Unit – 1 SYLLABUS

I.	**Introduction to Algorithms**: Algorithm Specification – Performance Analysis Growth of function – Recurrences – Randomized Algorithms
II.	**Divide and Conquer**: General method. Binary search, Finding the minimum and maximum - Merge sort - Quick sort - Selection sort - Strassen's matrix multiplication.
III.	**Greedy Strategy:** General method – Knapsack problem – Job Sequencing with deadlines.

PART – I: INTRODUCTION TO ALGORITHMS

Algorithm - Definition

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.
- An Algorithm is a finite set of instructions that if followed, accomplishes a particular task.

Characteristics of an Algorithm

An algorithm should have the following characteristics,

1. **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
2. **Input** − an algorithm should have 0 or more well-defined inputs.
3. **Output** − an algorithm should have atleast one or more well-defined outputs, and should match the desired output.
4. **Finiteness** − Algorithms must terminate after a finite number of steps.
5. **Feasibility** − should be feasible with the available resources.
6. **Independent** − an algorithm should have step-by-step directions, which should be independent of any programming code.
7. **Effectiveness** − Every instruction must be very basic so that it can be carried out, in principle, by a person using only a pencil and paper.

How to write an Algorithm?

- There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent.
- Algorithms are never written to support a particular programming code.
- As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.
- We write algorithms in a step-by-step manner, but it is not always the case.
- Algorithm writing is a process and is executed after the problem domain is well-defined.

- That is, we should know the problem domain, for which we are designing a solution.
- Algorithms tell the programmers how to code the program.
- Many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Example: Design an algorithm to add two numbers and display the result.

```
Step 1 – START
Step 2 – declare three integers a, b & c
Step 3 – define values of a & b
Step 4 – add values of a & b
Step 5 – store output of step 4 to c
Step 6 – print c
Step 7 – STOP
```

- Alternatively, the algorithm can be written as,

```
Step 1 – START ADD
Step 2 – get values of a & b
Step 3 – c ← a + b
Step 4 – display c
Step 5 – STOP
```

Algorithm – Specification

Algorithm can be described in many ways as,

1. **Natural language** like English – for this option make sure that the resulting instructions are definite.
2. **Graphical representation** called Flowchart – work well only if the algorithm is small and simple.
3. **Pseudo code** representation

Study of Algorithm

The study of algorithms includes many important and active phases as,

1. **Algorithm Design** – is a general approach to solve problems algorithmically that is applicable to a variety of problems from different areas of computing.
2. **Algorithm Validation** – to show that the algorithm computes the correct answer for all possible legal inputs.
3. **Algorithm Analysis** – (Performance analysis) refers to the task of determining how much computing time and storage an algorithm requires.
4. **Testing** – it consists of two phases: Debugging and Profiling.
   - **Debugging** – the process of executing programs on sample data sets to determine whether faulty results occur and if so, to correct them.
     - ✓ If the correctness of the output on sample data is not verified then the following strategy can be employed,

- ✓ Let more than one programmer develop programs for the same problem and compare the outputs produced by these programs. If the output matches then there is a good chance that the output is correct.
- ✓ A proof of correctness is much more valuable than tests, since it guarantees that the program will work correctly for all possible inputs.

- **Profiling** – (Performance measurement) is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.
  - ✓ It is determined in two ways, either analytically or experimentally.
  - ✓ Criteria for Measurement:
    - Space – refers to the amount of memory the program occupies.
    - Time – Execution time of the program.
  - ✓ Space Complexity is the amount of memory a program needs to run completion. The program Space = Instruction space + Data Space + Stack space
  - ✓ Time Complexity is the amount of computer time a program needs to run. It can be measured in three ways,
    - Operation Count – accounting for the time spent on the chosen operation (normally the basic operation involved in the task)
    - Step Count – account for all the time spent in all parts of the program.
    - Asymptotic Complexity.

ALGORITHM ANALYSIS

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- **A Priori Analysis** − is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed are constant and have no effect on the implementation.
- **A Posterior Analysis** − is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required are collected.

Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f (n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Both time and space efficiencies are measured as functions of the algorithm's input size.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.

The space required by an algorithm is equal to the sum of the following two components,

- A fixed part that is a space required to store certain data and variables that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S (P) of any algorithm P is S (P) = C + SP (I), where C is the fixed part and S (I) is the variable part of the algorithm, which depends on instance characteristic I.

Example:

Algorithm: SUM (A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop

Here there are three variables A, B, and C and one constant. Hence S (P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits. The time T(P) taken by a program P is the sum of the compile time and the run time. Here, we observe that T(n) grows linearly as the input size increases.

The compile time of the program does not depend on the instance characteristics of a program, as a compiled program will be run several times without recompilation. It is known as a run time of a program and denoted by t_P.

Example: Time Complexity of (priori analysis) of Max algorithm.

Pseudo-code	Cost	Times
Algorithm MAXIMUM (A, n)		
{		
// A is an array A[1..n] of n elements.		
MAX ← A[1];	C1	1
For i← 2 to n do	C2	n
If A[i] > MAX then	C3	n -1
MAX ← A[i];	C4	$\sum_{i=2}^{n} i$
Return MAX;		
}		

The Running time is represented by T (n),

$T(n) = C_1(1) + C_2(n) + C_3(n-1) + C_4\left\{\frac{n\,(n+1)}{2} - 1\right\}$

$= C_1 + C_2.n + C_3.n - C_3 + C_4\left\{\frac{n^2}{2}\right\} + C_4\left\{\frac{n}{2}\right\} - C_4$

$= \left\{\frac{C_4}{2}\right\} n^2 + (C_2 + C_3 + \left\{\frac{C_4}{2}\right\})\, n + (C_1 - C_3 - C_4)$

$= an^2 + bn + c.$

Hence the order of growth is Quadratic.

ASYMPTOTIC ANALYSIS

- Asymptotic analysis of an algorithm refers to defining the mathematical foundation or framing of its run-time performance.
- Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n^2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types

1. **Best Case** − Minimum time required for program execution.
2. **Average Case** − Average time required for program execution.
3. **Worst Case** − Maximum time required for program execution.

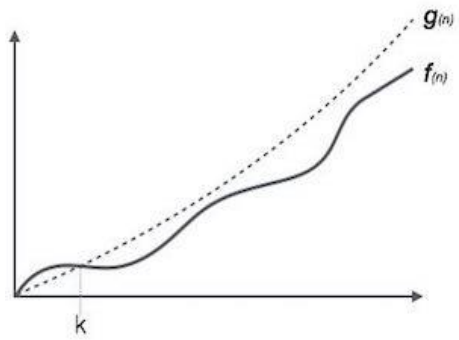
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

(i) Big Oh Notation (O – Notation)

The notation O (n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

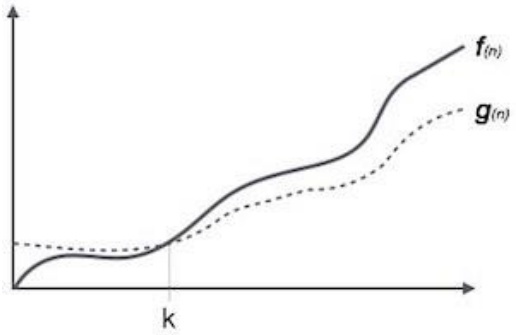


For example, for a function ***f* (n)**

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0. \}$

(ii) Omega Notation (Ω – Notation)

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

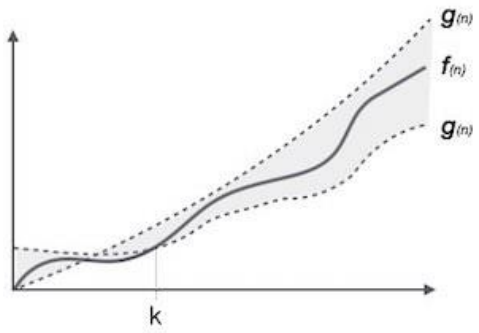


For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

(iii) Theta Notation (θ – Notation)

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows



$\Theta(f(n))$ = { $g(n)$ if and only if $g(n)$ = $O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0$. }

(iv) Little Oh Notation (o – Notation)

The Little –oh (0 – notation) is used to denote an upper bound that is not asymptotically tight.

$o(g(n)) = f(n)$: for any positive constant $c>0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$.

Little Oh and Big Oh Notation - Difference

The difference between O-notation and o-notation are that

in f(n) = O(g(n)), the bound 0 ≤ f (n) ≤ cg(n) holds for some constants c > 0, but in f(n) = o(g(n)), the bound 0 ≤ f (n) ≤ cg(n) holds for all constants c > 0.

The function f(n) becomes insignificant relative to g(n) as n approaches infinity, i.e.,

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

Example: Prove that 2n =o (n^2).

Solution: We Know that $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{2n}{n^2} = \frac{2}{n} = 0$. Thus 2n = o($n^2$).

(v) Little Omega Notation (ω – Notation)

The Little Omega or ω – Notation is used to denote a lower bound that is not asymptotically tight.

$\omega(g(n)) = f(n)$: for any positive constant $c>0$, If a constant $n_0 > 0$ such that $0 \le c\, g(n) < f(n)$ for all $n \ge n_0$.

Here ω-notation,

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

Example: Prove that $\frac{n^2}{2} = \omega(n)$.

Solution: We Know that $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{n^2}{2n} = \frac{n}{2} = \infty$. Thus $\frac{n^2}{2} = \omega(n)$.

Common Asymptotic Notations and Efficiency Classes

Even though the efficiency analysis framework puts together all the functions whose orders of growth differ by a constant multiple, there are still indefinitely many such classes as,

Name	Class	Explanation
Constant	O (1)	Short of best-case efficiencies. Algorithm's running time goes to infinitely when its input size grows infinitely large.
Logarithmic	O (log n)	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm. Logarithmic algorithms can not take into account all its input.
Linear	O (n)	Algorithm that scan a list of size n.
n log n	O (n log n)	Many divide-and-conquer algorithms in the average case fall into this category.
Quadratic	O (n^2)	Characterizes efficiency of algorithm with two embedded loops.
Cubic	O (n^3)	Characterizes efficiency of algorithm with three embedded loops.
Exponential	O (2^n)	Algorithm that generate all subsets of an n-element set. The term exponential is used in a broader sense to include this and larger orders of growth as well.
Factorial	O ($n!$)	Typical algorithms that generate all permutations of an n-element set.

ANALYZING THE TIME EFFICIENCY OF NON-RECURSIVE ALGORITHMS

General plan of analyzing Non – Recursive Algorithms:

1. Decide on a parameter (or parameters) indicating the size of the problem.
2. Identify the algorithm's basic operations
3. Check whether the number of times the basic operation is executed depends only on the size of the problem.
4. Set up a SUM expressing the number of times the algorithm's basic operation is executed.
5. Using standard formula and rules of sum manipulation find the solution and establish the order of growth.

Example #1: Write an algorithm to check whether all the elements in a array are distinct and analyze the time efficiency.

Description: Given an array A [1..n], check whether all the elements in A[1..n] are unique.

Goal: Solving Element uniqueness problem, Return TRUE if all the elements in A are distinct and FALSE otherwise.

Algorithm:

Algorithm Unique-Elements (A, n)
//Input: An array A [1..n] containing n elements.
//Output: Return TRUE, if all the element are distinct , FALSE otherwise
For i ← 1 to n-1 do
    For j ← i+1 to n do
        If A[i] == A[j] then return FALSE;
Return TRUE;

Analysis:

Parameter indicating i/p size: N – the number of elements in the array is the only parameter indicating the size of the given problem.

Algorithm's basic operation: Number of key comparison.

No of Key Comparisons made:

Let C (n) indicating the number of key comparisons made to determine whether the given 'n' elements are unique or not.

Instructions	Cost	Total Cost
For i ← 1 to n-1 do	C1	n
For j ← i+1 to n do	C2	$\sum_{j=i+1}^{n} i$
If A[i] == A[j] then return FALSE;	C3	$\sum_{j=i+1}^{n} 1$
Return TRUE;	C4	1

Exactly one comparison is made for each iteration of inner loop for each value of j between its limits i+1 and n: $\sum_{j=i+1}^{n} 1$

This is repeated for each value of i between its limit 1 to n-1, so C (n) = $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1$

C(n) = $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} [n - (i + 1) + 1]$

= $\sum_{i=1}^{n-1} [n - i]$

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

Hence, C (n) = $\frac{1}{2}$ n^2 - n, the order of growth is QUADRATIC. (i.e) C(n) ∈ O (N^2)

The algorithm needs to compare all $\frac{n(n-1)}{2}$ distinct pairs of its elements to declare the final result.

Example #2: Write an algorithm to sort the given elements into ascending order by using INSERTION SORT technique and analyze the time efficiency.

Description: Given an array A [1..n] of n unordered integers, choose A[i] and insert it into A[1..(i-1)] at proper position for i=2, 3, …, n.

Goal: Sort the elements in ascending order.

Algorithm:

```
Algorithm INSERTION-SORT (A, n)
//Input: An array A [1..n] of unordered or random elements.
//Output: The array A with the elements are ordered such that a1<a2<…<an.
for i ← 2 to n do
        t = a[i];
            for (j = i - 1; j >= 1 && t < a[j]; j--)
                a[j + 1] = a[j];
        a[j + 1] = t;
```

Analysis:

Parameter indicating i/p size: N – the number of elements in the array is the only parameter indicating the size of the given problem.

Algorithm's basic operation: Number of key comparison.

No of Key Comparisons made:

Let C (n) indicating the number of key comparisons made to determine whether the given 'n' elements are unique or not.

Instructions	Cost	Total Cost
for i ← 2 to n do	C1	n
t = a[i];	C2	n-1
for (j = i - 1; j >= 1&& t < a[j]; j--)	C3	$\sum_{j=i-1}^{1} i$
a[j + 1] ← a[j];	C4	$\sum_{j=i-1}^{1} 1$
a[j + 1] ← t;	C5	n-1

From the inner loop with t=0, the number of comparison is i, where i is limited from 2 to n.

So, C (n) = n-1 key comparisons.

C(n) $= \sum_{i=2}^{n} [i] = 2+3+..+n = \frac{n(n-1)}{2}$

Hence, C (n) = $\frac{1}{2}$ n^2 - n, the order of growth is QUADRATIC. (i.e) C (n) ∈ O (N^2)

Thus the worst case time-complexity of Insertion Sort algorithm is O (n^2).

Example #3: Write an algorithm to sort the given elements into ascending order by using SELECTION SORT technique and analyze the time efficiency.

Description: Given an array A [1..n] of n unordered integers, choose i^{th} smallest element and place it into A[i] the proper position, for i=1, 2, …, n.

Goal: Sort the elements in ascending order.

Algorithm:

Algorithm SELECTION SORT (A, n)
//Input: An array A [1..n] of unordered or random elements.
//Output: The array A with the elements are ordered such that $a_1<a_2<\ldots<a_n$.
for i ← 1 to n do
    j= i;
        for k← i + 1 to n do
            If A[k] < A[j] then j←k;
    t←A[i];A[i] ←A[j]; A [j] ←t;

Analysis:

Parameter indicating i/p size: N – the number of elements in the array is the only parameter indicating the size of the given problem.

