way of evaluating the situation and since each player tries to obtain the maximum benefits, best first search branch and bound algorithm do not serve the purpose. The basic methods available for game playing are:

1. Minimax search.
2. Minimax search with alpha beta cutoffs.

MINIMAX SEARCH

The standard algorithm for two-player games is called minimax search with static evaluation. We have to add more knowledge to improve search and to reduce the complexity. Heuristic search adds a small amount of knowledge to a problem space, giving, surprisingly, a fairly dramatic effect of the efficiency of search algorithm. A good heuristic is to select roads that go in the direction of the goal. In this case we call it as heuristic static evaluation function that takes a board position and returns a number that indicates how favorable that position is to one player or other. We call our players as **MAX** and **MIN**.

The large positive values would correspond to strong positions for one player whereas large negative values would represent advantageous situations for the opponent.

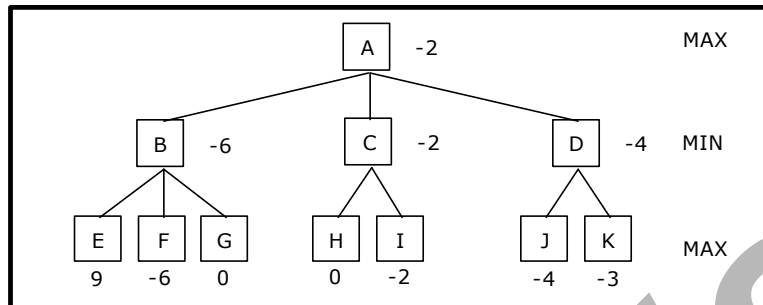
The player for whom large positive values are advantageous is called the MAX, and conversely the opponent is referred to as MIN. The value of a node where it is MAX's turn to move is maximum of the values of its children, while the value of a node where MIN is to move is the minimum of the values of its children.

MAX represents the player trying to win, or to MAXimize his/her advantage. MIN, the opponent attempts to MINimize MAX's score. (i.e. we can assume that MIN uses the same information and always attempts to move to a state that is worst for MAX).

At alternate levels of tree, the minimum and the maximum values of the children are backedup. The backup values is done as follows: The (MAX node) parent of MIN tip nodes is assigned a backedup value equal to the maximum of the evaluations of the tip nodes, on the other hand, if MIN were to choose among tip nodes, he will choose that having the smallest evaluation (the most negative). Therefore, the (MIN node), parent of MAX tip nodes is assigned a backedup value equal to the minimum of the evaluation of the tip nodes.

After the parents of all tip nodes have been assigned backup values, we backup values another level, assuming that MAX would choose that node with largest backup value, while MIN would choose that node with smallest backup value.

We continue the procedure level by level, until finally, the successors of the start node are assigned backup values. The search algorithm then uses these derived values to select among possible next moves.

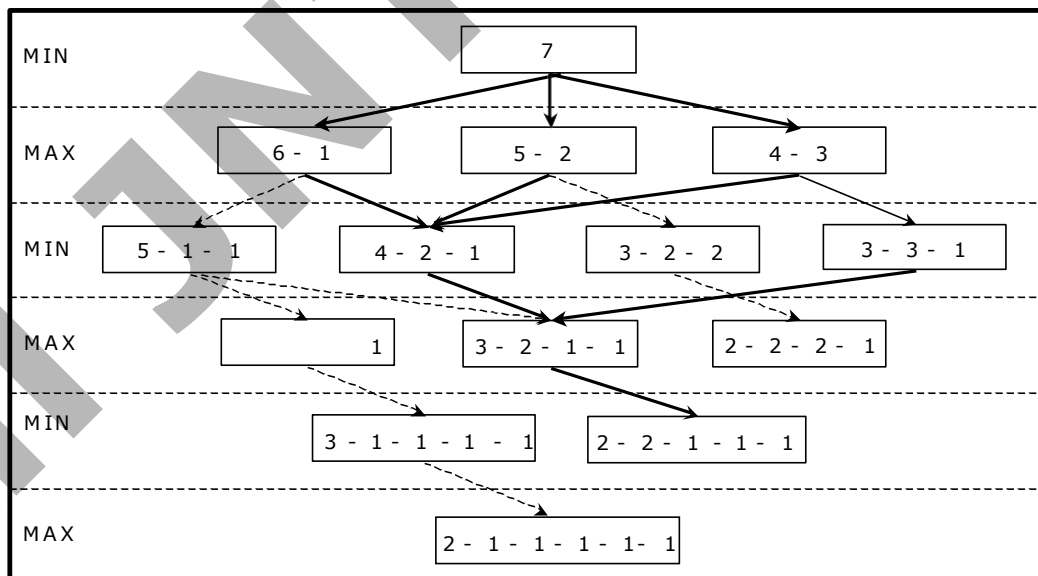


Backingup the values of a 2 - ply search

The best first move can be extracted from minimax procedure after search terminates.

EXAMPLE

To illustrate these ideas, let us consider a simple game called "Grundy's game". The rules of the game are as follows: Two players have in front of them a single pile of 7 matches. At each move the player must divide a pile of matches into two non-empty piles with different numbers of matches in each pile. Thus, 6 matches may be divided into piles of 5 and 1 or 4 and 2, but not 3 and 3. The first player who can no longer make a move loses the game. The following figure shows the space for a game with 7 matches and let MIN play first:



A GAME GRAPH FOR GRUNDY'S GAME

We can also use the game tree to show that, no matter what min does max can always win. A winning strategy for max is shown by heavy lines. Max, can always force the game to a win, regardless of min's first move. Min could win only if max played foolishly.

6.8. ALPHA-BETA PRUNING

The minimax pursues all branches in the space, including many that can be ignored or pruned by a more intelligent algorithm. Researchers in game playing have developed a class of search technique called Alpha-beta pruning to improve the efficiency of search in two-person games.

The idea for alpha-beta search is simple: Two values called **alpha** and **beta** are created during the search. The alpha value, associated with MAX nodes, can never decrease, and the beta value, associated with MIN nodes, can never increase.

Suppose for example, MAX node's alpha value is 6, then MAX need not consider any backedup value less than or equal to 6 that is associated with any MIN node below it. Similarly, if MIN has a beta value of 6, it need not consider further any MAX node below it that has a value of 6 or more.

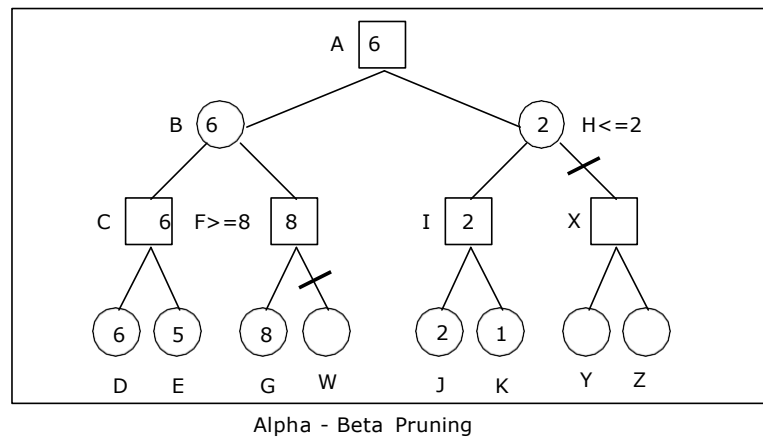
Because of these constraints, we can state the following two rules for terminating the search:

1. Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX node ancestors. The final backedup value of this MIN node can then be set to its beta value.
2. Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN node ancestors. The final backedup value of this MAX node can then be set to its alpha value.

Figure given below shows an example of alpha-beta pruning. The search proceeds depth-first to minimise the memory requirement, and only evaluates a node when necessary. After statically evaluating nodes D and E to 6 and 5, respectively, we back up their maximum value, 6 as the value of node C. After statically evaluating node G as 8, we know that the backed up value of node F must be greater than or equal to 8, since it is the maximum of 8 and the unknown value node W. The value of node B must be 6 then, because it is the maximum of 6 and a value that must be greater than or equal to 8. Since we have exactly determined the value of node B, we do not need to evaluate or even generate node W. This is called an **ALPHA CUTOFF**.

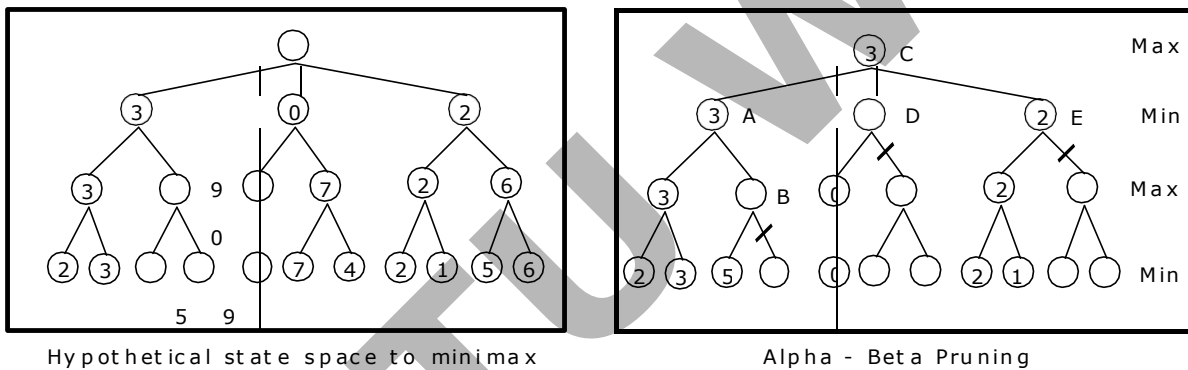
Similarly, after statically evaluating nodes J and K to 2 and 1, the backed up value is their maximum or 2. This tells that the backed up value of node H must be less than or equal to 2, since it is the minimum of 2 and the unknown value of node X. Since the value of node A is the maximum of 6 and a value that must be less than or equal to 2, it must be 6, and hence we have evaluated the root of the tree without generating or evaluating the nodes X, Y or Z. This is called **BETA CUTOFF**.

The whole process of keeping track of alpha and beta values and making cutoff's when possible is called as **alpha-beta procedure**.



6.8.1. EXAMPLE 2:

Taking the space of figure as shown below and when alpha-beta pruning applied is on this problem, is as follows:



A has $\beta = 3$ (A will be no larger than 3).
 B is β pruned, since $5 > 3$.
 C has $\alpha = 3$ (C will be no smaller than 3).
 D is α pruned, since $0 < 3$.
 E is α pruned, since $2 < 3$.
 C is 3.

AND/OR GRAPH:

And/or graph is a specialization of hypergraph which connects nodes by sets of arcs rather than by a single arcs. A hypergraph is defined as follows:

A hypergraph consists of:

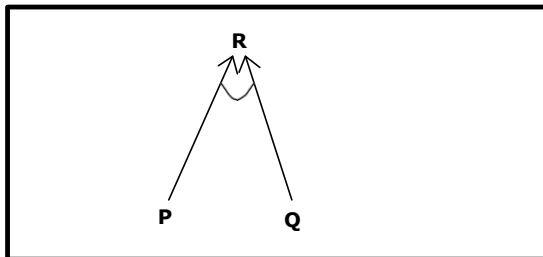
N , a set of nodes,

H , a set of hyperarcs defined by ordered pairs, in which the first implement of the pair is a node of N and the second implement is the subset of N .

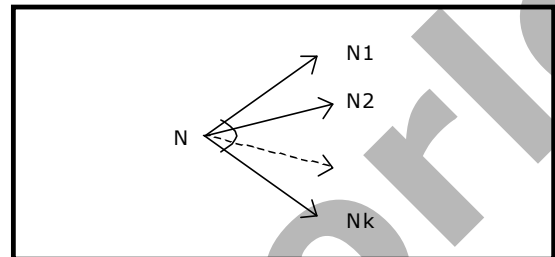
An ordinary graph is a special case of hypergraph in which all the sets of decendent nodes have a cardinality of 1.

Hyperarcs also known as K-connectors, where K is the cardinality of the set of decendent nodes. If $K = 1$, the descendent may be thought of as an OR nodes. If $K > 1$, the elements of the set of decendents may be thought of as AND nodes. In this case the connector is drawn with individual edges from the parent node to each of the decendent nodes; these individual edges are then joined with a curved link.

And/or graph for the expression P and Q \rightarrow R is follows:



Expression for P and Q \rightarrow R



A K-Connector

The K-connector is represented as a fan of arrows with a single tie is shown above.

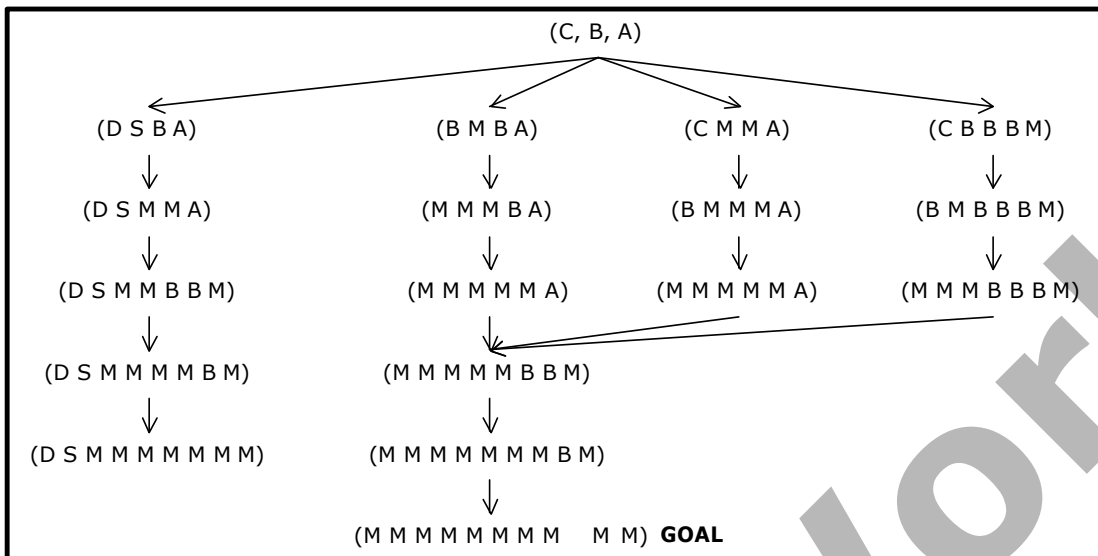
The and/or graphs consists of nodes labelled by global databases. Nodes labelled by compound databases have sets of successor nodes. These successor nodes are called AND nodes, in order to process the compound database to termination, all the compound databases must be processed to termination.

For example consider, consider a boy who collects stamps (M). He has for the purpose of exchange a winning conker (C), a bat (B) and a small toy animal (A). In his class there are friends who are also keen collectors of different items and will make the following exchanges.

1. 1 winning conker (C) for a comic (D) and a bag of sweets (S).
2. 1 winning conker (C) for a bat (B) and a stamp (M).
3. 1 bat (B) for two stamps (M, M).
4. 1 small toy animal (A) for two bats (B, B) and a stamp (M).

The problem is how to carry out the exchanges so that all his exchangeable items are converted into stamps (M). This task can be expressed more briefly as:

1. Initial state = (C, B, A)
2. Transformation rules:
 - a. If C then (D, S)
 - b. If C then (B, M)
 - c. If B then (M, M)
 - d. If A then (B, B, M)
3. The goal state is to left with only stamps (M, , M)



Expansion for the exchange problem using OR connectors only

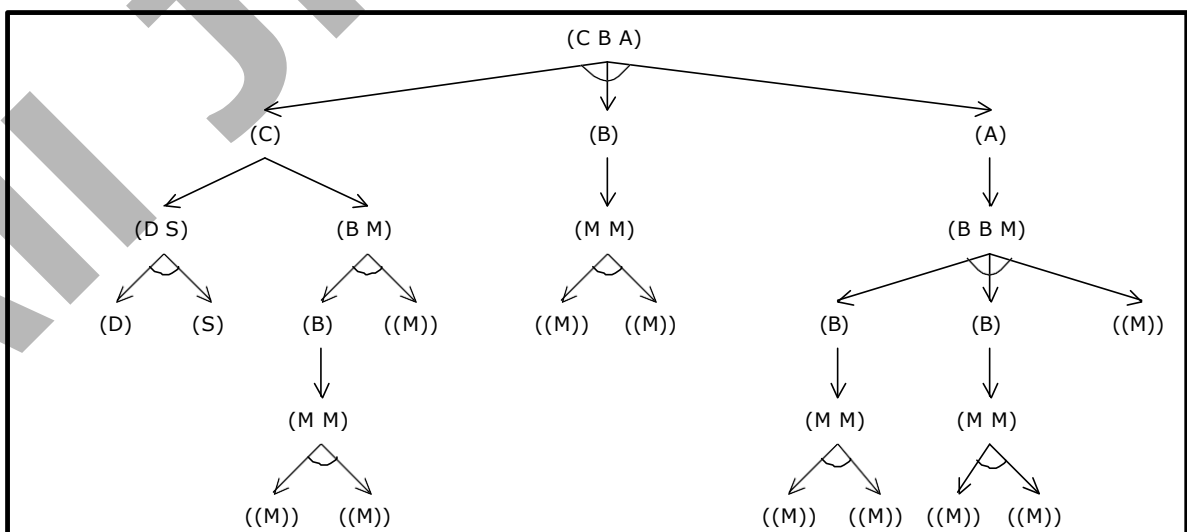
The figure shows that, a lot of extra work is done by redoing many of the transformations. This repetition can be avoided by decomposing the problem into subproblems. There are two major ways to order the components:

1. The components can either be arranged in some fixed order at the time they are generated (or).
2. They can be dynamically reordered during processing.

The more flexible system is to reorder dynamically as the processing unfolds. It can be represented by and/or graph. The solution to the exchange problem will be:

Swap conker for a bat and a stamp, then exchange this bat for two stamps. Swap his own bat for two more stamps, and finally swap the small toy animal for two bats and a stamp. The two bats can be exchanged for two stamps.