Algorithm's basic operation: Number of key comparison.

No of Key Comparisons made:

Let C (n) indicating the number of key comparisons made to determine whether the given 'n' elements are unique or not.

Instructions	Cost	Total Cost
for i ← 1 to n do	C1	n+1
j= i;	C2	n
for k← i + 1 to n do	C3	$\sum_{k=i+1}^{n} k$
If A[k] < A[j] then j←k;	C4	$\sum_{k=i+1}^{n} 1$
t←A[i];A[i] ←A[j]; A [j] ←t;	C5	n

From the inner loop with t=0, the number of comparison is i, where i is limited from 2 to n.

So, C (n) = n-1 key comparisons.

C(n) $= \sum_{k=2}^{n} [1] = 2+3+..+n = \frac{n(n-1)}{2}$

Hence, C (n) = $\frac{1}{2}n^2$ - n, the order of growth is QUADRATIC. (i.e) C (n) ∈ O (N^2)

Thus the worst case time-complexity of Insertion Sort algorithm is O (n^2).

For more examples: Please refer your Class-work notebook.

ANALYZING THE ALGORITHM – USING STEP COUNT (TABULATION) METHOD

Program Step: Program Step is a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

- It's accounting for all the time spent in all parts of the program. The number of steps any program statement is assigned depends on the kind of statement.
- The Comments / beginning or end of block / function header counted as 0 steps
- An Assignment statement does not involve any procedure calls is counted as 1 step
- For Iterative statements (for, while, do-while) the step counts only for the control part of the statement. (i.e) a step count is equal to the number of step counts assignable to it.

General Plan of Step Count Method:

1. Define Steps/Execution for every statement / steps.
2. Determine the frequency of the steps based on N
3. Add-up the steps
4. Reduce the Sum.
5. Define the order of growth.

Example #1: Algorithm to find the sum of n elements in the array.

Program Statements / steps	s/e	Frequency	Total steps
Algorithm Sum(A[1..n], n)			
sum = 0;	1	1	1
for i← 1 to n do	1	N+1	N+1
sum ← sum + A[i];	1	N	N
return sum;	1	1	1
Total			**2N + 3**

T(n) = 1 + (N+1) + N + 1 = 2N + 3. (ie) **T(n) ε O(N), Order of growth is Linear.**

Example #2: Consider a segment of a program,

for i = 1 to n	1	n+1
j = n * 2	1	n
while j >= 1	1	n(2n+1)
j = j – 1	1	n(2n)

So Total step count T(n) = n+1 + n + $2n^2$ + n + $2n^2$ = **$4n^2$ + 3n + 1**

T(n) ε O(N^2), Order of growth is Quadratic

Example #3: Sum of n integers – segment of code.

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tsum += list[i];	1	n	n
return tsum;	1	1	1
}	0	0	0
Total			**2n+3**

T(n) ε O(N), Order of growth is LINEAR

Example #4: Recursive Sum of n Integers

Statement	s/e	Frequency	Total steps
float Rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n>0)	1	n+1	n+1
return Rsum(list, n-1)+list[n-1];	1	n	n
else return list[0];	1	1	1
}	0	0	0
Total			**2n+2**

T(n) = 2n + 2, T(n) ε O(N), Order of growth is LINEAR

Example #5: Sum of two Matrices.

Statement	s/e	Frequency	Total steps
Void Matrix-Add (a[m][n], b[m][n])	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < m; i++)	1	m+1	m+1
for (j=0; j< n; j++)	1	m(n+1)	m(n+1)
c[i][j] = a[i][j] + b[i][j];	1	mn	mn
}	0	0	0
Total			**2mn+2m+1**

T(n) = 2n + 2, T(n) ε O(N), Order of growth is LINEAR [Quadratic for N x N Square matrix)

ANALYZING THE TIME EFFICIENCY OF RECURSIVE ALGORITHMS

RECURSIVE ALGORITHMS

- A recursive function is a function that is defined in terms of itself. Similarly an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An Algorithm that calls itself is ***direct recursive***. Algorithm A is said to be ***indirect recursive*** if it calls another algorithm which in turn calls A.

General plan of analyzing Recursive Algorithms

1. Decide on a parameter (or parameters) indicating the size of the problem.
2. Identify the algorithm's basic operations.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size.
4. Set up a RECURRENCE relation with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

Example #1: Write a recursive algorithm to compute the factorial of n and do the analysis.

Description: Given a positive integer say n, to find the factorial of n.
Goal: Recursive computation of n*!* = n. (n-1)*!* with the initial condition that 0*!* = 1.
Constraint: The input integer should be a greater than 0. (Positive integer)
Algorithm:

Algorithm Fact (n)
//Input: A Positive integer n.
//Output: The factorial value of n.
If n=0 then return 1;
else return Fact(n-1) * n;

Analysis:

Parameter indicating i/p size: N – the positive integer for which the factorial is to be computed.

Algorithm's basic operation: Multiplication during recursive call. The factorial is computed by using the formula,

Fact (n) = Fact (n-1) * n, for n>0.

No of Multiplication with respect to the problem instance

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

M (n) = M (n-1) + 1, for n>0.

Where M (n-1) is to compute Fact(n-1) and 1 to multiply Fact (n-1) by n.

Set up the Recurrence relation appropriate with an Initial condition

The Recurrence relation and the initial condition for the algorithm's number of multiplications M(n):

M (n) = M (n-1) + 1 for n>0.
M (0) = 0

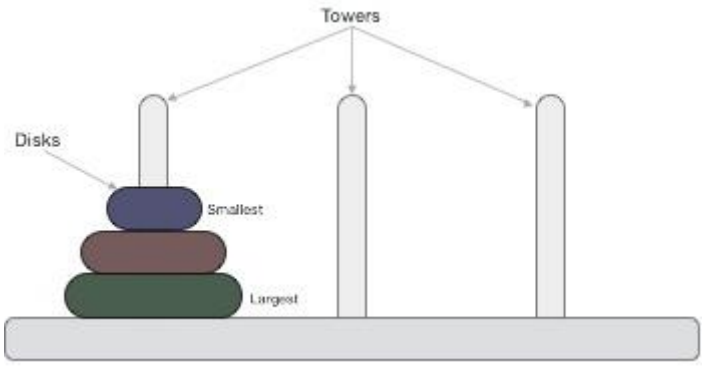
Solve the Recurrence

Solve by the method of backward substitution method,

$M(n) = M(n-1) + 1$
$= [M(n-2) + 1] + 1 = M(n-2) + 2$ substituted $M(n-2)$ for $M(n-1)$
$= [M(n-3) + 1] + 2 = M(n-3) + 3$ substituted $M(n-3)$ for $M(n-2)$
...
$= [M(n-i) +1] + (i-1) = M(n-i) + i$, substituted i^{th} term for $(i-1)^{th}$ term
…
$= M(0) + n$, substituted n^{th} term
$= 0 + n$
$= n$

Therefore $M(n) \in \Theta(n)$, Linear.

Example #2: Write a recursive algorithm to solve the Tower of Hanoi puzzle.



Description: There are n disks of different sizes and three pegs. Initially all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
Goal: The goal is to move all the disks to the third peg, using the second peg as an auxiliary peg, if necessary.
Constraint:

1. Move only one disk at a time.
2. Placing larger disk on top of the smaller disk is forbidden.

Algorithm:

```
Algorithm ToHanoi  (N, Source, Destination, Aux)
If N==1 then
        Move a Disk from Source to Destination
else
        ToHanoi(n-1, Source, Aux, Destination)
        Move a Disk from Source to Destination
        ToHanoi(n-1, Aux, Destination, Source)
```

Analysis:
Problem size is *n*, the number of disks.
The **basic operation** is moving a disc from one peg to another peg.
There is no worst or best case
No of **Moves:** To move n>1 disks from peg 1 to peg-3,
(i) Move recursively (n-1) disks from peg-1 to peg-2(with peg-3 as auxiliary)
(ii) Move the largest disk from peg-1 to peg-3.
(iii) Move recursively remaining (n-1) disks from peg-2 to peg-3 (with peg-1 as auxiliary)

Recursive relation for moving n discs. The number of moves depends on n only.

$M(n) = M(n\text{-}1) + 1 + M(n\text{-}1) = 2M(n\text{-}1) + 1$

Initial Condition: $M(1) = 1$

Solve using backward substitution

$M(n) = 2M(n\text{-}1) + 1$

$= 2[2M(n\text{-}2) + 1] + 1 = 2^2M(n\text{-}2) + 2 + 1$

$= 2^2[2M(n\text{-}3) + 1] + 2 + 1 = 2^3M(n\text{-}3) + 2^2 + 2 + 1$

...

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

...

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

Therefore, $\boldsymbol{M(n) \in \Theta(2^n)}$**, Exponential.**

Example #3: Write a recursive algorithm to find the number of binary digits in the binary representation of a positive decimal integer.

Description & Goal: to find the number of binary digits in the binary representation of a positive decimal integer.

Constraint:

1. N is a positive number.

Algorithm:

Algorithm BinaryRec(N)
Input: A positive decimal integer.
Output: The number of binary digits in n's binary representation
If N==1 then
    Return 1;
else return BinaryRec($\left\lfloor \frac{N}{2} \right\rfloor$) + 1;

Analysis:

1. Problem size is n
2. Basic operation is the addition in the recursive call
3. There is no difference between worst and best case
4. Recursive relation including initial conditions

   $A(n) = A(\text{floor}(n/2)) + 1$ with $A(1) = 0$
5. Solve recursive relation

   The division and floor function in the argument of the recursive call makes the analysis difficult. We could make the variable substitution, $n = 2^k$, could get rid of the definition, but the substitution skips a lot of values for n.

Let substitute $n = 2^k$

$A(2^k) = A(2^{k-1}) + 1$ and $A(2^0) = 0$

$A(2^k) = [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$

...

$= A(2^{k-i}) + i$

...

$= A(2^{k-k}) + k$

$A(2^k) = k$

Substitute back $k = \log(n)$

$\boldsymbol{A(n) = \log(n) \in \Theta(\log n)}$**, Logarithmic**.

Example #4: Write a recursive algorithm to find the n^{th} Fibonacci number.

Description: Given the two basic elements 0 and 1, the successive number is the sum of previous two numbers in the series.
Goal: To find the nth Fibonacci number.
Algorithm:

Algorithm Fib(N)
Input: A positive integer N.
Output: The value of N^{th} Fibonacci number in the series.
if $n \leq 1$ **then return** n
else return $F(n\text{-}1) + F(n\text{-}2)$

Analysis:

1. Problem size is *n*, the sequence number for the Fibonacci number
2. Basic operation is the sum in recursive call
3. No difference between worst and best case
4. Recurrence relation

$A(n) = A(n\text{-}1) + A(n\text{-}2) + 1$ with $A(0) = A(1) = 0$

or

$A(n) - A(n\text{-}1) - A(n\text{-}2) = 1$, **Inhomogeneous recurrences** because of the 1

In general solution to the inhomogeneous problem is equal to the sum of solution to homogenous problem plus solution only to the inhomogeneous part. The undetermined coefficients of the solution for the homogenous problem are used to satisfy the initial condition.

In this case $A(n) = B(n) + I(n)$ where

$A(n)$ is solution to complete inhomogeneous problem

$B(n)$ is solution to homogeneous problem

$I(n)$ solution to only the inhomogeneous part of the problem

We guess at $I(n)$ and then determine the new initial condition for the homogenous problem for $B(n)$. For this problem the correct guess is $I(n) = 1$

substitute $A(n) = B(n) - 1$ into the recursion and get

$B(n) - B(n\text{-}1) - B(n\text{-}2) = 0$ with IC $B(0) = B(1) = 1$

The same as the relation for $F(n)$ with different IC

We do not really need the exact solution; We can conclude

$\boldsymbol{A(n) = B(n) - 1 = F(n+1) - 1 \in \Theta(\varphi^n)}$**, Exponential.**

RANDOMIZED ALGORITHMS

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, random number is used to pick the next pivot.

Classification of Randomized Algorithms

Randomized algorithms are classified in two categories.

1. **Las Vegas:** These algorithms always produce correct or optimum result. Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value
2. **Monte Carlo:** Produce correct or optimum result with some probability. These algorithms have deterministic running time and it is generally easier to find out worst case time complexity.

Example to Understand Classification:

Consider a binary array where exactly half elements are 0 and half are 1.
The task is to find index of any 1.

A Las Vegas algorithm for this task is to keep picking a random element until find a 1. A Monte Carlo algorithm for the same is to keep picking a random element until either find 1 or have tried maximum allowed times say k. The Las Vegas algorithm always finds an index of 1, but time complexity is determined as expect value.

Analyzing Randomized Algorithms

- Time complexity of randomized algorithms (other than Las Vegas) is dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms. These algorithms are typically analyzed for expected worst case.
- To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated.
- Average of all evaluated times is the expected worst case time complexity.

Example #1: Identifying the repeated element from the array.
Description: Given an array A [1..n] of n elements, to identify the repeated element. It assumes that the array has n/2 distinct elements and n/2 copies of another element.
Goal: To identify the repeated element from the array.

```
Algorithm Repeated-Element (A, n)
Input: An Array A [1..n] of n elements.
Output: Return the index of the repeated element from the array, if any.
while TRUE do
        i ← Random() mod n+1;
        j ← Random() mod n+1;
        If (i≠j) and (A[i] = A[j]) then return i;
```

Analysis: This algorithm randomly selects any two elements from out of n elements and the two elements are the same, then it returns the array index of one of the copies of the repeated element. The run time of the above algorithm is O(log n).

Example #2: Search an element x from the array.

Description: Given an array A [1..n] of n elements, to search for an element x in an array.
Goal: to search for an element x in an array

```
Algorithm Random-Search (A, x)
Input: An Array A [1..n] of n elements.
Output: Return the index of the element x, if found in the array, otherwise -1.
while TRUE do
        i ← Random() mod n+1;
                If (A[i] = x ) then return i;
Return -1;
```

Applications and Scope:

1. Randomized algorithms have huge applications in Cryptography.
2. Number-Theoretic Applications: Primality Testing
3. Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
4. Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
5. Mathematical programming: Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
6. Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
7. Counting and enumeration: Matrix permanent Counting combinatorial structures.
8. Parallel and distributed computing: Deadlock avoidance distributed consensus.
9. Probabilistic existence proofs: Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.
10. De-randomization: First devise a randomized algorithm then argue that it can be de-randomized to yield a deterministic algorithm.