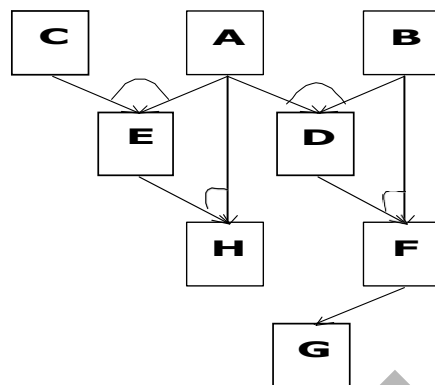
The previous exchange problem, when implemented as an and/or graph looks as follows:



Example 1:

Draw an And/Or graph for the following prepositions:

1. A
2. B
3. C
4. $A \wedge B \rightarrow D$
5. $A \wedge C \rightarrow E$
6. $B \wedge D \rightarrow F$
7. $F \rightarrow G$
8. $A \wedge E \rightarrow H$



Chapter 7

BACKTRACKING

General Method:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n -tuple (x_1, \dots, x_n) where each $x_i \in S$, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function $P(x_1, \dots, x_n)$. Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I .

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i 's must relate to each other.

- For 8-queens problem:

Explicit constraints using 8-tuple formation, for this problem are $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

Terminology:

Problem state is each node in the depth first search tree.

Solution states are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

State space is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Branch and Bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

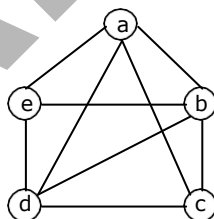
Depth first node generation with bounding functions is called **backtracking**. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

Planar Graphs:

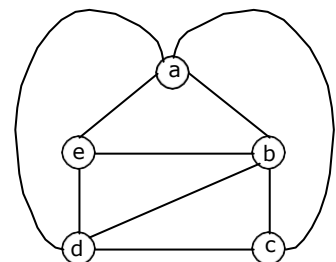
When drawing a graph on a piece of a paper, we often find it convenient to permit edges to intersect at points other than at vertices of the graph. These points of intersections are called crossovers.

A graph G is said to be planar if it can be drawn on a plane without any crossovers; otherwise G is said to be non-planar i.e., A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.

Example:



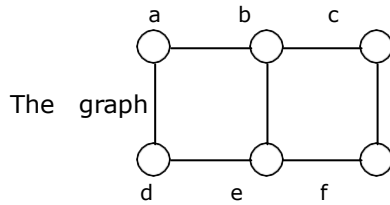
the following graph can be redrawn without crossovers as follows:



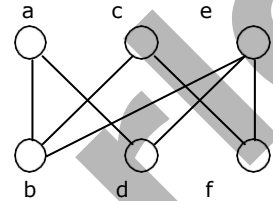
Bipartite Graph:

A bipartite graph is a non-directed graph whose set of vertices can be partitioned into two sets V_1 and V_2 (i.e. $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$) so that every edge has one end in V_1 and the other in V_2 . That is, vertices in V_1 are only adjacent to those in V_2 and vice-versa.

Example:



is bipartite. We can redraw it as



The vertex set $V = \{a, b, c, d, e, f\}$ has been partitioned into $V_1 = \{a, c, e\}$ and $V_2 = \{b, d, f\}$. The complete bipartite graph for which $V_1 = n$ and $V_2 = m$ is denoted $K_{n,m}$.

N-Queens Problem:

Let us consider, $N = 8$. Then 8-Queens Problem is to place eight queens on an 8×8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples (x_1, \dots, x_8) , where x_i is the column of the i^{th} row where the i^{th} queen is placed.

The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples.

The implicit constraints for this problem are that no two x_i 's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

This realization reduces the size of the solution space from 8^8 tuples to $8!$ Tuples.

The promising function must check whether two queens are in the same column or diagonal:

Suppose two queens are placed at positions (i, j) and (k, l) Then:

- Column Conflicts: Two queens conflict if their x_i values are identical.
- Diag 45 conflict: Two queens i and j are on the same 45° diagonal if:

$$i - j = k - l.$$

This implies, $j - l = i - k$

- Diag 135 conflict:

$$i + j = k + l.$$

This implies, $j - l = k - i$

Therefore, two queens lie on the same diagonal if and only if:

$$|j - l| = |i - k|$$

Where, j be the column of object in row i for the i^{th} queen and l be the column of object in row ' k ' for the k^{th} queen.

To check the diagonal clashes, let us take the following tile configuration:

	*						
				*			
*							
							*
			*				
						*	
		*					
				*			

In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

Let us consider for the 3rd row and 8th row case whether the queens on are conflicting or not. In this case $(i, j) = (3, 1)$ and $(k, l) = (8, 6)$. Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8| \\ \Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.

						*	
				*			
		*					
*							

Step 1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less than 8, repeat Step 1.

Step 3:

If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

1	2	3	4	5	6	7	8	Remarks
7	5	3	1					
7	5	3	1*	2*				$ j - i = 1 - 2 = 1$ $ i - k = 4 - 5 = 1$
7	5	3	1	4				
7*	5	3	1	4	2*			$ j - i = 7 - 2 = 5$ $ i - k = 1 - 6 = 5$
7	5	3*	1	4	6*			$ j - i = 3 - 6 = 3$ $ i - k = 3 - 6 = 3$
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		$ j - i = 4 - 2 = 2$ $ i - k = 5 - 7 = 2$
7	5	3	1	4*	8	6*		$ j - i = 4 - 6 = 2$ $ i - k = 5 - 7 = 2$
7	5	3	1	4	8			<i>Backtrack</i>
7	5	3	1	4				<i>Backtrack</i>
7	5	3	1	6				
7*	5	3	1	6	2*			$ j - i = 1 - 2 = 1$ $ i - k = 7 - 6 = 1$
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	$ j - i = 3 - 8 = 5$ $ i - k = 3 - 8 = 5$
7	5	3	1	6	4	2		<i>Backtrack</i>
7	5	3	1	6	4			<i>Backtrack</i>
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	SOLUTION

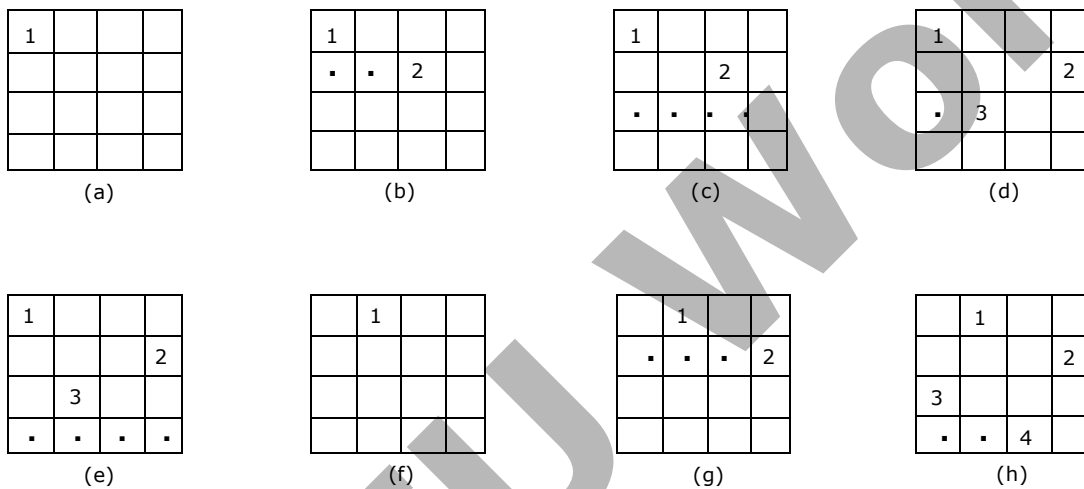
* indicates conflicting queens.

On a chessboard, the **solution** will look like:

						*	
				*			
		*					
*							
					*		
						*	
	*						
			*				

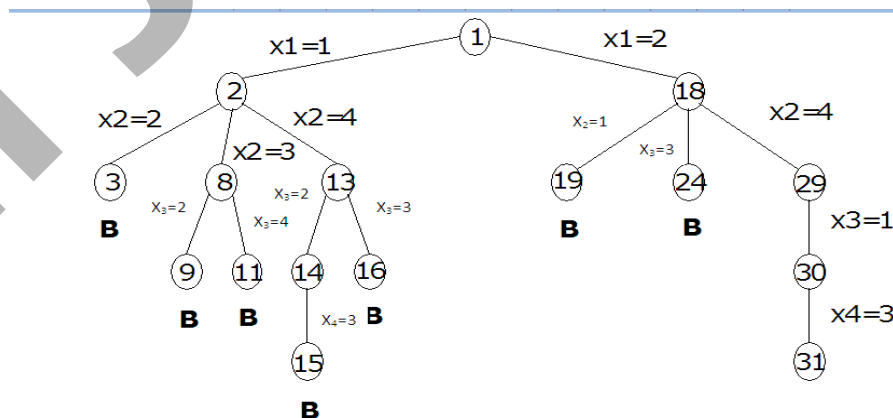
4 – Queens Problem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:



The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking

Complexity Analysis:

$$1 + n + n^2 + n^3 + \dots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

For the instance in which $n = 8$, the state space tree contains:

$$\frac{8^{8+1} - 1}{8 - 1} = 19, 173, 961 \text{ nodes}$$

Program for N-Queens Problem:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

int x[10] = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5};

place (int k)
{
    int i;
    for (i=1; i < k; i++)
    {
        if ((x[i] == x[k]) || (abs (x[i] - x[k]) == abs (i - k)))
            return (0);
    }
    return (1);
}

nqueen (int n)
{
    int m, k, i = 0;
    x[1] = 0;
    k = 1;
    while (k > 0)
    {
        x[k] = x[k] + 1;
        while ((x[k] <= n) && (!place (k)))
            x[k] = x[k] + 1;
        if (x[k] <= n)
        {
            if (k == n)
            {
                i++;
                printf ("\ncombination; %d\n", i);
                for (m=1; m<=n; m++)
                    printf ("row = %3d\t column=%3d\n", m, x[m]);
                getch();
            }
            else
            {
                k++;
                x[k]=0;
            }
        }
        else
            k--;
    }
    return (0);
}
```