PART – II: DIVIDE AND CONQUER TECHNIQUE

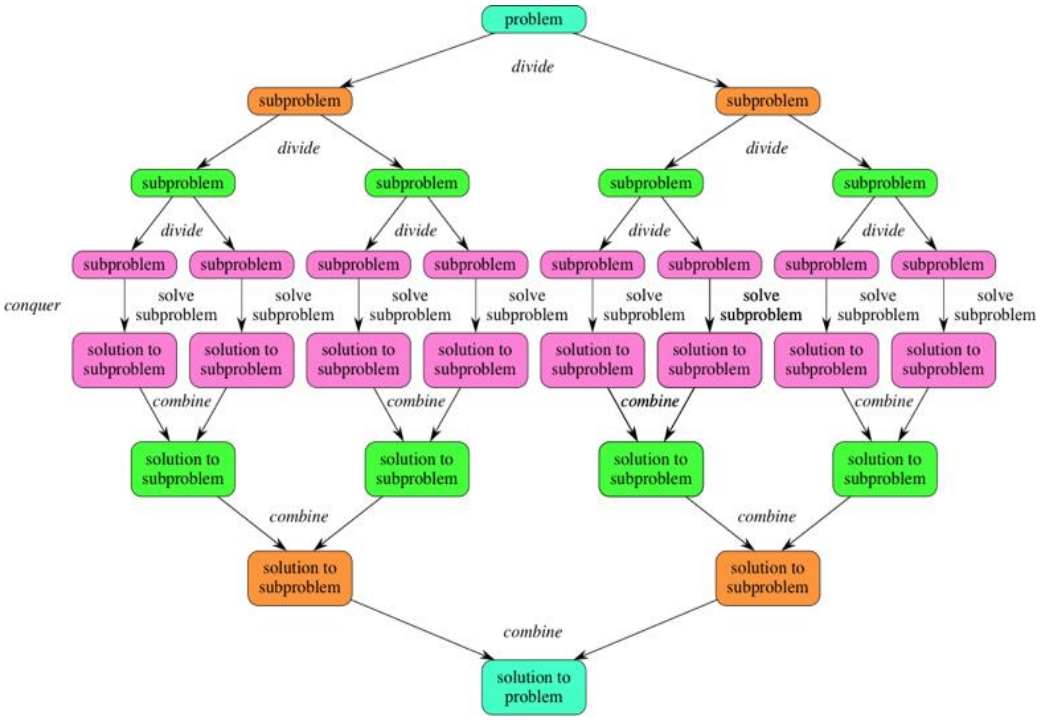
DIVIDE AND CONQUER

- Divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion.
- A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem.
- This divide and conquer technique is the basis of efficient algorithms for all kinds of problems like,
    1. Sorting (e.g., quicksort, merge sort),
    2. Multiplying large numbers (e.g. The karatsuba algorithm),
    3. Finding the closest pair of points,
    4. Syntactic analysis (e.g., top-down parsers), and
    5. Computing the discrete Fourier transform (FFTs).

General Method of Divide and Conquer Technique

Divide and Conquer algorithms work according to the following general plan:

- **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer** the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine** the solutions to the sub-problems into the solution for the original problem.



ANALYZING DIVIDE-AND-CONQUER ALGORITHMS

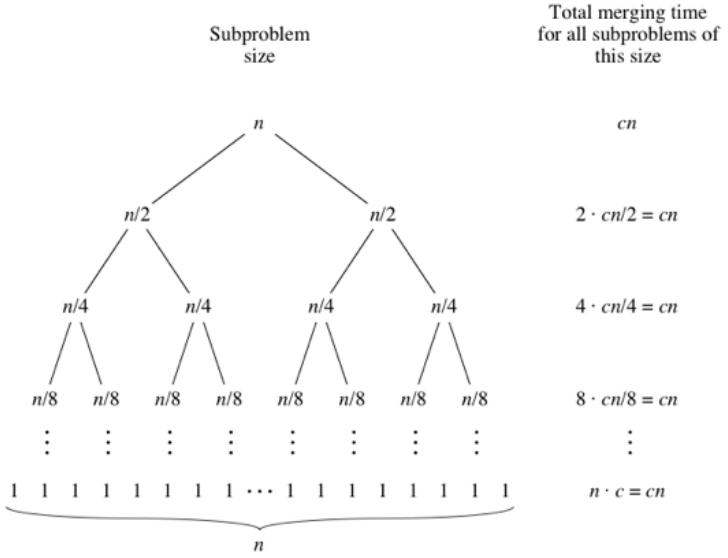
(i) Recurrence for Divide-and-Conquer Algorithms

Typically in case of Divide-and-Conquer, a problem instance of size n is divided into two instances of size $\frac{n}{2}$. More generally, an instance of size n, can be divided into 'b' instances of size $\frac{n}{b}$ with 'a' of them needing to be solved. It leads to the following recurrence for the running time T (n),

$$\mathbf{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. This recurrence is called the ***general divide-and-conquer recurrence.***

The order of growth of divide-and-conquer solution T (n) depends on the value of the constants a, b and the order of growth of a function f(n).



(ii) Solving Divide-and-Conquer Recurrences

The efficiency analysis of the divide-and-conquer algorithm is greatly simplified by the master's theorem as follows,

Theorem: If f(n) = $O(n^d)$ with d ≥ 0 in the recurrence equation, then

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

1. MERGE SORT ALGORITHM

- Merge sort is a classical recursive divide-and-conquer algorithm for sorting an array.
- The algorithm splits the array in half, recursively sorts the two halves, and then merges the two sorted sub-arrays into the final sorted array.

ALGORITHM

Description:

- This algorithm sorts a given array A[1..n] by dividing it into two halves A[1..n/2] and A[(n/2)+1..n].
- Then, Sorting each of them recursively and then merging the two smaller sorted arrays into a single sorted one.

Merging:

- Two pointers are initialized to point the first elements of the array being merged.
- The elements pointed to are compared and then smaller of them is added to a resultant array being constructed. Index of smaller element is incremented to point its successor.

Goal: To order the elements in increasing order by merging two parts of an array.

Algorithm:

```
Algorithm MergeSort (A [1..n])
Input: An array A [1..n] of orderable elements
Output: Array A [1..n] of Sorted elements.
if (n > 1)
        m ← Floor(n/2)
        MergeSort (A [1 .. m])
        MergeSort (A [m + 1 .. n])
        Merge (A [1 .. n], m)
```

```
Algorithm Merge (A [1..n], m)
i ← 1; j ← m + 1
for k ← 1 to n
 if j > n
        B[k] ← A[i]; i ← i + 1
else if i > m
        B[k] ← A[ j]; j ← j + 1
else if A[i] < A[ j]
        B[k] ← A[i]; i ← i + 1
else
        B[k] ← A[ j]; j ← j + 1
for k ← 1 to n
        A[k] ← B[k]
```

ANALYSIS OF MERGE SORT ALGORITHM

Parameter indicating the size of the Problem: The number of elements in the array represented as n, is determining the size of the problem.

Identify the basic operations: The basic operation for any sorting algorithm is the key Comparison.

No of Comparisons made: The recurrence for the number of key comparisons C(n) is

$$\mathbf{C(n) = 2\,C\left(\frac{n}{2}\right) + C_{merge}(n)} \qquad \text{for } n>1 \text{ and } C(1) = 0.$$

Setting-up the Recurrence:

Let T(n) denote the worst-case running time of Merge Sort when the input array has size n. The Merge procedure clearly runs in Θ(n) time, so the function T(n) satisfies the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \text{ [No of comparison} \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$$

Solving the Recurrence:

For analysis taking the value of the input parameter n is a power of 2, allows us to take the floors and ceilings out of the recurrence.

The simplified recurrence now looks like this:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2\,T\left(\frac{n}{2}\right) + n & \text{otherwise.} \end{cases}$$

To solve a solution, unrolling the recurrence. [by substitution method]

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

$= 2.[2T(n/4) + n/2] + n = 4T(n/4) + 2n$

$= 8\,T(n/8) + 3n = \cdots$

So, It looks like T(n) satisfies the recurrence

$T(n) = 2^k T(n/2^k) + k.n$ for any positive integer k.

Let's verify this by induction.

$T(n) = 2T(n/2) + n = 2^1\,T(n/2^1) + 1 \cdot n$ [k = 1, given recurrence]

Taking $(k-1)^{th}$ term,

$T(n) = 2^{k-1}\,T(n/2^{k-1}) + (k-1)n$ [inductive hypothesis]

$= 2^{k-1}\,[2T(n/2^k) + n/2^{k-1}] + (k-1)n$ [substitution]

$= 2^{k-1}.2\,T(n/2^k) + [(n/2^{k-1}).\,2^{k-1}] + kn - n$

$= 2^k\,T(n/2^k) + n + kn - n = 2^k\,T(n/2^k) + kn$ [algebra]

The recurrence becomes trivial when $n/2^k = 1$, or equivalently, when $k = \log_2 n$

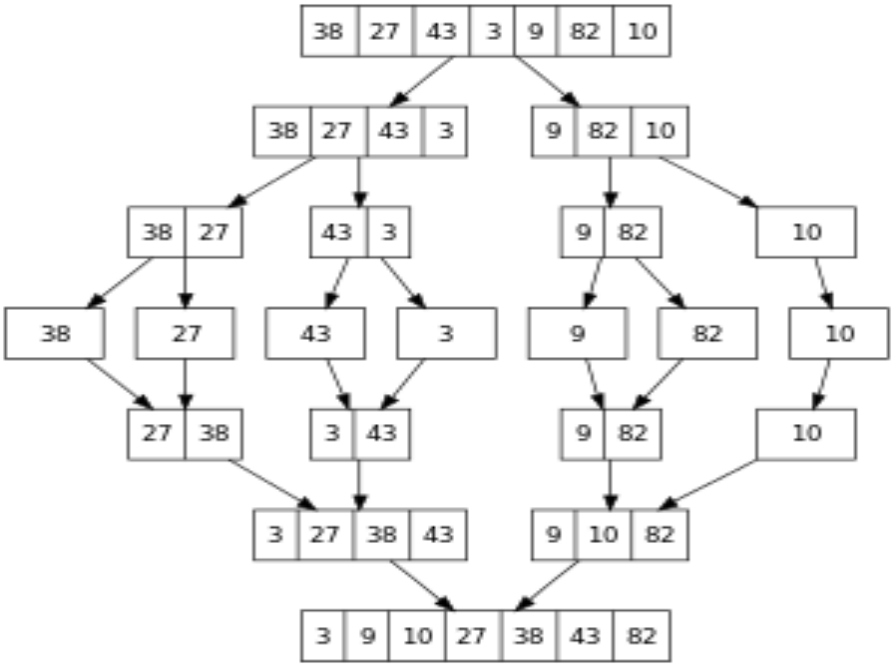
$T(n) = n\,T(1) + n \log_2 n$

$= n \log_2 n + n.$

Finally, the solution is $T(n) = \Theta(n \log n)$.

Hence the worst case performance of the Merge sort algorithm is O(n log n).

Example for Merge Sort and Merge Algorithms (tree of calls)



2. BINARY SEARCH ALGORITHM

- Binary search is a fast search algorithm with run-time complexity of O (log n).
- This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.
- **Divide & Conquer**: Search either lower or upper half
  - ✓ **Divide:** Select lower or upper half
  - ✓ **Conquer:** Search selected half
  - ✓ **Combine:** None.

Procedure:

1. Binary search looks for a particular item by comparing the middle most item of the collection.
2. If a match occurs, then the index of item is returned.
3. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item.
4. This process continues on the sub-array as well until the size of the sub-array reduces to zero.

Algorithm

```
Algorithm Binary-Search (A[0..N-1], value, low, high)
Input: An Array A [1..n] has elements are ordered & the element value.
Output: Return the position of the element value, if found in the array.
    if high < low then
        mid = low +(high – low) / 2)
        if A[mid] =value then return mid;
        else if A[mid] > value then
            return Binary-Search (A, value, low, mid-1)
        else
            return Binary-Search(A, value, mid+1, high)
    else return -1;
```

Analysis:

1. Problem size is *n*, the number of elements in the given array A.
2. Basic operation is the key comparison.
3. The number of key comparisons is C (n),

Worst case: The worst case inputs include all array that do not contain a given key , since after one comparison the algorithm half the size of an array, the recurrence for C_{worst} (n) is,

$$\mathbf{C_{worst}(n) = C_{worst}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1, \text{ for } n>1} \text{ with Initial condition } C_{worst}(1) = 1.$$

Solving the Recurrence: The standard way of solving such a recurrences is to assume $n = 2^k$ and solve the resulting recurrence by backward substitution,

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

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

$= [C_{worst}(2^{k-3}) + 1] + 2 = C_{worst}(2^{k-3}) + 3$

....

$= [C_{worst}(2^{k-i}) + 1] + (i-1) = C_{worst}(2^{k-i}) + i$

....

$= [C_{worst}(2^{k-k}) + 1] + (K-1) = C_{worst}(2^{0}) + k$

$= K+1$

Since $n=2^k$,

$C_{worst}(n) = (\log_2 n) + 1 = \log_2 (n+1)$. , Therefore $C_{worst}(n) \varepsilon O(\log_2 (n+1))$, Logarithmic.

Average case: The average number of comparison made by binary search algorithm is only slightly smaller than that in the worst case so,

$C_{average}(n) = \log_2 n$, Logarithmic.

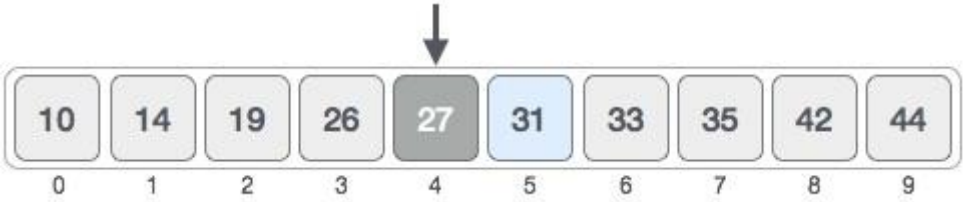
Example: Binary Search

For a binary search to work, it is mandatory for the target array to be sorted. The following is sorted array and let assume that the element need to be searched is 31,

First, determine the mid of the array by using this formula,

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now compare the value stored at location 4, with the value being searched, i.e. 31. The value at location 4 is 27, which is not a match. As the value is greater than 27 and the array is sorted, so the target value must be in the upper portion of the array.

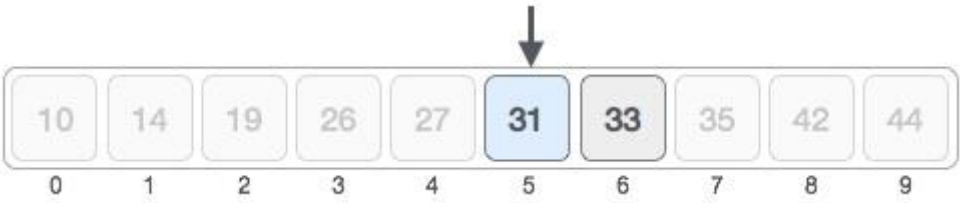
Now update low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

The new mid is 7. Now compare the value stored at location 7 with the target value 31.

The value stored at location 7 is not a match, rather it is less than 31. So, the value must be in the lower part from this location.

Hence, calculate the mid again. This time it is 5.



Now compare the value stored at location 5 with the target value. It is a match.

The algorithm concludes that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

3. MAXIMUM & MINIMUM ALGORITHM

- The problems of finding the maximum and minimum element in a set of n elements can also be solved by divide and conquer technique.
- The straight MAXMIN algorithm requires 2(n-1) element comparisons in the best, average and worst cases.