```

    }
    main ()
    {
        int n;
        clrscr ();
        printf ("enter value for N: ");
        scanf ("%d", &n);
        nqueen (n);
    }

```

Output:

Enter the value for N: 4

Combination: 1

Row = 1 column = 2
 Row = 2 column = 4
 Row = 3 column = 1
 Row = 4 column = 3

Combination: 2

3
 1
 4
 2

For N = 8, there will be 92 combinations.

Sum of Subsets:

Given positive numbers w_i , $1 \leq i \leq n$, and m , this problem requires finding all subsets of w_i whose sums are m .

All solutions are k -tuples, $1 \leq k \leq n$.

Explicit constraints:

- $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$.

Implicit constraints:

- No two x_i can be the same.
- The sum of the corresponding w_i 's be m .
- $x_i < x_{i+1}$, $1 \leq i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

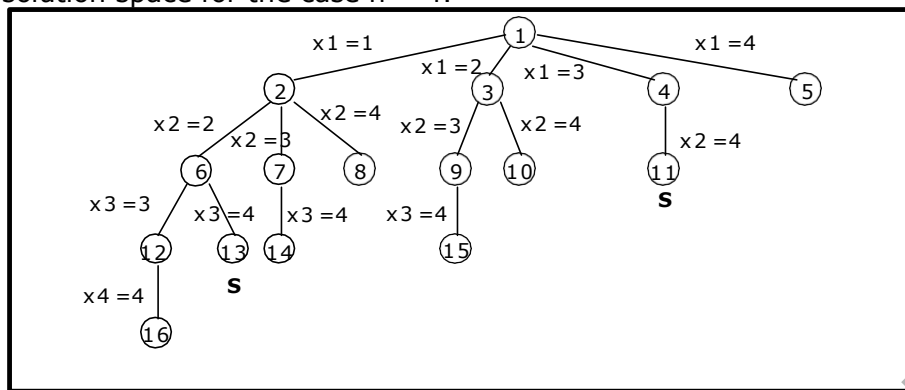
A better formulation of the problem is where the solution subset is represented by an n -tuple (x_1, \dots, x_n) such that $x_i \in \{0, 1\}$.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1).

For both the above formulations, the solution space is 2^n distinct tuples.

For example, $n = 4$, $w = (11, 13, 24, 7)$ and $m = 31$, the desired subsets are (11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case $n = 4$.



A possible solution space organisation for the sum of the subsets problem.

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level $i+1$ node represents a value for x_i . At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1) , $(1, 2)$, $(1, 2, 3)$, $(1, 2, 3, 4)$, $(1, 2, 4)$, $(1, 3, 4)$, (2) , $(2, 3)$, and so on. Thus, the left most sub-tree defines all subsets containing w_1 , the next sub-tree defines all subsets containing w_2 but not w_1 , and so on.

Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m -colorability decision problem. The m -colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m -coloring will begin by first assigning the graph to its adjacency matrix, setting the array $x[]$ to zero. The colors are represented by the integers $1, 2, \dots, m$ and the solutions are given by the n -tuple (x_1, x_2, \dots, x_n) , where x_i is the color of node i .

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement `mcoloring(1);`

Algorithm mcoloring (k)

// This algorithm was formed using the recursive backtracking schema. The graph is
 // represented by its Boolean adjacency matrix $G[1:n, 1:n]$. All assignments of
 // $1, 2, \dots, m$ to the vertices of the graph such that adjacent vertices are assigned
 // distinct integers are printed. k is the index of the next vertex to color.
 {

```

  repeat
  {
    NextValue (k);           // Generate all legal assignments for  $x[k]$ .
    // Assign to  $x[k]$  a legal color.
    If ( $x[k] = 0$ ) then return; // No new color possible
    If ( $k = n$ ) then           // at most  $m$  colors have been
                                // used to color the  $n$  vertices.
      write ( $x[1:n]$ );
      else mcoloring ( $k+1$ );
    } until (false);
  }

```

Algorithm NextValue (k)

// $x[1], \dots, x[k-1]$ have been assigned integer values in the range $[1, m]$ such that
 // adjacent vertices have distinct integers. A value for $x[k]$ is determined in the range
 // $[0, m]$. $x[k]$ is assigned the next highest numbered color while maintaining distinctness
 // from the adjacent vertices of vertex k . If no such color exists, then $x[k]$ is 0.
 {

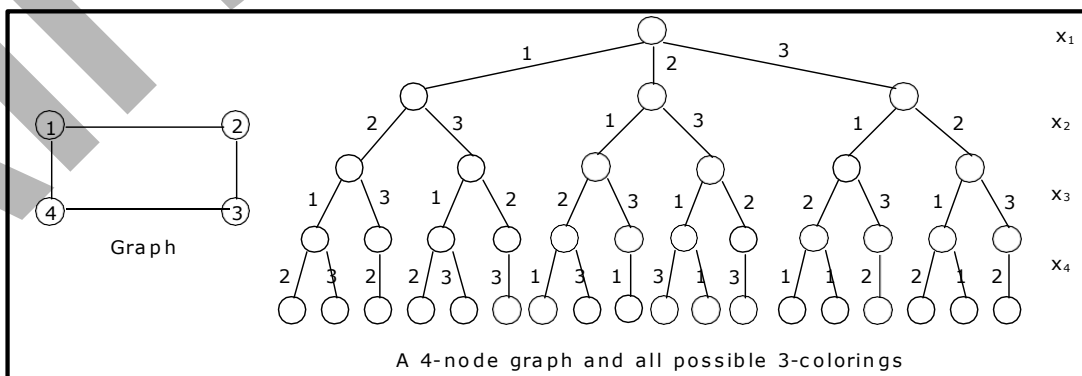
```

  repeat
  {
     $x[k] := (x[k] + 1) \bmod (m+1)$            // Next highest color.
    If ( $x[k] = 0$ ) then return;             // All colors have been used
    for  $j := 1$  to  $n$  do
    {
      // check if this color is distinct from adjacent colors
      if ( $(G[k, j] \neq 0) \text{ and } (x[k] = x[j])$ )
        // If ( $k, j$ ) is an edge and if adj. vertices have the same color.
        then break;
    }
    if ( $j = n+1$ ) then return;              // New color found
  } until (false);                          // Otherwise try to find another color.
}

```

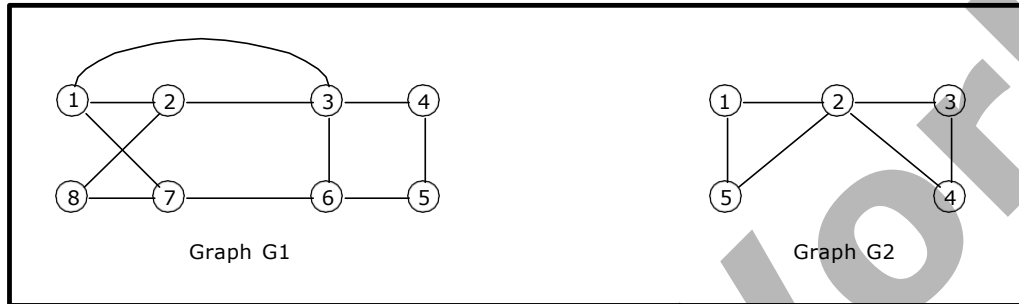
Example:

Color the graph given below with minimum number of colors by backtracking using state space tree.



Hamiltonian Cycles:

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order v_1, v_2, \dots, v_{n+1} , then the edges (v_i, v_{i+1}) are in E , $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1} , which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.



Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector (x_1, \dots, x_n) is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from x_1, x_2, \dots, x_{k-1} and v is connected by an edge to x_{k-1} . The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1 .

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ to zero and $x[1]$ to 1, and then executing $\text{Hamiltonian}(2)$.

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

Algorithm NextValue (k)

```
// x [1: k-1] is a path of k - 1 distinct vertices . If x[k] = 0, then no vertex has as yet been
// assigned to x [k]. After execution, x[k] is assigned to the next highest numbered vertex
// which does not already appear in x [1 : k - 1] and is connected by an edge to x [k - 1].
// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].
{
    repeat
    {
        x [k] := (x [k] + 1) mod (n+1);           // Next vertex.
        If (x [k] = 0) then return;
        If (G [x [k - 1], x [k]] ≠ 0) then
        {
            // Is there an edge?
            for j := 1 to k - 1 do if (x [j] = x [k]) then break;
            // check for distinctness.
            If (j = k) then
            // If true, then the vertex is distinct.
            If ((k < n) or ((k = n) and G [x [n], x [1]] ≠ 0))
            then return;
        }
    } until (false);
}
```

Algorithm Hamiltonian (k)

// This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles begin at node 1.