Divide and conquer approach

A divide and conquer algorithm for this problem would proceed as follows,

Let P=(n, a[i],…,a[j]) denote an arbitrary instance of the problem where n is the number of elements in the list a[i]…a[j] and to find the maximum and minimum,

Case – 1: If n=1, then no comparison is needed as maximum = minimum = a[1].

Case – 2: If n=2, then the problem is solved by making only one comparison.

Case – 3: If P has more than two elements, P has to be divided into smaller instances. P is divided into P1(n/2, a[1],…a[n/2]) and P2(n-n/2, a[(n/2)+1…a[n]). After having divided P into two smaller sub-problems, solve them by recursively invoking the same algorithm.

Algorithm

Algorithm MAXMIN (low, high, MAX, MIN)

Input: A global Array A [1..n]. Parameters low and high are integers $1 \le low \le high \le n$

Output: To set MAX and MIN to the largest and smallest value in A[low...high] resp.

```
if low = high then MAX ← MIN ← A[low]                    // one element
else if low = (high-1) then                              // two elements
        if A[low] < A[high] then MIN ← A[low], MAX ← A[high]
        else MAX ← A[low], MIN ← A[high]
```

```
else
        mid ← ⌊((low+high)/2)⌋                          // n>2
        MAXMIN (low, mid, MAX, MIN)
        MAXMIN (mid+1, high, MAX, MIN)
// Combine the solutions
If MAX1 > MAX then MAX ← MAX1
If MIN1 < MIN then MIN ← MIN1
```

Analysis:

1. Problem size is *n*, the number of elements in the global array A.
2. Basic operation is the key comparison.
3. The number of key comparisons needed for MAXMIN algorithm is denoted by C (n), and the resulting recurrence relation is ,

$$\text{C (n)} = \begin{cases} C\left(\frac{n}{2}\right) + C\left(\frac{n}{2}\right) + 2, & for\ n > 2 \\ 1 & for\ n = 2 \\ 0 & for\ n = 1 \end{cases}$$

4. Solving the Recurrence,

Let n= 2^k for some positive integer k, and by backward substitution method,

C(n) = $2\,C\left(\frac{n}{2}\right) + 2$

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

…

= $2^{k-1}\,C(2) + 2^{k-1} + \ldots + 2^0$

= $2^{k-1} + 2^k - 2$ = $3\frac{n}{2} - 2$

So, $\frac{3n}{2} - 2$ is the best, worst and average number of comparisons when n=2^k. For the random value of n, **C (n) = ($\log_2$ n) + 1. Therefore C (n) εO ($\log_2$ (n+1)), Logarithmic.**

Note: When compared with 2n-2 comparisons for the straight MINMAX algorithm, this divide and conquer algorithm is saving 25% in comparison.

Also the comparisons among the elements of A[1..n] are much more costly than comparisons of integer variables (low, high, mid), hence the divide and conquer algorithm has yielded a more efficient algorithm than a straight algorithm.

4. QUICK SORT ALGORITHM

- The divide-and-conquer approach can be used to arrive at an efficient sorting method than merge sort.
- The Quick-Sort algorithm is obtained by exploring the possibility of dividing the elements such that there is no need of merging, that is Partition *a* [1…*n*] into sub-arrays A′ = A [1..q] and A′′ = A[q + 1…n] such that all elements in A′′ are larger than all elements in A′. Recursively sort A′ and A′′.

Algorithm:

Algorithm QUICKSORT (A, low, high)	***Algorithm PARTITION (A, p, r)***
If low < high THEN	**pivot ← A [p]**
q = PARTITION (A, low, high)	**Repeat**
QUICKSORT (A, low, q – 1)	**Repeat i←i+1 Until (A[i]≥pivot);**
QUICKSORT (A, q + 1, high)	**Repeat j←j-1 Until (A[j]≤pivot);**
	A[i]↔A [j];
	Until (i ≥ j);
	A[j]↔A [pivot];
	Return j

- The algorithm PARTITION is called by QUICKSORT, which always divides *A* in two halves. After a partition has been achieved, A[q] will be in its final position in the sorted array.
- Then, sort the two sub-arrays of the elements preceding and following A[q] independently by recursively calling the same algorithm.
- The partition algorithm first select an element (pivot) with respect to whose value it divide the sub-array. The simplest strategy of selecting the sub-array's first element Pivot=A[low] is used.

Example:

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**i**	**j**	**pivot**
12	18i→	17	10	13	15	6	←14j	2	8	12
12	18i	17	10	13	15	6j	14	2	7	
12	6	17i	10	13	15j	18	14	3	6	
12	6	17i	10j	13	15	18	14	3	4	
12	6	10j	17i	13	15	18	14	4	3	q=3
10	6	12	17	13	15	18	14	-	-	
10	6ij	12	17	13i	15	18	14j	2, 5	2, 8	10,17
6	10	12	17	13	15	18i	14j	7	8	
6	10	12	17	13	15	14j	18i	8	7	q=7
6	10	12	14	13	15	17	18	-	-	
6	10	12	14	13i	15j	17	18	5	6	14
6	10	12	14	13j	15i	17	18	6	5	q=5
6	10	12	13	14	15	17	18	-	-	14
6	**10**	**12**	**13**	**14**	**15**	**17**	**18**	**-**	**-**	**-**

- There are two scan of sub-arrays: one from left-to-right and the other from right-to-left, each comparing the sub-array's elements with pivot.
- The left-to-right scan denoted by index 'i', starts with second element and skips over elements that are smaller than pivot. It stops on encountering the first element greater than or equal to pivot.

- Similarly the right-to-left scan denoted by index 'j' and skips over the elements that are greater than pivot. It stops on encountering the first element smaller than or equal to pivot.
- If scanning of indices 'i' and 'j' have not crossed, then simply exchange A[i] and A[j] and resume scan by incrementing 'i' and decrementing 'j'. If scanning of indices have crossed over then exchange the pivot with A [j].

Analysis of Quick Sort

1. Parameter indicating is n, the number of elements in the given array.
2. The basic operation is key comparison.
3. The number of key comparison C (n) is computed based on the cases,

Worst case analysis:

- A possible input on which Quick Sort displays worst case behaviour is one in which the elements are already in sorted order. Then the partition will be such that there will be only one element in one part and the rest of the elements will fall in the other part. (i.e) having 0 elements in one sub-array and $n - 1$ elements in the other sub-array.
- It leads to the recurrence,

$$C_{worst}(n) = C(n-1) + C(0) + O(n)$$
$$= C(n-1) + O(n)$$
$$= O(n^2)$$

Therefore, $C_{worst}(n) = n^2$, $C_{worst}(n) \in O(n^2)$, quadratic.

Best case analysis:

- It occurs when the sub-arrays are completely balanced every time.
- Each sub-array has $\leq n/2$ elements.
- Get the recurrence,

$$C_{best}(n) = 2C(n/2) + \Omega(n)$$
$$= \Omega(n \log n).$$

Therefore, $C_{best}(n) = n \log n$, $C_{best}(n) \in \Omega(n \log n)$. Logarithmic (n log n)

Average case analysis:

The average value $C_A(n)$ of C(n) is computed under the assumption that the partitioning element pivot has an equal probability of being the i^{th} smallest element, $1 \leq i \leq r-p$ in A[p..r]. hence the two sub-arrays to be sorted are A[p..q] and A[q+1..r] with the probability that 1/(p-r). This leads to the recurrence,

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} [C_A(k-1)) + C_A(n-k)]$$

Multiplying both sides by n,

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)]$$

Replacing n by (n-1),

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

Subtracting the above equation from the previous,

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

Solving this by substitution method,

$$\begin{aligned}\frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= \frac{C_A(1)}{2} + 2\sum_{3\le k\le n+1}\frac{1}{k} \\ &= 2\sum_{3\le k\le n+1}\frac{1}{k}\end{aligned}$$

$$\sum_{3\le k\le n+1}\frac{1}{k} \le \int_2^{n+1}\frac{1}{x}\,dx = \log_e(n+1) - \log_e 2$$

Thus the average case performance is,

$$C_A(n) \le 2(n+1)[\log_e(n+2) - \log_e 2] = O(n\log n)$$

5. RANDOMIZED QUICK SORT ALGORITHM

- When the input is already in sorted order, then the partition will be such that there will be 0 element in one sub-array and *n* − 1 elements in the other sub-array. This leads to the worst case performance of the quicksort is O (n^2).
- The performance of the quicksort will be improved if the resultant sub-arrays (after placing the pivot element) are as evenly sized as possible, so that it performs well on every input.
- It can be achieved by using a randomizer (i.e.,) while sorting the array A [p...r], instead of picking A[p], pick a random element from A[p]…A[r] as partitioning element (pivot).

Idea behind *Randomized-QUICK-SORT algorithm*

1. Choose an element x ∈ A, uniformly at random. Take x as the pivot.
2. Partition A around x.
3. Sort the elements less than x recursively.
4. Sort the elements greater than x recursively.

Algorithm

Algorithm Randomized-QUICKSORT (A, p, r)	***Algorithm PARTITION (A, p, r)***
If p< r THEN	**pivot ← A [p]**
A[Random()%(r-p)+p] ⟷A[p]	**Repeat**
q = PARTITION (A, p, r)	*Repeat i ←i+1 Until (A[i]≥pivot);*
QUICKSORT (A, p, q – 1)	*Repeat j ←j-1 Until (A[j]≤pivot);*
QUICKSORT (A, q + 1, r)	*A[i] ⟷A [j];*
	Until (i ≥ j);
	A[j]⟷A [pivot];
	Return j

Analysis of Randomized Quick Sort Algorithm

- Although the Quicksort algorithm is very efficient in practice, its worst-case running time is rather slow. When sorting n elements, the number of comparisons may be O (n^2).
- The worst case happens if the sizes of the sub-problems are not balanced. The selection of the pivot element determines the sizes of the sub-problems. It is thus crucial to select a 'good' pivot.

- If the pivot is selected uniformly at random, the expected number of comparisons of this randomized version of Quicksort is bounded by O (n log n).

Let a = (a1, a2, . . . , ai , . . . , aj , . . . , an) denote the list of elements to be sorted, in sorted order. For all $1 \le i < j \le n$, let $X_{i,j}$ denote the random variable indicating whether Quicksort compared a_i and a_j.

$$\mathbf{X}_{i,j} = \begin{cases} 1, & \text{if quicksort compares } a_i \text{ and } a_j \\ 0, & \text{Otherwise} \end{cases}$$

Any two elements get compared at most once. The expected number of comparisons is thus,

$$E(X) = E\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} E\left[X_{ij}\right]$$

Quicksort compares a_i and a_j if and only if either a_i or a_j is selected as pivot before any of the elements between a_i and a_j.

Selecting an element between ai and aj as pivot will partition the list into two parts such that ai and aj lie in different parts and do not get compared.

The number of elements between a_i and a_j is $j - i - 1$. Since in each step the pivot element is selected uniformly at random, the probability that either a_i or a_j is selected as pivot before any of the $j - i - 1$ elements between them is $2/(j - i + 1)$. If this happens, a_i and a_j are compared and $X_{i,j}$ is one. Thus,

$$E[X_{i,j}] = \frac{2}{j - i + 1}.$$

$$\begin{aligned} E\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{i,j}\right] &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} E[X_{i,j}] \\ &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\ &\leqslant 2\sum_{i=1}^{n}\sum_{k=1}^{n} \frac{1}{k} \\ &= 2n\sum_{k=1}^{n} \frac{1}{k} \end{aligned}$$

= O (n log n).

Therefore, E[X] = O (n log n), E[X] ε O (n log n), **Logarithmic.**

6. PARTITIONED SELECTION (RANDOMIZED SELECTION) ALGORITHM

- Finding the k^{th} smallest element in an array doesn't require sorting but it needs to find out where the k^{th} smallest element will fit in.
- The right way to find the k^{th} minimum element is randomized selection algorithm.
- The randomized partition which is used for Randomized Quick Sort can also be used in randomized selection.

Concept: Do randomized partition. Get the location of the pivot which is also random. After you get the location, just subtract the location from the start of the array. This is nothing but the k^{th} position in the sorted array.

Algorithm Random_Partition (A, low, high)	***Algorithm Random_Selection (A, Low, high, k)***

p = low + RAND() % (high-low+1);
pivot = A[p];
swap(A[p], A[high]);
p=high;
i = low -1;
for j←low to high-1
if(A[j] <= pivot)
i = i+1;
swap(A[i], A[j]);
swap(A[i+1], A[p]);
return i+1;
	if(low == high)
	return A[low];
	if(k ==0) return -1;
	if(low < high)
	mid = Random_Partition (A, low, high);
	i = mid - low + 1;
	if(i == k)
	return A[mid];
	else if(k < i)
	return Random_Selection (A, low, mid-1, k);
	else
	return Random_Selection (A, mid+1, high, k-i);

7. STRASSEN's MATRIX MULTITPLICATION ALGORITHM

The naive Matrix Multiplication algorithm for multiplying two 2 X 2 matrices,

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \quad \mathbf{X} \quad \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} \quad = \quad \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

$$\mathbf{C_{11} = A_{11}B_{11} + A_{12}B_{21}}$$
$$\mathbf{C_{12} = A_{11}B_{12} + A_{12}B_{22}}$$
$$\mathbf{C_{21} = A_{21}B_{11} + A_{22}B_{21}}$$
$$\mathbf{C_{22} = A_{21}B_{12} + A_{22}B_{22}}$$

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \mathbf{X} \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} \mathbf{A11B11 + A12B21} & \mathbf{A11B12 + A12B22} \\ \mathbf{A21B11 + A22B21} & \mathbf{A21B12 + A22B22} \end{bmatrix}$$

ANALYSIS

- In the naive Matrix Multiplication algorithm, there are 8 multiplications and 4 additions to multiply two 2X2 matrices.
- The addition of two nXn matrices takes O (n^2) time. The time complexity can be written as

$$\mathbf{T(n) = 8\,T\left(\frac{n}{2}\right) + O(n^2)}$$

By Master's Theorem, the time complexity of the naïve matrix multiplication algorithm is,

$$\mathbf{T(n) = O(n^3),\ Cubic.}$$

Strassen's Algorithm

- The idea of **Strassen's method** is to reduce the number of multiplications.
- Strassen's method is similar to simple divide and conquers method in the sense that this method also divides matrices to sub-matrices of size N/2 x N/2 and the sub-matrices of result are calculated using following formulae.

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_1 + P_5 - P_3 - P_7$$

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \quad \mathbf{x} \quad \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} \quad = \quad \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$

Where,

$$P_1 = A_{11}\,(B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12})\,B_{22}$$
$$P_3 = (A_{21} + A_{22})\,B_{11}$$
$$P_4 = A_{22}\,(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22})\,(B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22})\,(B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21})\,(B_{11} + B_{12})$$

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes O (N^2) time. So time complexity can be written as,

$$T(N) = 7\,T\left(\frac{N}{2}\right) + O(N^2)$$

By Master's Theorem, the time complexity of the naïve matrix multiplication algorithm is,

$$T(N) = O(N^{\log_2 7}) = O(N^{2.8})$$

Performance comparison

Input Size	Sequential Algorithm	Strassen's Algorithm
2	8	7
4	64	49
8	512	343
16	4096	2401
…	….	…
N	N^3	$N^{2.8}$

Strassen's Method is not preferred for practical applications for following reasons.