```
{
  repeat
  {
    NextValue (k); // Generate values for x [k].
    //Assign a legal Next value to x [k].
    if (x [k] = 0) then return;
    if (k = n) then write (x [1: n]);
    else Hamiltonian (k + 1)
  } until (false);
}
```

0/1 Knapsack:

Given n positive weights w_i , n positive profits p_i , and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of the weights such that:

$$\sum_{1 \leq i \leq n} w_i x_i \leq m \text{ and } \sum_{1 \leq i \leq n} p_i x_i \text{ is maximized.}$$

The x_i 's constitute a zero-one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i 's.

Bounding functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far, than that live node can be killed.

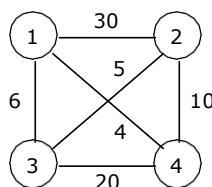
We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i , $1 \leq i \leq k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements $x_i = 0$ or 1.

(Knapsack problem using backtracking is solved in branch and bound chapter)

7.8 Traveling Sale Person (TSP) using Backtracking:

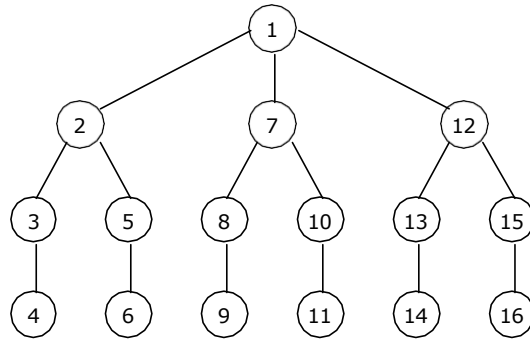
We have solved TSP problem using dynamic programming. In this section we shall solve the same problem using backtracking.

Consider the graph shown below with 4 vertices.



A graph for TSP

The solution space tree, similar to the n-queens problem is as follows:



We will assume that the starting node is 1 and the ending node is obviously 1. Then $1, \{2, \dots, 4\}, 1$ forms a tour with some cost which should be minimum. The vertices shown as $\{2, 3, \dots, 4\}$ forms a permutation of vertices which constitutes a tour. We can also start from any vertex, but the tour should end with the same vertex.

Since, the starting vertex is 1, the tree has a root node R and the remaining nodes are numbered as depth-first order. As per the tree, from node 1, which is the live node, we generate 3 branches node 2, 7 and 12. We simply come down to the left most leaf node 4, which is a valid tour $\{1, 2, 3, 4, 1\}$ with cost $30 + 5 + 20 + 4 = 59$. Currently this is the best tour found so far and we backtrack to node 3 and to 2, because we do not have any children from node 3. When node 2 becomes the E-node, we generate node 5 and then node 6. This forms the tour $\{1, 2, 4, 3, 1\}$ with cost $30 + 10 + 20 + 6 = 66$ and is discarded, as the best tour so far is 59.

Similarly, all the paths from node 1 to every leaf node in the tree is searched in a depth first manner and the best tour is saved. In our example, the tour costs are shown adjacent to each leaf nodes. The optimal tour cost is therefore 25.

Chapter 8

Branch and Bound

General method:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.
2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node

Branch and Bound is the generalisation of both graph search strategies, BFS and D-search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).
- A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

Definition 1: Live node is a node that has been generated but whose children have not yet been generated.

Definition 2: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Definition 3: Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Definition 4: Branch-and-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Definition 5: The adjective "heuristic", means "related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of the time.

Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node is rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can be speeded by using an "intelligent" ranking function $c(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$$c(x) = f(h(x)) + g(x)$$

where, $c(x)$ is the cost of x .

$h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any non-decreasing function.

$g(x)$ is an estimate of the additional effort needed to reach an answer node from x .

A search strategy that uses a cost function $c(x) = f(h(x)) + g(x)$ to select the next E-node would always choose for its next E-node a live node with least $c(\cdot)$ is called a LC-search (Least Cost search)

BFS and D-search are special cases of LC-search. If $g(x) = 0$ and $f(h(x)) = \text{level of node } x$, then an LC search generates nodes by levels. This is eventually the same as a BFS. If $f(h(x)) = 0$ and $g(x) > g(y)$ whenever y is a child of x , then the search is essentially a D-search.

An LC-search coupled with bounding functions is called an LC-branch and bound search

We associate a cost $c(x)$ with each node x in the state space tree. It is not possible to easily compute the function $c(x)$. So we compute a estimate $\hat{c}(x)$ of $c(x)$.

Control Abstraction for LC-Search:

Let t be a state space tree and $c()$ a cost function for the nodes in t . If x is a node in t , then $c(x)$ is the minimum cost of any answer node in the subtree with root x . Thus, $c(t)$ is the cost of a minimum-cost answer node in t .

A heuristic $\hat{c}(\cdot)$ is used to estimate $c()$. This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then $c(x) = \hat{c}(x)$.

LC-search uses \hat{c} to find an answer node. The algorithm uses two functions $\text{Least}()$ and $\text{Add}()$ to delete and add a live node from or to the list of live nodes, respectively.

$\text{Least}()$ finds a live node with least $\hat{c}()$. This node is deleted from the list of live nodes and returned.

Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

Listnode = **record**

```
{
    Listnode * next, *parent; float cost;
}
```

Algorithm **LCSearch**(t)