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The sub-matrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen's algorithm than in Naive Method.

PART – III: GREEDY TECHNIQUE

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seems best at the moment is chosen.
- Final output is an optimal solution.
- Greedy algorithms don't always yield optimal solutions but, when they do, they're usually the simplest and most efficient algorithms available.
- Hence, the proof of correctness is necessary.

ELEMENTS OF THE GREEDY STRATEGY

- There are two ingredients that are exhibited by most problems that lend themselves to a greedy strategy:
    1. The greedy-choice property and
    2. Optimal substructure.

1. Greedy-choice property

- The first key ingredient is the ***greedy-choice property***: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Here is where greedy algorithms differ from dynamic programming.
    - In dynamic programming, a choice at each step is made which may depend on the solutions to sub-problems.
    - In a greedy algorithm, whatever choice seems best at the moment and then solves the sub-problems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems.
    - Thus, unlike dynamic programming, which solves the sub-problems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

2. Optimal substructure

A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

Greedy method – Control abstraction

```
Algorithm Greedy ( A, n)
Inputs: A[1..n] contains the n inputs
Solution ← Ø
For i← 1 to n do
        X← Select(A)
        If Feasible(solution, x) then
                Solution ←Union(solution, x)
Return solution.
```

The Greedy function describes the essential way that a greedy algorithm will look , once a particular problem is chosen and the functions Select, Feasible and Unions are properly implemented.

Select () – this function selects an input from A[] and removes it. The selected input's value is then assigned into x.

Feasible () – is a boolean-valued function that determines whether x can be included into the solution vector.

Union () – It combines x with the solution and updates the objective function.

General Method

The Greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

On each step the choice made must be

Feasible – i.e it has to satisfy the problem's constraints

Locally optimal – i.e it has to be the best local choice among all feasible choices available on that step

Irrevocable – i.e., once made, it cannot be changed on subsequent steps of the algorithm.

1. KNAPSACK PROBLEM (A Fractional Knapsack)

Problem Description: Given n items of known weights, values and a knapsack of capacity, find the most valuable subset of the items that fit into the knapsack.

Fractional knapsack problem*: It is permitted to take fractions of items, meaning that the items can be broken into smaller pieces so that we decide to carry only a fraction of x_i of item i, where $0 \le x_i \le 1$.*

If a fraction x_i, $1 \le i \le n$, of item 'i' is placed into the knapsack, then a profit of $x_i p_i$ is earned also proportionally varied.

Input:

There are n items.

Each item i has its own Weight w_i, $1 \le i \le n$ and Value (profit) p_i, $1 \le i \le n$

The knapsack (bag) has a capacity W.

Objective function: The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the capacity of the knapsack is W, it require the total weight of all chosen items to be atmost W.

The objective is to maximize

$$\sum_{i=1}^{n} x_i \cdot p_i$$

subject to,

$\sum_{i=1}^{n} x_i . w_i \leq W$

and

$0 \leq x_i \leq 1$ and $1 \leq i \leq n$.

Output: A feasible solution is any set $(x_1,x_2,\ldots,x_n)$ satisfying the objective function. An optimal solution is a feasible solution for which the total profit is maximized.

Greedy Solution for Fractional Knapsack Problem:

1. Calculate the profit-per-weight, say Score $S_i = v_i / w_i$ for i = 1, 2, . . . , n.
2. Sort the items by decreasing S_i.
3. Let the sorted item sequence be 1, 2, . . . , i, . . . n, and the corresponding profit and weight be ρ_i and w_i respectively.
4. Let k be the current weight limit (Initially, k = W).
5. In each iteration,
    a. Choose item i from the head of the unselected list.
    b. If $k \geq w_i$, set $x_i = 1$ (we take item i) and reduce $k = k - w_i$, then consider the next unselected item.
    c. If $k < w_i$, set $x_i = k/w_i$ (take a fraction k/w_i of item i)

Algorithm

Algorithm ***Greedy_Fractional Knapsack*** (W, n)
Inputs: p[1...n] and w[1…n] contain the profits and weights respectively of the n items which are ordered such that (p[i] /w[i]) ≥ (p[i+1]/w[i+1]).
W is the knapsack capacity.
Output: x[1…n] - the solution vector.

```
For i ← 1 to n do
        x[i] ← 0; // Initialize the solution
K ←W;
For i ← 1 to n do
        If (w[i] > K) then break;
              x[i] ←1
              K ←K – w[i]
If (i≤n) then x[i] ←K/w[i]
```

Greedy Knapsack - Analysis

The optimal solution to this problem is to sort by the value of the item in decreasing order. Then pick up the most valuable item which also has a least weight. First, if its weight is less than the total weight that can be carried. Then deduct the total weight that can be carried by the weight of the item just pick. The second item to pick is the most valuable item among those remaining. Keep follow the same strategy until the knapsack is full.
If the items are already sorted into decreasing order of p_i / w_i, then the algorithm takes a time in *O(n)*; Therefore, the total time including the sort is in ***O(n log n)*.**

Proof of Correctness:

- One way to proof the correctness of the above algorithm is to prove the greedy choice property and optimal substructure property. It consist of two steps.

- First, prove that there exists an optimal solution begins with the greedy choice given above. The second part prove that if *A* is an optimal solution to the original problem *S*, then *A* - *a* is also an optimal solution to the problem *S* - *s* where *a* is the item thief picked as in the greedy choice and *S* - *s* is the sub-problem after the first greedy choice has been made. T
- The second part is easy to prove since the more valuable items have less weight.

Example: Consider the instance given by the following data:

Item	Weight	Profit
1	2	$12
2	1	$10
3	3	$20
4	2	$15

The knapsack capacity is 5.

Solution:

Given:

No of Items: n = 4

Weights of Items: w_1=2; w_2=1; w_3=3; w_4=2;

Profits of Items: P_1=12; P_2=10; P_3=20; P_4=15;

Knapsack capacity: W = 5.

Step – 1: calculate Score = Profit / weight for each item.

$S_1 = P_1 / w_1 = 12 / 2 = 6.0$

$S_2 = P_2 / w_2 = 10 / 1 = 10.0$

$S_3 = P_3 / w_3 = 20 / 3 = 6.66$

$S_4 = P_4 / w_4 = 15 / 2 = 7.5$

Step – 2: Sort the items by decreasing order of Score.

Item	Weight	Profit	Score = P_i/w_i
2	1	$10	10.0
4	2	$15	7.5
3	3	$20	6.66
1	2	$12	6.0

Step – 3: Find the Solution set.

Knapsack contains	Item	Weight	Profit	xi	$x_i.p_i$	x_i. w_i	Decision Item taken by
{Empty}	--	0	0	--	0	0	---
{2}	2	1	$10	1	10.0	1	Unit
{2, 4}	4	2	$15	1	15.0	2	Unit
{2, 4, 3}	3	3	$20	2/3	13.33	2	Fraction 2/3
{2,4,3}	1	2	$12	0	0	0	Dropped
{2,4,3}					38.33	5	

The Optimal Solution to the given instance is Xi = {1, 1, 2/3, 0} and

The maximum profit earned is 38.33

2. JOB SEQUENCING WITH DEADLINES

Problem description:

- There are n jobs to be processed on a machine.
- Each job has a deadline $d_i \geq 1$ and profit $p_i \geq 0$.
- The profit p_i is earned if and only if job-i is completed by its deadline.
- The job is completed if it is processed in a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on a machine.

Goal:

- A feasible solution is a subset of jobs J such that each job is completed by its deadline. The value of feasible solution J is the sum of the profits of the jobs in J or $\sum_{i \in J} P_i$. An optimal solution is a feasible solution with maximum profit value.
- Find a feasible schedule J whose profit is as large as possible;

Greedy approach to Job sequencing problem:

- This algorithm begins by sorting the jobs in order of decreasing (actually non-increasing) profits.
- Then, starting with the empty schedule, it considers the jobs one at a time;
- If a job can be (feasibly) added, then it is added to the schedule in the latest possible (feasible) slot.

Algorithm ***Greedy-Jobs(d, J, n)***
Input: The job P[i] with the deadline d[i]≥ 1 such that 1≤i≤n // total n jobs.
Output: Subset J the optimal sequence of jobs that gives the maximum profit.
Sort the jobs so that P[1] > P[2] > ….P[n]
J← ∅;
For i← 1 to n do
Schedule job P[i] in the latest possible free slot meeting its deadline;
if there is no such slot, do not schedule P[i].

Example: Let n=5, (p1, p2, …, p5) = (10, 15, 20, 5, 9) and (d1, d2, …, d5) = (2, 2, 1, 3, 3). Find the optimal sequence which gives maximum profit.

Step 1: Sorted sequence is (p3, p2, p1, p5, p4).

Solution set J	Assigned Slot	Job Considered	Action	Profit earned
∅	---	3	Assign to [0, 1]	0
{3}	[0, 1]	2	Assign to [1, 2]	20
{3, 2}	[0, 1], [1, 2]	1	Reject	35
{3, 2}	[0, 1], [1, 2]	5	Assign to [2, 3]	35
{3, 2, 5}	[0, 1], [1, 2], [2,3]	4	Reject	44

The Optimal solution is J = {3, 2, 5} with a profit of 44.

UNIT – 2

I.	***Dynamic Programming:*** The general method - Multistage graphs - Optimal BST - 0/1 knapsack – Traveling Salesperson Problem – Flow shop scheduling.
II.	***Backtracking:*** General Method – 8 Queens problem – Sum of Subsets problem – Graph Coloring problem – Hamiltonian Cycles – Knapsack Problem.
III.	***Branch and Bound:*** The Method, 0/1 Knapsack problem – Travelling Salesperson problem.

PART – I : DYNAMIC PROGRAMMING

- Dynamic programming is a fancy name for using divide-and-conquer technique with a table. As compared to divide-and-conquer, dynamic programming is more powerful and subtle design technique.
- This technique was developed back in the days when "programming" meant "tabular method" (like linear programming). It does not really refer to computer programming. Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions.
- The most attractive property of this strategy is that during the search for a solution it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution.
- Dynamic programming takes advantage of the duplication and arrange to solve each subproblem only once, saving the solution (in table or in a globally accessible place) for later use.
- The underlying idea of dynamic programming is: avoid calculating the same stuff twice, usually by keeping a table of known results of subproblems.
- Unlike divide-and-conquer, which solves the subproblems top-down, a dynamic programming is a bottom-up technique. The dynamic programming technique is related to divide-and-conquer, in the sense that it breaks problem down into smaller problems and it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient.
- The dynamic programming is among the most powerful for designing algorithms for optimization problem. This is true for two reasons. Firstly, dynamic programming solutions are based on few common elements. Secondly, dynamic programming problems are typical optimization problems i.e., find the minimum or maximum cost solution, subject to various constraints.
- In other words, this technique used for optimization problems:
  - ✓ Find a solution to the problem with the optimal value.
  - ✓ Then perform minimization or maximization.

The dynamic programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of caching subproblem solutions and appealing to the "principle of optimality."

Elements of Dynamic Programming

There are three basic elements that characterize a dynamic programming algorithm:

1. Substructure

Decompose the given problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems. Note that unlike divide-and-conquer problems, it is not usually sufficient to consider one decomposition, but many different ones.

2. **Table-Structure**

After solving the subproblems, store the answers (results) to the subproblems in a table. This is done because (typically) subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.

3. Bottom-up Computation

Using table (or something), combine solutions of smaller subproblems to solve larger subproblems, and eventually arrive at a solution to the complete problem. The idea of bottom-up computation is as follow:

Bottom-up means

i. Start with the smallest subproblems.
ii. Combining theirs solutions obtain the solutions to subproblems of increasing size.
iii. Until arrive at the solution of the original problem.

Two important elements that a problem must have in order for dynamic programming technique to be applicable

1. Optimal Substructure

- Show that a solution to a problem consists of making a choice, which leaves one or sub-problems to solve.
- Try all the choices of subproblems, solve all the subproblems resulting from each choice, and pick the choice whose solution, along the subproblem solutions, is best.

2. Polynomially many (Overlapping) Subproblems

- An important aspect to the efficiency of dynamic programming is that the total number of distinct sub-problems to be solved should be at most a polynomial number.
- Overlapping subproblems occur when recursive algorithm revisits the same problem over and over.
    1. Store, don't recomputed.
    2. Make a table indexed by subproblem.
    3. When solving a subproblem:
        - **Lookup** in the table.
        - If answer is there, **use it**.
        - Otherwise, **compute** answer, then **store it**.

GENERAL APPROACH – DYNAMIC PROGRAMMING

The development of a dynamic programming algorithm can be broken into a sequence of following four steps,

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information

THE PRINCIPLE OF OPTIMALITY

The dynamic programming relies on a principle of optimality. ***This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal***.

- The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its subproblems.
- The difficulty in turning the principle of optimally into an algorithm is that it is not usually obvious which subproblems are relevant to the problem under consideration.

Characterize the space of subproblems

- Keep the space as simple as possible.
- Expand it as necessary.

Optimal substructure varies across problem domains:

1. How *many subproblems* are used in an optimal solution.
2. How *many choices* in determining which subproblem(s) to use.

Running time of dynamic programming algorithms:

Informally, the running time of the dynamic programming algorithm depends on the overall number of subproblems times the number of choices. For example, in case of *optimal binary search tree problem*, we have $\Theta(n^2)$ sub-problems and $\Theta(n)$ choices for each implying $\Theta(n^3)$ running time.

Dynamic programming uses optimal substructure bottom up fashion:

- First find optimal solutions to subproblems.
- Then choose which to use in optimal solution to the problem.

1. MULTI – STAGE GRAPHS

- A multistage graph G=(V,E) is a directed graph in which vertices are partitioned into K>=2 disjoint set (set Vi) where [1<= i <= K].
- The sets V_1 and V_k are containing only one vertex, the Source and Sink vertex respectively.
- In addition, if (u, v) is an edge E then u ∈ Vi, v ∈ Vi+1 for some 1≤i≤k, and c(i, j) be the cost of edge (i, j).
- The cost of a path from (S to T) is the sum of costs of the edges on the path.

The Multistage graph problem is to find the minimum cost path from "S" to "T". The value on edges is called cost of edges.

Dynamic Programming Solution to Multi-stage Graph:

A dynamic programming formula for a k-stage graph problem is obtained by,

- Every path from S to T is the result of a sequence of (k-2) decisions.
- The i^{th} decision involves determining which vertex in Vi+1 stage, 1≤i≤(k-2) is to be on the path.
- By using the forward approach,

$$\textbf{Cost (i, j)} = \min_{l \in Vi+1} \{ c(j, l) + \textbf{cost}\ (i+1, l) \}$$

where,

cost(i, j) = cost(level, no. of nodes at that level)

'c' is a weight assigned on each edge,

'k' is the no. of stages

Steps:

1. Compute the cost (K-2, j) for all j∈ V_{k-2}
2. Compute the cost (K-3, j) for all j∈ V_{k-3}
3. Compute the cost (K-4, j) for all j∈ V_{k-4} and so on.
4. Finally the cost (1, S).