```
{
    //Search t for an answer node
    if *t is an answer node then output *t and return;
    E := t;          //E-node.
    initialize the list of live nodes to be empty;
    repeat
    {
        for each child x of E do
        {
            if x is an answer node then output the path from x to t and return;
            Add (x);          //x is a new live node.
            (x → parent) := E;    // pointer for path to root
        }
        if there are no more live nodes then
        {
            write ("No answer node");
            return;
        }
        E := Least();
    } until (false);
}
```

The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E. When all the children of E have been generated, E becomes a dead node. This happens only if none of E's children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

Bounding:

A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost $c(x)$ associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. The three search methods differ only in the selection rule used to obtain the next E-node.

A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.

A cost function $c(.)$ such that $c(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x . If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $c(x) \geq c(x) > \text{upper}$. The starting value for upper can be obtained by some heuristic or can be set to ∞ .

As long as the initial value for upper is not less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of upper can be updated.

Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.

To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function $c(.)$, such that $c(x)$ is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for $c(.)$.

- For nodes representing feasible solutions, $c(x)$ is the value of the objective function for that feasible solution.
- For nodes representing infeasible solutions, $c(x) = \infty$.
- For nodes representing partial solutions, $c(x)$ is the cost of the minimum-cost node in the subtree with root x .

Since, $c(x)$ is generally hard to compute, the branch-and-bound algorithm will use an estimate $c(x)$ such that $c(x) \leq c(x)$ for all x .

The 15 – Puzzle Problem:

The 15 puzzle is to search the state space for the goal state and use the path from the initial state to the goal state as the answer. There are $16!$ ($16! \approx 20.9 \times 10^{12}$) different arrangements of the tiles on the frame.

As the state space for the problem is very large it would be worthwhile to determine whether the goal state is reachable from the initial state. Number the frame positions 1 to 16.

Position i is the frame position containing tile numbered i in the goal arrangement of Figure 8.1(b). Position 16 is the empty spot. Let $\text{position}(i)$ be the position number in the initial state of the tile number i . Then $\text{position}(16)$ will denote the position of the empty spot.

For any state let:

less(i) be the number of tiles j such that $j < i$ and $\text{position}(j) > \text{position}(i)$.

The goal state is reachable from the initial state iff: $\sum_{i=1}^{16} \text{less}(i) + x$ is even.

Here, $x = 1$ if in the initial state the empty spot is at one of the shaded positions of figure 8.1(c) and $x = 0$ if it is at one of the remaining positions.

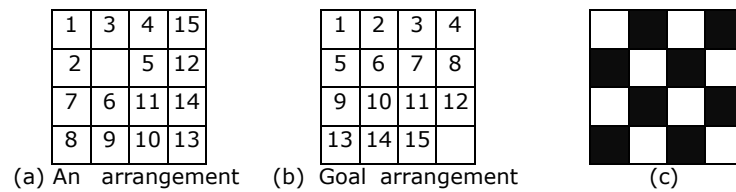


Figure 8.1. 15-puzzle arrangement

Example 1:

For the state of Figure 8.1(a) we have less(i) values as follows:

less(1) = 0	less(2) = 0	less(3) = 1	less(4) = 1
less(5) = 0	less(6) = 0	less(7) = 1	less(8) = 0
less(9) = 0	less(10) = 0	less(11) = 3	less(12) = 6
less(13) = 0	less(14) = 4	less(15) = 11	less(16) = 10

Therefore, $\sum_{i=1}^{16} less(i) + x = (0 + 0 + 1 + 1 + 0 + 0 + 1 + 0 + 0 + 0 + 3 + 6 + 0 + 4 + 11 + 10) + 0 = 37 + 0 = 37$.

Hence, goal is *not reachable*.

Example 2:

For the root state of Figure 8.2 we have less(i) values are as follows:

less(1) = 0	less(2) = 0	less(3) = 0	less(4) = 0
less(5) = 0	less(6) = 0	less(7) = 0	less(8) = 1
less(9) = 1	less(10) = 1	less(11) = 0	less(12) = 0
less(13) = 1	less(14) = 1	less(15) = 1	less(16) = 9

Therefore, $\sum_{i=1}^{16} less(i) + x = (0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 + 1 + 1 + 0 + 0 + 1 + 1 + 1 + 9) + 1 = 15 + 1 = 16$.

Hence, goal is *reachable*.

LC Search for 15 Puzzle Problem:

A depth first state space tree generation will result in the subtree of Figure 8.3 when the next moves are attempted in the order: move the empty space up, right, down and left. The search of the state space tree is blind. It will take the leftmost path from the root regardless of the starting configuration. As a result, the answer node may never be found.

A breadth first search will always find a goal node nearest to the root. However, such a search is also blind in the sense that no matter what the initial configuration, the algorithm attempts to make the same sequence of moves.

We need a more intelligent search method. We associate a cost $c(x)$ with each node x in the state space tree. The cost $c(x)$ is the length of a path from the root to a nearest goal node in the subtree with root x . The easy to compute estimate $c(x)$ of $c(x)$ is as follows:

$$c(x) = f(x) + g(x)$$

where, $f(x)$ is the length of the path from the root to node x and

$g(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x . Here, $g(x)$ is the number of nonblank tiles not in their goal position.

An LC-search of Figure 8.2, begin with the root as the E-node and generate all child nodes 2, 3, 4 and 5. The next node to become the E-node is a live node with least $c(x)$.

$$c(2) = 1 + 4 = 5$$

$$c(3) = 1 + 4 = 5$$

$$c(4) = 1 + 2 = 3 \text{ and}$$

$$c(5) = 1 + 4 = 5.$$

Node 4 becomes the E-node and its children are generated. The live nodes at this time are 2, 3, 5, 10, 11 and 12. So:

$$c(10) = 2 + 1 = 3$$

$$c(11) = 2 + 3 = 5 \text{ and}$$

$$c(12) = 2 + 3 = 5.$$

The live node with least c is node 10. This becomes the next E-node. Nodes 22 and 23 are generated next. Node 23 is the goal node, so search terminates.

LC-search was almost as efficient as using the exact function $c()$, with a suitable choice for $c()$, LC-search will be far more selective than any of the other search methods.

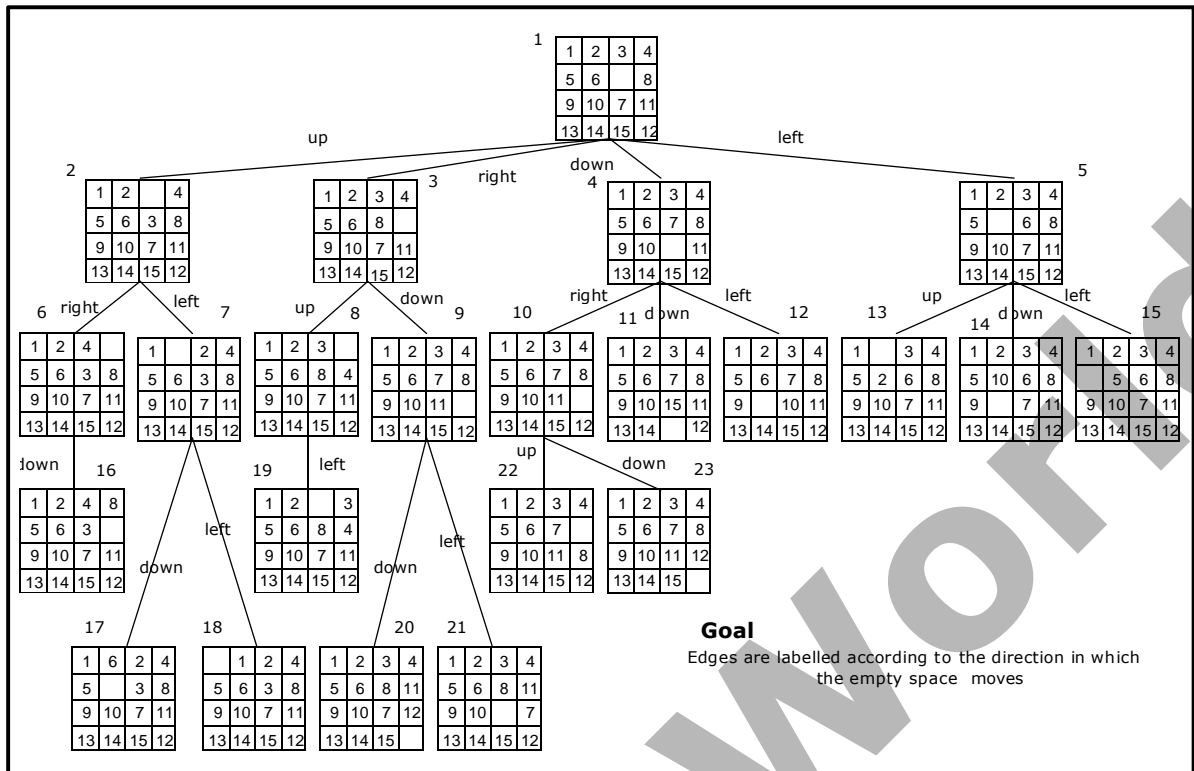


Figure 8.2. Part of the state space tree for 15-puzzle problem

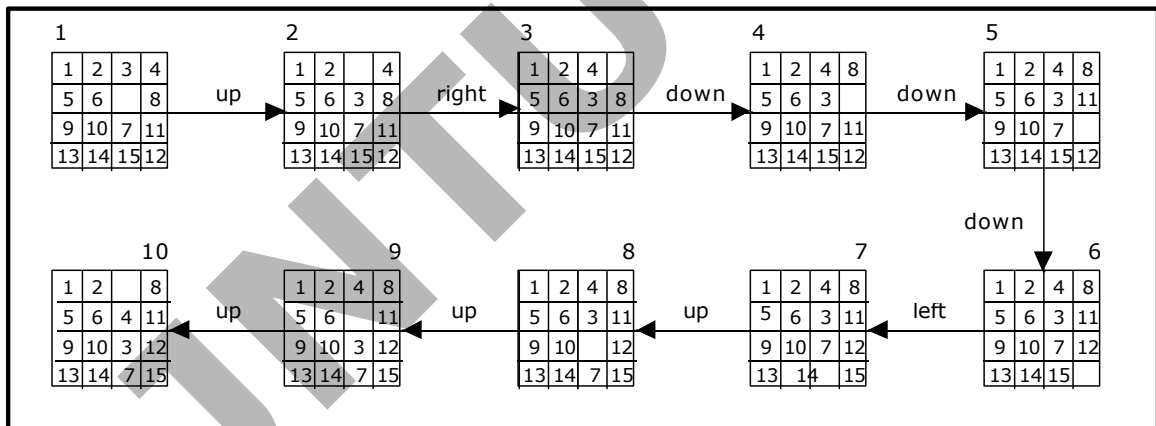


Figure 8. 3. First ten steps in a depth first search

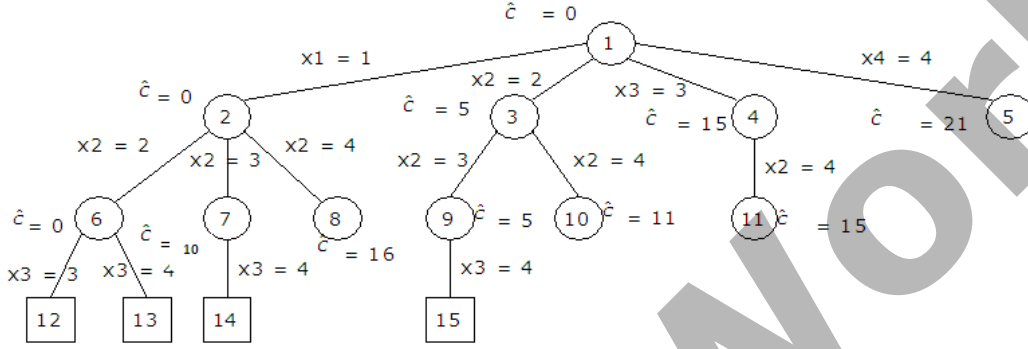
Job sequencing with deadlines:

We are given n jobs and one processor. Each job i is associated with it a three tuple (p_i, d_i, t_i) . Job i requires t_i units of processing time. If its processing is not completed by the deadline d_i , then a penalty p_i is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in J . The subset J should be such that the penalty incurred is minimum among all possible subsets J . Such a J is optimal.

Example:

Consider the following instance with $n = 4$, $(p_1, d_1, t_1) = (5, 1, 1)$, $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6, 2, 1)$ and $(p_4, d_4, t_4) = (3, 1, 1)$.