Algorithm K-stage Graph(G, k, n, P)

Input: A k-stage graph G = (V, E) with n vertices indexed in order of stages.

Output: P[1..k] is a minimum cost path.

Cost[n] = 0;

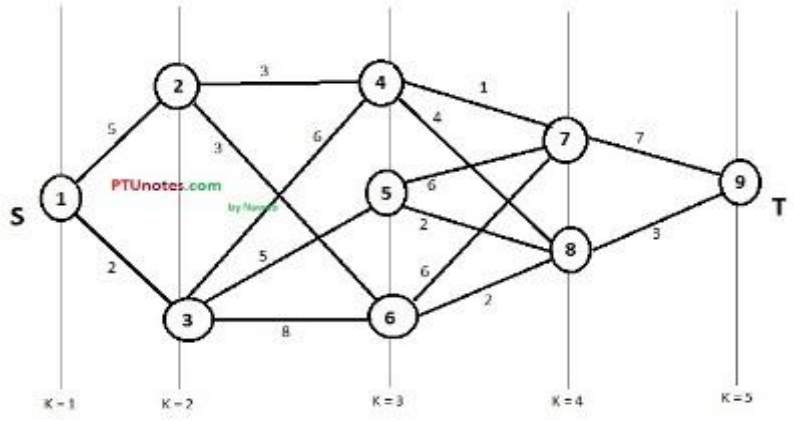
For j ← n-1 to 1 do step -1

Let r be a vertex such that (j,r) is an edge in G and C(j,r) + cost(r) is minimum.

Cost[j] = c(j,r) + cost (r)

d(j) = r;

Example: From the given multi-stage graph, calculate the minimum cost path from vertex s to vertex t.



Solution: By Forward approach,

$$\textbf{Cost (i, j)} = \min_{l \in Vi+1} \{ c(j, l) + \textbf{cost}\ (i+1, l) \}$$

Step 1: Cost (K-2, j)

- Three nodes are selected as 'j' because at stage K- 2=3, there are three nodes i.e. 4, 5 & 6. So that the value of i=3 and for j=4, 5 & 6.
- There are three options here to choose minimum Cost at stage K=3

Cost(3, 4)

Cost(3, 4) = min{ c(4, 7) + cost(7, 9) } or { c(4, 8) + cost(8, 9) }
Cost(3, 4) = min{ 1 + 7 } or { 4 + 3 }
Cost(3, 4) = min{ 8 } or { 7 }
Cost(3, 4) = 7

Cost(3, 5)

Cost(3, 5) = min{ c(5, 7) + cost(7, 9) } or { c(5, 8) + cost(8, 9) }
Cost(3, 5) = min{ 6 +7 } or { 2+ 3 }
Cost(3, 5) = min{ 13 } or { 5 }
Cost(3, 5) = 5

Cost(3, 6)

Cost(3, 6) = min{ c(6, 7) + cost(7, 9) } or { c(6, 8) + cost(8, 9) }
Cost(3, 6) = min{ 6 + 7 } or { 2 + 3 }
Cost(3, 6) = min{ 13 } or { 5 }
Cost(3, 6) = 5

From step 1, node 8 is selected as it gives the minimum path from stage K-2 to stage K.

Step 2: Cost (K-3, j)

- Two nodes are selected as 'j' because at stage K-3=2, there are two nodes i.e. 2 & 3. so the value of i=2 and for j=2 & 3.
- There are three options here to choose minimum Cost at stage K=2

Cost(2, 2)

Cost(2, 2) = min{ c(2, 4) + cost(4, 8) + cost(8, 9) } or { c(2, 6) + cost(6, 8) + cost(8, 9) }
Cost(2, 2) = min{ 3 + 4 + 3 } or { 3 + 2 + 3 }
Cost(2, 2) = min{ 10 } or { 8 }
Cost(2, 2) = 8

Cost(2, 3)

Cost(2, 3) = min{ c(3, 4) + cost(4, 8) + cost(8, 9) } or { c(3, 5) + cost(5, 8) + cost(8, 9) }
or { c(3, 6) + cost(6, 8) + cost(8, 9) }
Cost(2, 3) = min{ 6 + 4 + 3 } or { 5 + 2 + 3 } or { 8 + 2 + 3 }
Cost(2, 3) = min{ 13 } or { 10 } or { 13 }
Cost(2, 3) =10

From step 2, **node 6 is selected as it has minimum cost.**

Step 3: Cost (K-4, j)

- Only one node is selected as 'j' because at stage K-4=1, So, there are only one node i.e. 1 and the value of i=1 and for j=1.
- There are three options here to choose minimum Cost at stage K=1

Cost(1, 1)

Cost(1, 1) = min{ c(1, 2) + cost(2, 6) + cost(6, 8) + cost(8, 9) } or { c(1, 3) + cost(3, 6) + cost(6, 8) + cost(8, 9) }
Cost(1, 1) = min{ 5 + 3 + 2 + 3 } or { 2 + 8 + 2 + 3 }
Cost(1, 1) = min{ 13 } or { 15 }
Cost(1, 1) = 13

From step 3, node 2 is selected as it has minimum cost.

Decision table: 1 –> 2 –> 6 –> 8 –> 9 is having the minimum cost from 'S' to 'T'.

2. OPTIMAL BST

- An Optimal binary search tree (OBST) is a weight-balanced binary tree.
- It is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities).

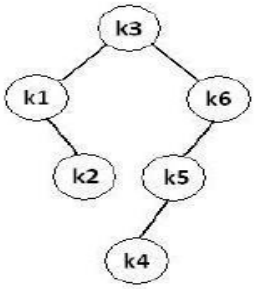
Given a set of N ordered elements and a set of 2N+1 probabilities.

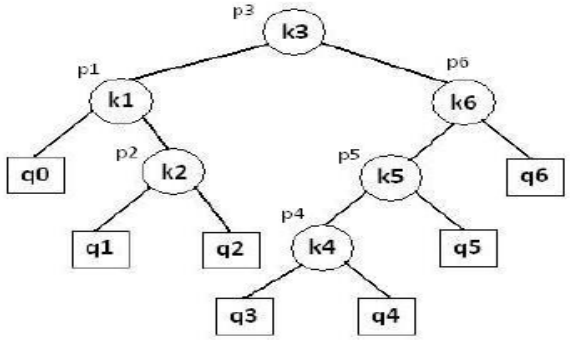
Let $k_1, k_2 \ldots k_n$ with $k_1 < k_2 < \ldots < k_n$ be the element keys and the probabilities P_1 through P_n and q_0 through q_n.

P_i is the probability of a search being done for element k_i.

For $1 < i < n$, q_i is the probability of a search being done for an element between k_i and k_{i+1}, q_0 is the probability of a search being done for an element strictly less than k_1, and q_n is the probability of a search being done for an element strictly greater than k_n.

These 2N+1 probabilities cover all possible searches, and therefore add up to one.





The total cost of any binary search tree for the set of keys $k_1, k_2 \ldots k_n$ is

$$\sum_{i=1}^{n} \mathbf{Pi} \,.\, \boldsymbol{level(ki)}$$

When only successful searches are made.

Since unsuccessful searches for keys not in the tree will also be made, the cost of unsuccessful saerches should also be included. So the cost of failure search nodes is

$$\sum_{i=0}^{n} \mathbf{qi} \,.\, \boldsymbol{level(ki - 1)}$$

Therefore the total cost of a BST is

$$\sum_{i=1}^{n} \mathbf{Pi} \,.\, \boldsymbol{level(ki)} + \sum_{i=0}^{n} \mathbf{qi} \,.\, \boldsymbol{level((ki) - 1)} \quad \text{…… (1)}$$

The optimal Binary Search tree for k_1, $k_2 \ldots k_n$ is the one that minimizes the total cost (i.e) Eqn (1) over all possible binary search trees for this set of keys.

To find the Optimal BST,

1. Generate each possible BST for the keys
2. Calculate the weighted path length.
3. Keep the tree with the smallest weighted path length.

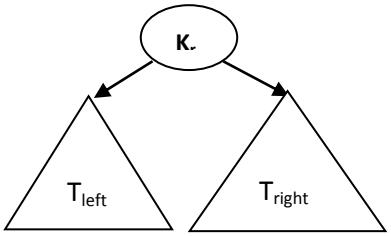
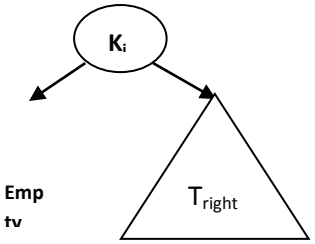
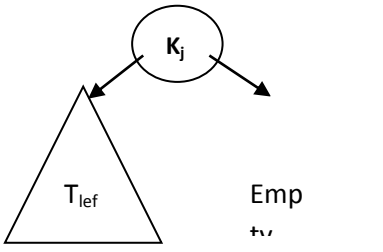
Step – 1: Structure of an OBST

To characterize the optimal structure of an OBST start with an observation about subtrees.

Consider any subtree of a BST containing keys K_i, ki_{+1}, … K_j for i≤j.

This subtree containing keys k_i through k_j must also have as its leaves d_{i-1}, di, d_{i+1}, …d_j

From the given keys k_i through k_j, one of the key K_r is designated as a root of the subtree.

Taking some key k_r as root: k_i, k_{i+1},… k_r,… k_{j-1}, k_j T_{left} includes keys K_i, K_{i+1}, …K_{r-1} & dummy keys d_{i-1}, d_i, …d_{r-1} whereas T_{right} includes keys K_{r+1}, … , K_{j-1}, …K_j & dummy keys d_r, …, d_{j-1}, d_j	
If Ki be the root then, T_{left} is empty. It contains only d_{i-1} whereas T_{right} includes keys K_{i+1}, … , K_{j-1}, …K_j & dummy keys d_i, …, d_{j-1}, d_j	
Similarly if Kj be the root then, T_{left} includes keys K_i, … , K_{j-1} & dummy keys d_{i-1}, …, d_{j-1} whereas T_{right} is EMPTY. It contains only d_j	

Hence the empty subtrees also to be considered for computing the optimal search cost of the tree.

Step 2: Computing the Expected Cost

When j =i-1, there are no actual keys (ie) it represents the empty subtree.
The cost of tree containing key between k_i and k_j is

$$C_{i,j} = P_k + \text{Cost}(T_L) + \text{Cost}(T_R) + \text{Weight}(T_L) + \text{Weight}(T_R)$$

Where

Weight (T_L) is the weighted sum of probabilities of the Left subtree
Weight (T_R) is the weighted sum of probabilities of the Sum subtree

Step – 3: Computing the Optimal Search Cost

Let $C_{i,j}$ denote the cost of an optimal BST containing keys k_i through k_j. The Cost of the Optimal BST with k_r as its root

$$C_{0,n} = MIN_{0<k\le n}\{Pk + (q0 + \sum_{i=1}^{k-1}(Pi + qi) + C1, k-1\} + \{qk + \sum_{i=k+1}^{n}(Pi + qi) + Ck + 1, n\}$$

By generalizing the above equation,

$$C_{i,j} = MIN_{i<k\le j} \{Pk + (qi - 1 + \sum_{m=i}^{k-1}(Pm + qm) + Ci, k-1\} + \{qk + \sum_{m=k-1}^{j}(Pm + qm) + Ck + 1, j\}$$

So,

$$C_{i,i} = W_{i,j} \text{ for } j=i \;\&\; C_{i,j} = W_{i,j} + MIN_{i<k\le j}(Ci, k-1 + Ck, j), \text{ for } i<j.$$

Notations used:

- OBST(i, j) denotes the optimal binary search tree containing the keys ki,ki+1, …, kj.
- $W_{i,j}$ denotes the weight matrix for OBST(i, j)
- $W_{i,j}$ can be defined using the following formula:

$$W_{i,j} = \sum_{k=i+1}^{j} Pk + \sum_{k=i}^{j} qk$$

- $C_{i,j}$, $0 \le i \le j \le n$ denotes the cost matrix for OBST(i, j)
- $C_{i,j}$ can be defined recursively, in the following manner:

$$C_{i,i} = W_{i,j} \text{ for } j=i \;\&\; C_{i,j} = W_{i,j} + MIN_{i<k\le j}(Ci, k-1 + Ck, j), \text{ for } i<j$$

- $R_{i,j}$, $0 \le i \le j \le n$ denotes the root matrix for OBST(i, j)
- Assigning the notation $R_{i,j}$ to the value of k for which we obtain a minimum in the above relations, the optimal binary search tree is OBST(0, n) and each subtree OBST(i, j) has the root k R_{ij} and as subtrees the trees denoted by OBST(i, k-1) and OBST(k, j).
- *OBST(i, j) will involve the weights qi-1, pi, qi, …, pj, qj.

Example: please refer class work note book.

3. TSP – TRAVELING SALESPERSON PROBLEM

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the *shortest possible route that visits every city exactly once* and returns to the starting point .

Let G = (V, E) be a directed graph with edge costs Ci,j, the variable Ci,j is defined such that

Ci,j > 0 for all i and j
Ci,j =∞, if (i,j) is not in G

- A tour of G is a directed simple cycle that includes every vertex in V. The cost of a vertex is the sum of the cost of edges on the tour. The traveling salesperson problem is to find the tour of minimum tour.
- A tour can be a simple path that starts and ends at vertex 1. Every tour consists of an edge (1,k) for some k ∈ V-{1} and a path from vertex k to vertex 1.
- The path from vertex 1 goes through each vertex V – {1,K} exactly once. It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest path from K to 1 going through all vertices in V – {1, K}. Hence the principle of optimality holds.
- ***Let g(j,S) be t he length of a shortest path starting at vertex j, going through all vertices in S and terminating at 1.***
- *From the principle of optimality it follows that*

  $g(1, V\text{-}\{1\}) = \min_{2 \le k \le n} [C_{1,k} + g\{k, V - \{1,k\})]$

  Generalizing the above eqn, the g() values can be obtained by,

  $\boldsymbol{g(j, S) = \min_{k \in S}}$ **$[C_{j,k} + g\{k, V - \{k\})]$** for j not belongs to S.

where,

- *g(x, S)* - starting from 1, min cost path ends at vertex x, passing vertices in set S exactly once
- c_{xy} - edge cost ends at x from y
- *p(x, S)* - the vertex to x from set S. Used for constructing the TSP path back at the end.

- **Steps:**
  1. Find g(j,Ø) for all 2≤j≤n
  2. Obtain g(j, S) for all S of size 1.
  3. Obtain g(j, S) for all S with size 2, and then size 3 and so on.
  4. Finally obtain g(1, V – {1}) is the length of an optimal salesperson tour.

Example: From the given cost matrix, find the optimal tour with minimum path cost.

0	**2**	**9**	**10**
1	**0**	**6**	**4**
15	**7**	**0**	**8**
6	**3**	**12**	**0**

Step 1: k = 1, for NULL set,

$g(2, Ø) = c_{21} = 1$

$g(3, Ø) = c_{31} = 15$

$g(4, Ø) = c_{41} = 6$

Step 2: K=1, for sets of size 1

Set {2}:

$g(3,\{2\}) = c_{32} + g(2, \emptyset) = c_{32} + c_{21} = 7 + 1 = 8$ $p(3,\{2\}) = 2$

$g(4,\{2\}) = c_{42} + g(2, \emptyset) = c_{42} + c_{21} = 3 + 1 = 4$ $p(4,\{2\}) = 2$

Set {3}:

$g(2,\{3\}) = c_{23} + g(3, \emptyset) = c_{23} + c_{31} = 6 + 15 = 21$ $p(2,\{3\}) = 3$

$g(4,\{3\}) = c_{43} + g(3, \emptyset) = c_{43} + c_{31} = 12 + 15 = 27$ $p(4,\{3\}) = 3$

Set {4}:

$g(2,\{4\}) = c_{24} + g(4, \emptyset) = c_{24} + c_{41} = 4 + 6 = 10$ $p(2,\{4\}) = 4$

$g(3,\{4\}) = c_{34} + g(4, \emptyset) = c_{34} + c_{41} = 8 + 6 = 14$ $p(3,\{4\}) = 4$

Step 3: K=2, for sets of size 2

Set {2,3}:

$g(4,\{2,3\}) = \min \{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} = \min \{3+21, 12+8\} = \min \{24, 20\} = 20$

$p(4,\{2,3\}) = 3$

Set {2,4}:

$g(3,\{2,4\}) = \min \{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} = \min \{7+10, 8+4\} = \min \{17, 12\} = 1$

$p(3,\{2,4\}) = 4$

Set {3,4}:

$g(2,\{3,4\}) = \min \{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} = \min \{6+14, 4+27\} = \min \{20, 31\} = 20$

$p(2,\{3,4\}) = 3$

Length of an optimal tour:

$g(1,\{2,3,4\}) = \min \{ c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\}) \}$

$= \min \{2 + 20, 9 + 12, 10 + 20\}$

$= \min \{22, 21, 30\}$

$= 21$

Successor of node 1: p (1, {2, 3, 4}) = 3

Successor of node 3: p (3, {2, 4}) = 4

Successor of node 4: p (4, {2}) = 2

The optimal TSP tour reaches: 1 → 3 → 4 → 2 → 1

Analysis:

Let N be the number of g(j, S) values that have to be computed before to compute g(1, V – {1}),

- For each value of S there are n-1 choices for i.
- The number of distinct sets S of size k not including 1 and j is $\binom{n-2}{k}$
- Hence, N = $\sum_{k=0}^{n-2}(n-k-1)\binom{n-2}{k} = (n-1)\,2^{n-2}$

To find the optimal tour this algorithm requires **O ($n^2 2^n$)** time as the computation of g(j,S) with |S| = k requires k-1 comparisons.

4. BINARY KNAPSACK (0 / 1)

Problem description

Given n items of known weights w1, w2, …, wn and values v1, v2, …, vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Dynamic programming approach to binary knapsack

- Derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solution to its subproblem.
- Consider an instance of defined by the first p items, 1≤p≤n, with the weights w1, w2, …, wp; values v1, v2, …, vp and the knapsack of capacity q, 1≤q≤W.
- Let V[p,q] be the value of an optimal solution to that instance, then the subset defined by V[p,q] that fit the knapsack of capacity q is divided into two categories as
    1. Among the subsets that do not include the p^{th} item, the value of an optimal subset is defined as **V[p-1, q].**
    2. Among the subsets that do include the p^{th} item (hence q - $w_p \geq 0$), an optimal subset is made up of this item and an optimal subset of p-1 items that fit into the knapsack of capacity q-w_p. The value of such an optimal subset is defined as **v_p + V[p-1, q-w_p].**
- Thus the value of an optimal solution among all feasible subsets of the first p items is the maximum of these two values.
- If the pth item does not fit into the knapsack, the value of an optimal subset selected from the first p items is the same as the value of an optimal subset selected from the first p-1 items.
- This leads to the following recurrence, (generalized)

$$V[i,j] = \begin{cases} \textbf{Max } \{ V[i-1,j], v_i + V[i-1, j-w_i] \}, & \text{if } j-w_i \geq 0 \\ V[i-1, j] \text{ if if } j-w_i < 0 \end{cases}$$

with the initial conditions that,
V[0,j] = 0 for all j≥0
V[i,0] = 0 for all i≥0

Goal: To find V[n,W] – the maximal value of a subset of the n items that fit into the knapsack of capacity W.

Table for solving the knapsack problem by dynamic programming.

0	1	….	j-wi	…	j	….	…..	W
0	0	0	0	0	0	0	0	0
….								
i-1	0		**V[i-1, j-wi]**		**V[i-1, j]**			
i	0				**V[i,j]**			
…..	0							
n	0							

The maximal value is V[n, W].

To find the composition of optimal subset by tracking the computations of the table,

If V[n, W]≠ V[n-1, W] then

choose item n; So the knapsack capacity remaining is W – w_n

else drop the item n.

Then if V[n-1, remaining capacity] V[n-2, remaining capacity] then

Choose iten n-1; so the knapsack capacity if remaining is W – w_{n-1}

else drop the item n-1.

and so on.

Algorithm BinaryKnapsack(i,j)

Input: A positive integer i indicating the number of the first items being considered and j indicating the knapsack capacity.

Output: The value of an optimal feasible solution of the first i-items.

```
If V[i,j] < 0
    If j<Weigt[i]
        Value ← BinaryKnapsack(i-1, j)
    Else
        Value ← MAX(Knapsack(i-1, j), value[i]+binaryknapsack(i-1, j-weight[i]))
    V[i,j] ← value.
Return V[i,j];
```

Analysis:

The time efficiency of this algorithm is O(nW). The time needed to find the composition of an optimal solution is in O(n+W).

Example: Consider the following instance, with the capacity of 10

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

Calculations: please refer classwork notebook.

W = 10

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

Composition of an optimal Subset:

Solution Set	Value comparisons	Action taken	Knapsack capacity remaining W = 10
∅	---	---	10
∅	Is V[5, 10] and V[4,10] equal 19 ≠ 18	Choose the item I_5	10 – 4 = 6
$\{I_5\}$	Is V[4, 6] and V[3,6] equal 12 = 12	Drop the item I_4	6
$\{I_5\}$	Is V[3, 6] and V[2,6] equal 12 ≠ 10	Choose the item I_3	6 – 1 = 5
$\{I_5, I_3\}$	Is V[2, 5] and V[1,5] equal 9 ≠ 6	Choose the item I_2	5 – 3 = 2
$\{I_5, I_3, I_2\}$	Is V[1, 2] and V[0,2] equal 4 ≠ 0	Choose the item I_1	2 – 2 = 0, Knapsack full
$\{I_5, I_3, I_2, I_1\}$	V[0, 0] , Knapsack full	Drop the item I_0	0

The optimal solution is: (I_1, I_2, I_3, I_5)

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 (x1, ... , x_n) where each x_i Є 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, ... , 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.

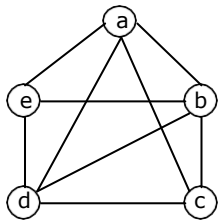
Depth first node generation with bounding functions is called ***backtracking***. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

Planar Graphs:

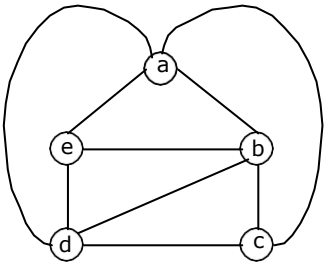
When drawing a graph on a piece of a paper, we often find it convenient to permit edges to intersect at points other than at vertices of the graph. These points of interactions are called crossovers.

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:



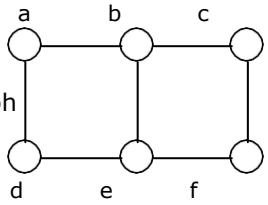
the following graph can be redrawn without crossovers as follows:

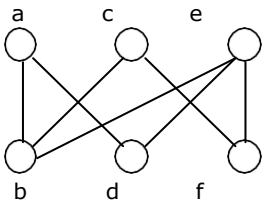


Bipartite Graph:

A bipartite graph is a non-directed graph whose set of vertices can be portioned 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:

The graph 

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 x 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, ... , 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 45^0 diagonal if:

  $$i - j = k - l.$$

  This implies, j – l = i – k
- Diag 135 conflict:

  $$i + j = k + l.$$

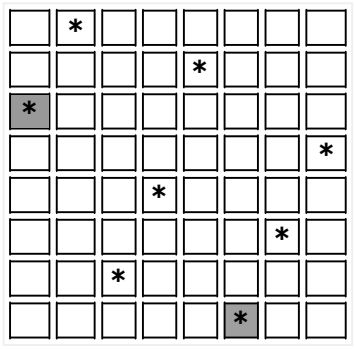
  This implies, j – l = k – i

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

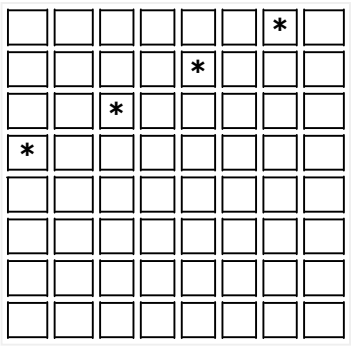
Let us consider for the case whether the queens on 3^{rd} row and 8^{th} row are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$$
$$\Rightarrow 5 = 5$$

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 then 8, repeat Step 1.

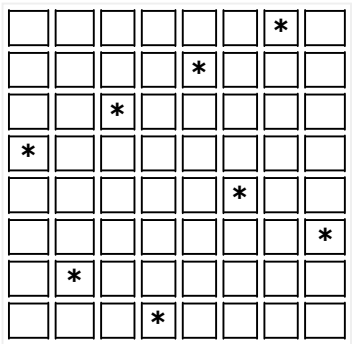
Step 3:

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 - l| = |1 – 2| = 1 |i – k | = |4 – 5| = 1
7	5	3	1	4				
7*	5	3	1	4	2*			|j - l| = |7 – 2| = 5 |i – k | = |1 – 6| = 5
7	5	3*	1	4	6*			|j - l| = |3 – 6| = 3 |i – k | = |3 – 6| = 3
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		|j - l| = |4 – 2| = 2 |i – k | = |5 – 7| = 2
7	5	3	1	4*	8	6*		|j - l| = |4 – 6| = 2 |i – k | = |5 – 7| = 2
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*
7	5	3	1	6				
7*	5	3	1	6	2*			|j - l| = |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 - l| = |3 – 8| = 5 |i – k| =|3 – 8| = 5
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

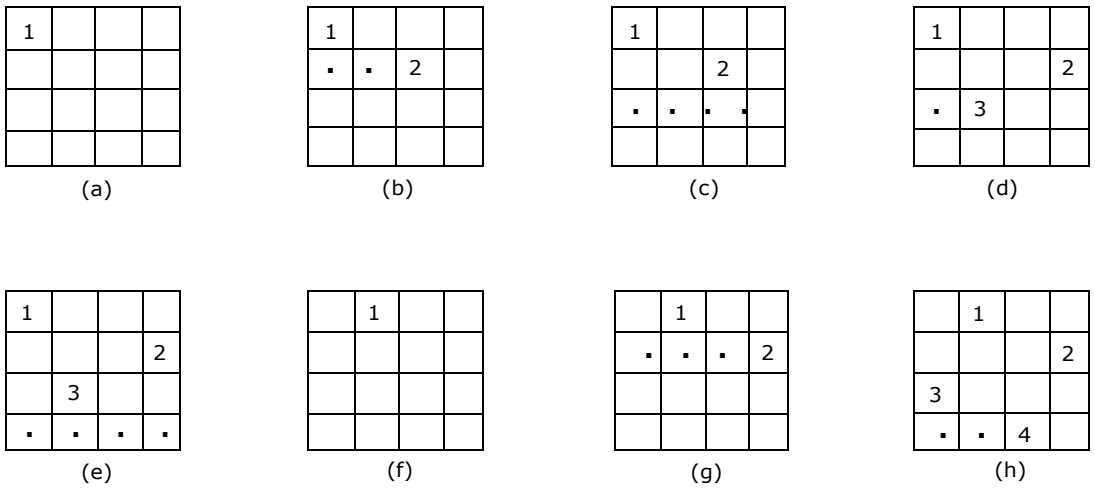
* indicates conflicting queens.

On a chessboard, the **solution** will look like:



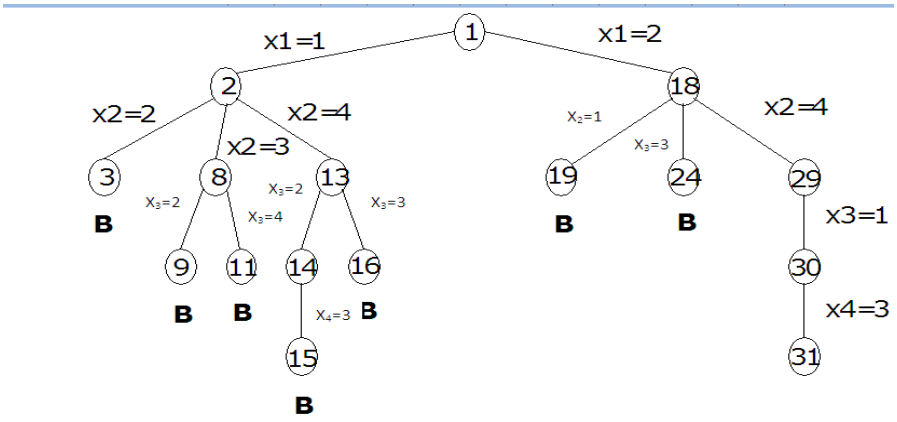
4 – Queens Problem:

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:

$$1 + n + n^2 + n^3 + \ldots\ldots\ldots\ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

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		Combination: 2
Row = 1	column = 2	3
Row = 2	column = 4	1
Row = 3	column = 1	4
Row = 4	column = 3	2

For N = 8, there will be 92 combinations.

Sum of Subsets:

Given positive numbers wi, $1 \le i \le n$, and m, this problem requires finding all subsets of w_i whose sums are 'm'.

All solutions are k-tuples, $1 \le k \le n$.

Explicit constraints:

- $x_i \in \{j \mid j \text{ is an integer and } 1 \le j \le n\}$.

Implicit constraints:

- No two x_i can be the same.
- The sum of the corresponding w_i's be m.
- $x_i < x_{i+1}$, $1 \le i < k$ (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).