The solution space for this instance consists of all possible subsets of the job index set $\{1, 2, 3, 4\}$. The solution space can be organized into a tree. Figure 8.4 corresponds to the variable tuple size formulation. In figure square nodes represent infeasible subsets and all non-square nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node. For this node $J=\{2, 3\}$ and the penalty is 8.



The cost function $c(x)$ for any circular node x is the minimum penalty corresponding to any node in the subtree with root x .

The value of $c(x) = \infty$ for a square node.

Let S_x be the subset of jobs selected for J at node x .

If $m = \max \{ i / i \in S_x \}$, then $\hat{c}(x) = \sum_{\substack{i < m \\ i \notin S_x}} p_i$ is an estimate for $c(x)$ with the property $\hat{c}(x) \leq c(x)$.

An upper bound $u(x)$ on the cost of a minimum-cost answer node in the subtree x is $u(x) = \sum_{i \in S_x} p_i$. The $u(x)$ is the cost of the solution S_x corresponding to node x .

$S(2) = \{1\}$	$m = 1$	$\hat{c}(2) = \sum_{\substack{i < m \\ i \notin S_x}} p_i = 0$
$S(3) = \{2\}$	$m = 2$	$\hat{c}(3) = \sum_{i < 2} p_i = \sum_{i=1} p_i = 5$
$S(4) = \{3\}$	$m = 3$	$\hat{c}(4) = \sum_{i < 3} p_i = \sum_{i=1,2} p_i = p_1 + p_2 = 5 + 10 = 15$
$S(5) = \{4\}$	$m = 4$	$\hat{c}(5) = \sum_{i < 4} p_i = \sum_{i=1,2,3} p_i = p_1 + p_2 + p_3 = 5 + 10 + 6 = 21$
$S(6) = \{1, 2\}$	$m = 2$	$\hat{c}(6) = \sum_{\substack{i=1,2 \\ i \in S_x}} p_i = \sum_{\substack{i < 1 \\ i \notin S(6)}} p_i = 0$
$S(7) = \{1, 3\}$	$m = 3$	$\hat{c}(7) = \sum_{\substack{i < 3 \\ i \notin S(7)}} p_i = p_2 = 10$

$S(8) = \{1, 4\}$	$m = 4$	$c(8) = \sum_{\substack{i \leq 4 \\ i \notin S(8)}} p_i = p_2 + p_3 = 10 + 6 = 16$
$S(9) = \{2, 3\}$	$m = 3$	$c(9) = 5$
$S(10) = \{2, 4\}$	$m = 3$	$c(10) = 11$
$S(11) = \{3, 4\}$	$m = 4$	$c(11) = 15$

Calculation of the Upper bound $u(x) = \sum_{i \notin S_x} p_i$

$$U(1) = 0$$

$$U(2) = \sum_{i \notin S(2)} p_i = p_2 + p_3 + p_4 = 10 + 6 + 3 = 19$$

eliminate job 1

$$U(3) = p_1 + p_3 + p_4 = 5 + 6 + 3 = 14$$

eliminate job 2

$$U(4) = p_1 + p_2 + p_4 = 5 + 10 + 3 = 18$$

eliminate job 3

$$U(5) = p_1 + p_2 + p_4 = 5 + 10 + 6 = 21$$

eliminate job 4

$$U(6) = p_3 + p_4 = 6 + 3 = 9$$

eliminate jobs 1, 2

$$U(7) = p_2 + p_4 = 10 + 3 = 13$$

eliminate jobs 1, 3

$$U(8) = p_2 + p_3 = 10 + 6 = 16$$

eliminate jobs 1, 4

$$U(9) = p_1 + p_4 = 5 + 3 = 8$$

eliminate jobs 2, 3

$$U(10) = p_1 + p_3 = 5 + 6 = 11$$

eliminate jobs 2, 4

$$U(11) = p_1 + p_2 = 5 + 10 = 15$$

eliminate jobs 3, 4

FIFO Branch and Bound:

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with upper = ∞ as an upper bound on the cost of a minimum-cost answer node.

Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.4, nodes 2, 3, 4, and 5 are generated. Then $u(2) = 19$, $u(3) = 14$, $u(4) = 18$, and $u(5) = 21$.

The variable upper is updated to 14 when node 3 is generated. Since $c(4)$ and $c(5)$ are greater than upper, nodes 4 and 5 get killed. Only nodes 2 and 3 remain alive.

Node 2 becomes the next E-node. Its children, nodes 6, 7 and 8 are generated. Then $u(6) = 9$ and so upper is updated to 9. The cost $c(7) = 10 > \text{upper}$ and node 7 gets killed. Node 8 is infeasible and so it is killed.

Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then $u(9) = 8$ and so upper becomes 8. The cost $c(10) = 11 > \text{upper}$, and this node is killed.

The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $c(x) > \text{upper}$ each time upper is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $c(x) > \text{upper}$ are distributed in some random way in the queue. Instead, live nodes with $c(x) > \text{upper}$ can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate $c(.)$ and $u(.)$ is called FIFOBB.

LC Branch and Bound:

An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with $\text{upper} = \infty$ and node 1 as the first E-node.

When node 1 is expanded, nodes 2, 3, 4 and 5 are generated in that order.

As in the case of FIFOBB, upper is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $c(4) > \text{upper}$ and $c(5) > \text{upper}$.

Node 2 is the next E-node as $c(2) = 0$ and $c(3) = 5$. Nodes 6, 7 and 8 are generated and upper is updated to 9 when node 6 is generated. So, node 7 is killed as $c(7) = 10 > \text{upper}$. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6.

Node 6 is the next E-node as $c(6) = 0 < c(3)$. Both its children are infeasible.

Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as $u(9) = 8$. So, node 10 with $c(10) = 11$ is killed on generation.

Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

The path $\overset{2}{} \rightarrow \overset{3}{} \rightarrow 9 = 5 + 3 = 8$

Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let $G = (V, E)$ is a connected graph. Let $C(i, j)$ be the cost of edge $\langle i, j \rangle$. $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$ and let $|V| = n$, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by $S = \{1, \pi, 1 \mid \pi \text{ is a}$

permutation of $(2, 3, \dots, n)$ and $|S| = (n - 1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \dots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n - 1$ and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:
 - a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.
 - b) Find the sum of elements, which were subtracted from rows.
 - c) Apply column reductions for the matrix obtained after row reduction.

Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.
 - d) Find the sum of elements, which were subtracted from columns.
 - e) Obtain the cumulative sum of row wise reduction and column wise reduction.

Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

Associate the cumulative reduced sum to the starting state as lower bound and ∞ as upper bound.
2. Calculate the reduced cost matrix for every node R . Let A is the reduced cost matrix for node R . Let S be a child of R such that the tree edge (R, S) corresponds to including edge $\langle i, j \rangle$ in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:
 - a) Change all entries in row i and column j of A to ∞ .
 - b) Set $A(j, 1)$ to ∞ .
 - c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞ . Let r is the total amount subtracted to reduce the matrix.
 - c) Find $\bar{c}(S) = \bar{c}(R) + A(i, j) + r$, where ' r ' is the total amount subtracted to reduce the matrix, $\bar{c}(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $\bar{c}(S)$ is called the cost function.
3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows:

The cost matrix is
$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Step 1: Find the reduced cost matrix.

Apply row reduction method:

*Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.*

The resulting row wise reduced cost matrix =
$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Row wise reduction sum = $10 + 2 + 2 + 3 + 4 = 21$

Now apply column reduction for the above matrix:

*Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 3rd column.*

The resulting column wise reduced cost matrix (A) =
$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Column wise reduction sum = $1 + 0 + 3 + 0 + 0 = 4$

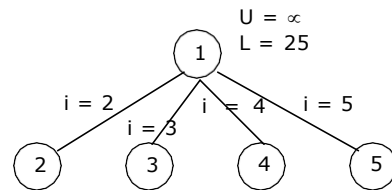
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= $21 + 4 = 25$.

This is the cost of a root i.e., node 1, because this is the initially reduced cost matrix.

The lower bound for node is 25 and upper bound is ∞ .

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1, 5).

The tree organization up to this point is as follows:



Variable 'i' indicates the next node to visit.

Step 2:

Consider the path (1, 2):

Change all entries of row 1 and column 2 of A to ∞ and also set A(2, 1) to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

$$\text{Row reduction sum} = 0 + 0 + 0 + 0 = 0$$

$$\text{Column reduction sum} = 0 + 0 + 0 + 0 = 0$$

$$\text{Cumulative reduction (r)} = 0 + 0 = 0$$

$$\text{Therefore, as } c(S) = c(R) + A(1, 2) + r$$

$$c(S) = 25 + 10 + 0 = 35$$

Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to ∞ and also set A(3, 1) to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Row reduction sum = 0

Column reduction sum = 11

Cumulative reduction (r) = 0 + 11 = 11

Therefore, as $c(S) = c(R) + A(1, 3) + r$
 $c(S) = 25 + 17 + 11 = 53$

Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∞ and also set A(4, 1) to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(1, 4) + r$

$$\bar{c}(S) = 25 + 0 + 0 = 25$$

Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∞ and also set A(5, 1) to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

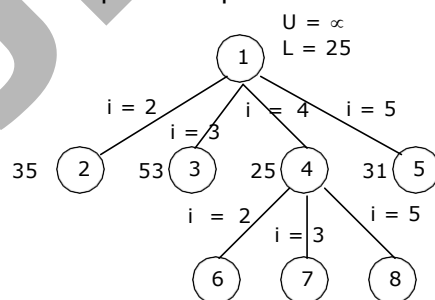
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Row reduction sum = 5
 Column reduction sum = 0
 Cumulative reduction (r) = 5 + 0 = 5

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(1, 5) + r$

$$\bar{c}(S) = 25 + 1 + 5 = 31$$

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (4, 2), (4, 3) and (4, 5).

Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to ∞ and also set A(2, 1) to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(4, 2) + r$

$$\bar{c}(S) = 25 + 3 + 0 = 28$$

Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to ∞ and also set A(3, 1) to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\left[\begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ & 1 & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \end{array} \right]$$

Row reduction sum = 2

Column reduction sum = 11

Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(4, 3) + r$

$$\bar{c}(S) = 25 + 12 + 13 = 50$$

Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to ∞ and also set A(5, 1) to ∞ .

$$\left[\begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{array} \right]$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\left[\begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ & 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{array} \right]$$

Row reduction sum = 11

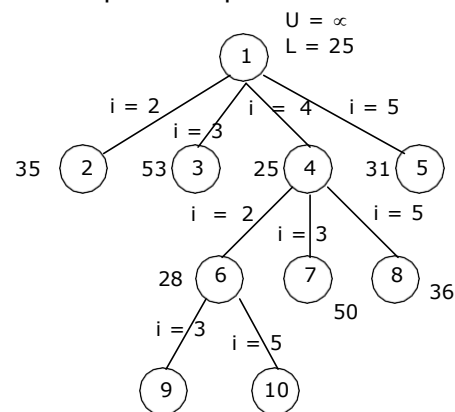
Column reduction sum = 0

Cumulative reduction (r) = 11+0 = 11

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(4, 5) + r$

$$\bar{c}(S) = 25 + 0 + 11 = 36$$

The tree organization up to this point is as follows:



The cost of the paths between $(4, 2) = 28$, $(4, 3) = 50$ and $(4, 5) = 36$. The cost of the path between $(4, 2)$ is minimum. Hence the matrix obtained for path $(4, 2)$ is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are $(2, 3)$ and $(2, 5)$.

Consider the path $(2, 3)$:

Change all entries of row 2 and column 3 of A to ∞ and also set $A(3, 1)$ to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Row reduction sum = 2

Column reduction sum = 11

Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(2, 3) + r$

$$\bar{c}(S) = 28 + 11 + 13 = 52$$

Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∞ and also set A(5, 1) to ∞ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Row reduction sum = 0

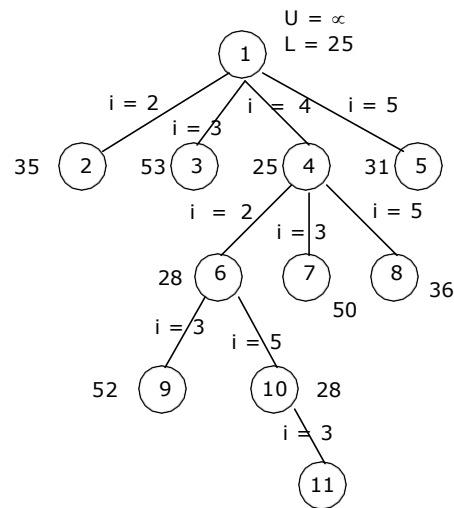
Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(2, 5) + r$

$$\bar{c}(S) = 28 + 0 + 0 = 28$$

The tree organization up to this point is as follows:



The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths is (5, 3).

Consider the path (5, 3):

Change all entries of row 5 and column 3 of A to ∞ and also set A(3, 1) to ∞ . Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞ .

Then the resultant matrix is

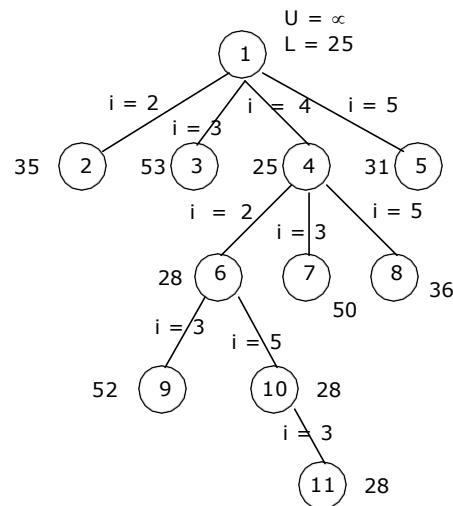
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as

$$\begin{aligned} c(S) &= c(R) + A(5, 3) + r \\ c(S) &= 28 + 0 + 0 = 28 \end{aligned}$$

The overall tree organization is as follows:



The path of traveling sale person problem is:

$1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 1$

The minimum cost of the path is: $10 + 6 + 2 + 7 + 3 = 28$.

8.9. 0/1 Knapsack Problem

Consider the instance: $M = 15$, $n = 4$, $(P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$ and $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$.

0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Place first item in knapsack. Remaining weight of knapsack is $15 - 2 = 13$. Place next item w_2 in knapsack and the remaining weight of knapsack is $13 - 4 = 9$. Place next item w_3 in knapsack then the remaining weight of knapsack is $9 - 6 = 3$. No fractions are allowed in calculation of upper bound so w_4 cannot be placed in knapsack.

$$\text{Profit} = P_1 + P_2 + P_3 = 10 + 10 + 12$$

$$\text{So, Upper bound} = 32$$

To calculate lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

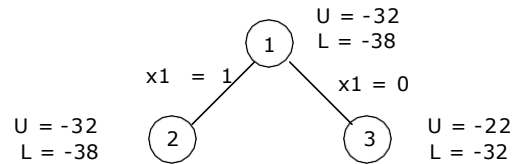
$$\text{Lower bound} = 10 + 10 + 12 + \left(\frac{3}{9} \times 18\right) = 32 + 6 = 38$$

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

$$\text{Therefore, Upper bound (U)} = -32$$

$$\text{Lower bound (L)} = -38$$

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.



Now we will calculate upper bound and lower bound for nodes 2, 3.

For node 2, $x_1 = 1$, means we should place first item in the knapsack.

$$U = 10 + 10 + 12 = 32, \text{ make it as } -32$$

$$L = 10 + 10 + 12 + \frac{3}{9} \times 18 = 32 + 6 = 38, \text{ make it as } -38$$

For node 3, $x_1 = 0$, means we should not place first item in the knapsack.

$$U = 10 + 12 = 22, \text{ make it as } -22$$

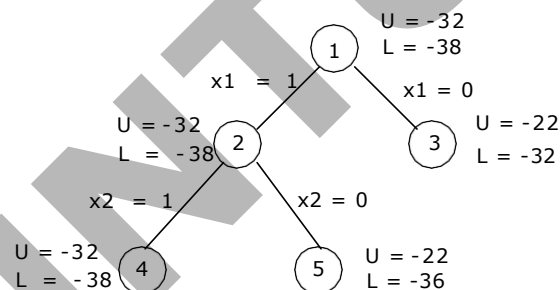
$$L = 10 + 12 + \frac{5}{9} \times 18 = 10 + 12 + 10 = 32, \text{ make it as } -32$$

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

$$\text{For node 2, } U - L = -32 + 38 = 6$$

$$\text{For node 3, } U - L = -22 + 32 = 10$$

Choose node 2, since it has minimum difference value of 6.

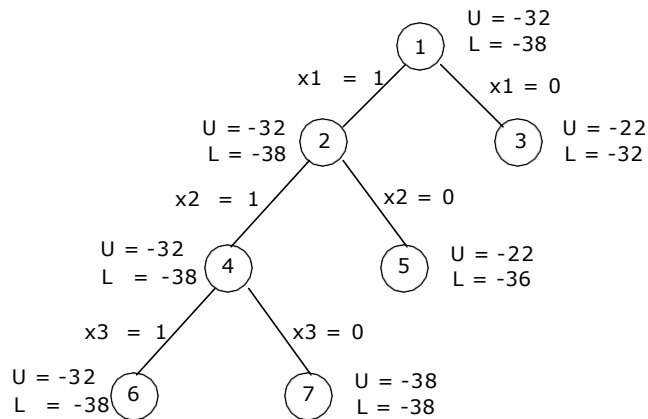


Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

$$\text{For node 4, } U - L = -32 + 38 = 6$$

$$\text{For node 5, } U - L = -22 + 36 = 14$$

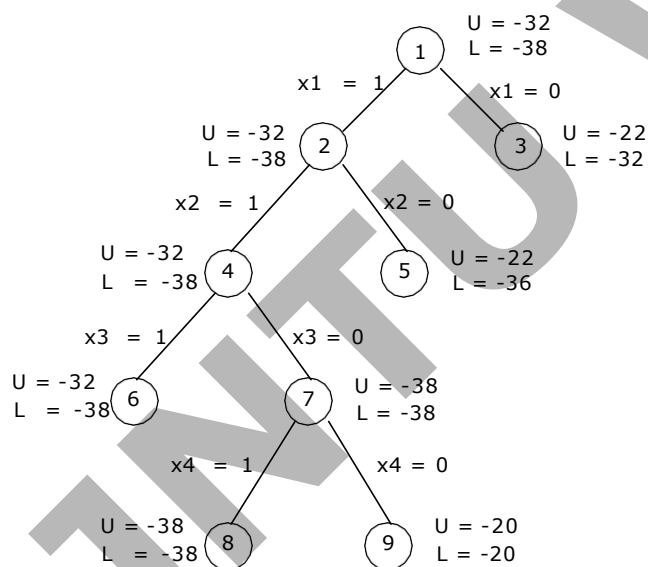
Choose node 4, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 6, $U - L = -32 + 38 = 6$
 For node 7, $U - L = -38 + 38 = 0$

Choose node 7, since it is minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 8, $U - L = -38 + 38 = 0$
 For node 9, $U - L = -20 + 20 = 0$

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from $1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 8$

$X_1 = 1$
 $X_2 = 1$
 $X_3 = 0$

$$x_4 = 1$$

The solution for 0/1 Knapsack problem is $(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

Maximum profit is:

$$\begin{aligned}\sum p_i x_i &= 10 \times 1 + 10 \times 1 + 12 \times 0 + 18 \times 1 \\ &= 10 + 10 + 18 = 38.\end{aligned}$$