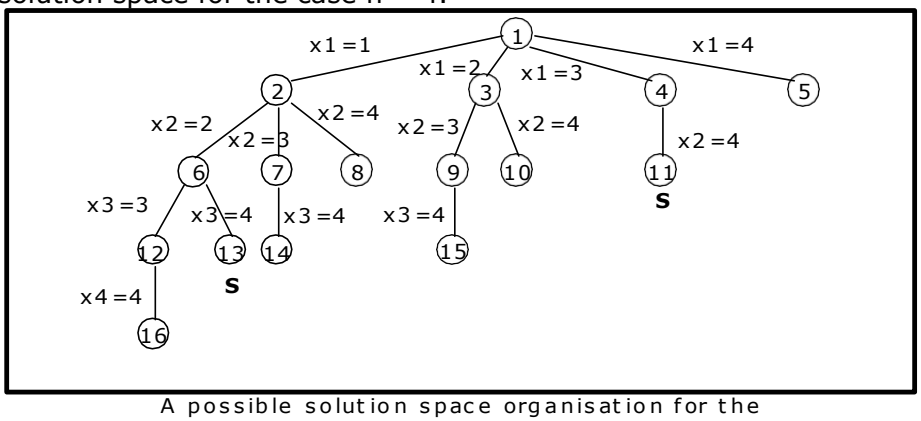
A better formulation of the problem is where the solution subset is represented by an n-tuple $(x_1, \ldots, x_n)$ such that $x_i \in \{0, 1\}$.

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 mot 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-colorabiltiy 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, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., 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)

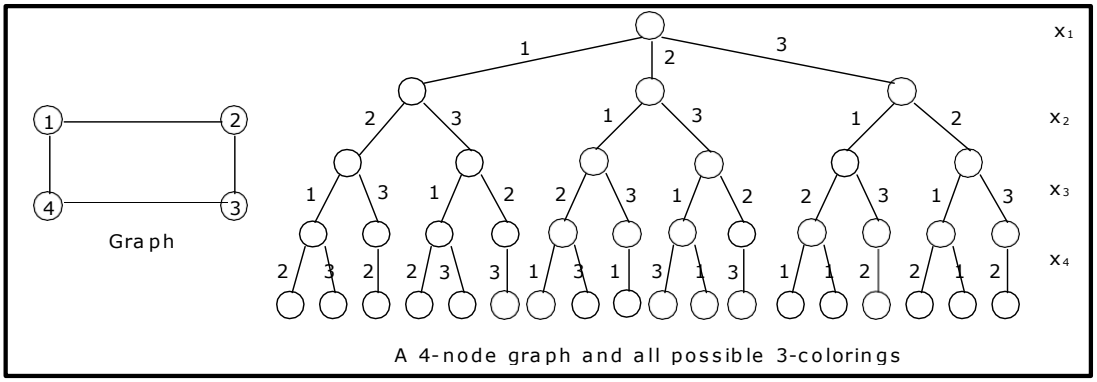
```
// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
// 1, 2, . . . . . , m to the vertices of the graph such that adjacent vertices are assigned
// distinct integers are printed. k is the index of the next vertex to color.
{
        repeat
        {                                       // Generate all legal assignments for x[k].
                NextValue (k);          // Assign to x [k] a legal color.
                If (x [k] = 0) then return;     // No new color possible
                If (k = n) then         // at most m colors have been
                                        // used to color the n vertices.
                        write (x [1: n]);
                        else mcoloring (k+1);
                } until (false);
        }
```

Algorithm NextValue (k)

```
// x [1] , . . . . x [k-1] have been assigned integer values in the range [1, m] such that
// adjacent vertices have distinct integers. A value for x [k] is determined in the range
// [0, m].x[k] is assigned the next highest numbered color while maintaining distinctness
// from the adjacent vertices of vertex k. If no such color exists, then x [k] is 0.
{
        repeat
        {
                x [k]: = (x [k] +1) mod (m+1)                   // Next highest color.
                If (x [k] = 0) then return;                     // All colors have been used
                for j := 1 to n do
                {       // check if this color is distinct from adjacent colors
                        if ((G [k, j] ≠ 0) and (x [k] = x [j]))
                        // If (k, j) is and edge and if adj. vertices have the same color.
                        then break;
                }
                if (j = n+1) then return;               // New color found
        } until (false);                        // Otherwise try to find another color.
}
```

Example:

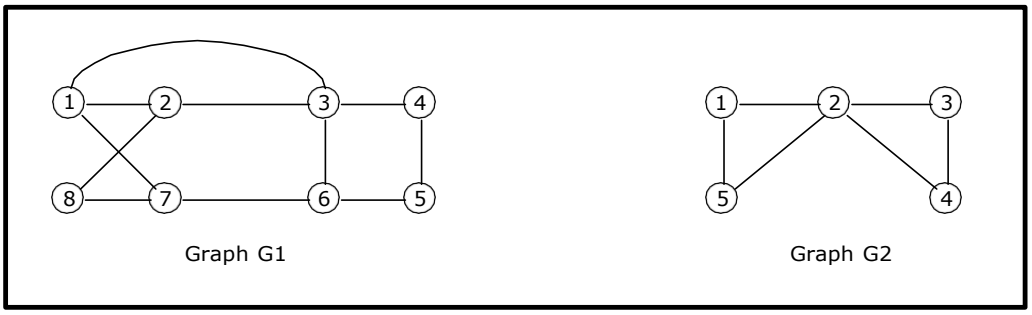
Color the graph given below with minimum number of colors by backtracking using state space tree.



A 4-node graph and all possible 3-colorings

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 $v_1, v_2, \ldots\ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct expect 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 $(x_1, \ldots\ldots x_n)$ 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 $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to k_{x-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 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
        {                                                       // Generate values for x [k].
                NextValue   (k);                    //Assign a legal Next value to x [k].
        if (x [k] = 0) then return;
                if (k = n) then write (x [1: n]);
                else Hamiltonian (k + 1)
        } until (false);
}
```

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:

$$\sum_{1 \le i \le n} w_i \ x_i \le m \ \ and \ \ \sum_{1 \le i \le n} p_i \ x_i \ \ is \ \max imized.$$

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.

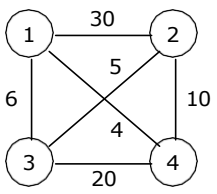
We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i, $1 \le i \le k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements $x_i = 0$ or 1.

(Knapsack problem using backtracking is solved in branch and bound chapter)

7.8 Traveling Sale Person (TSP) using Backtracking:

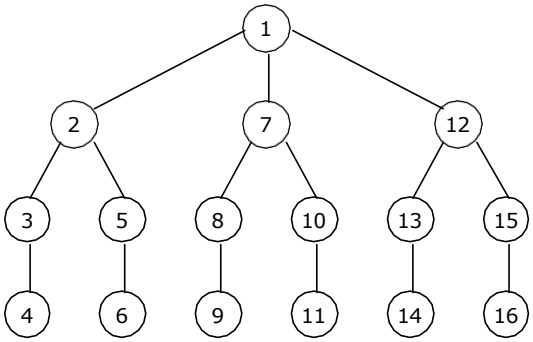
We have solved TSP problem using dynamic programming**.** In this section we shall solve the same problem using backtracking.

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, … ,4}, 1 forms a tour with some cost which should be minimum. The vertices shown as {2, 3, …. ,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 braches 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-an-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 in 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 $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

where, $\hat{c}(x)$ is the cost of x.

h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

$\hat{g}(x)$ is an estimate of the additional effort needed to reach an answer node from x.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(.)$ is called a LC–search (Least Cost search)

BFS and D-search are special cases of LC-search. If $\hat{g}(x) = 0$ and f(h(x)) = 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 $\hat{g}(x) > \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search.

An LC-search coupled with bounding functions is called an LC-branch and bound search

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 $\hat{c}(.)$ 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 Least() and Add() to delete and add a live node from or to the list of live nodes, respectively.

Least() finds a live node with least 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 $\hat{c}(.)$ such that $\hat{c}(x) \leq c(x)$ 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 $c(x) \geq \hat{c}(x) >$ 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 $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ 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.

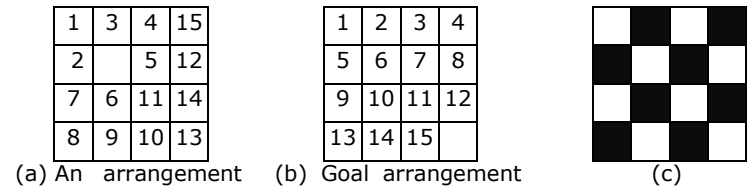
Position i is the frame position containing title numbered i in the goal arrangement of Figure 8.1(b). Position 16 is the empty spot. Let position(i) be the position number in the initial state of the title number i. Then position(16) will denote the position of the empty spot.

For any state let:

less(i) be the number of tiles j such that j < i and position(j) > position(i).

The goal state is reachable from the initial state iff: $\sum_{i=1}^{16} less(i) + x$ 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.



(a) An arrangement (b) Goal arrangement (c)

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 $\hat{c}(x)$ of c(x) is as follows:

$$\hat{c}(x) = f(x) + \hat{g}(x)$$

where, f(x) is the length of the path from the root to node x and

$\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. Here, $\hat{g}(x)$ 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 $\hat{c}(x)$.

$\hat{c}(2) = 1 + 4 = 5$
$\hat{c}(3) = 1 + 4 = 5$
$\hat{c}(4) = 1 + 2 = 3$ and
$\hat{c}(5) = 1 + 4 = 5$.

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:

$\hat{c}(10) = 2 + 1 = 3$
$\hat{c}(11) = 2 + 3 = 5$ and
$\hat{c}(12) = 2 + 3 = 5$.

The live node with least $\hat{c}$ 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 $\hat{c}()$, LC-search will be far more selective than any of the other search methods.

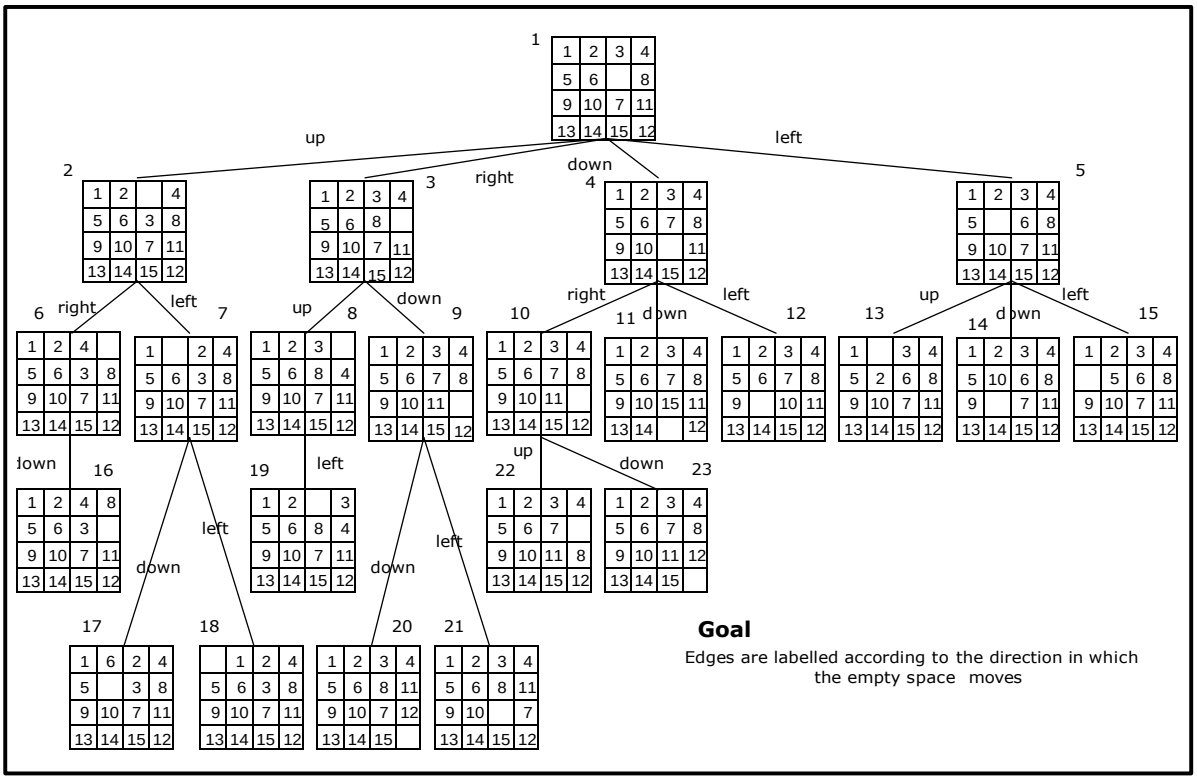


Figure 8.2. Part of the state space tree for 15-puzzle problem

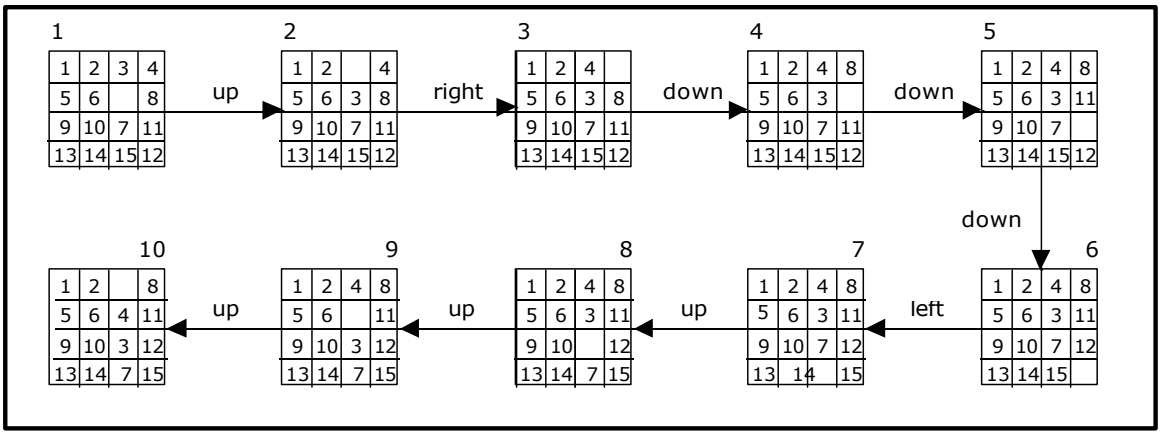


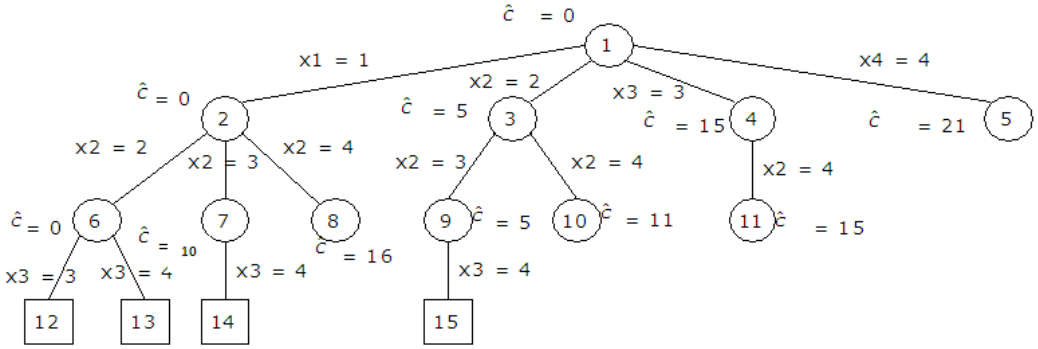
Figure 8. 3. First ten steps in a depth first search

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) = ∞ 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 $\hat{c}(x) = \sum_{\substack{i<m \\ i \neq s_x}} p_i$ 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 $u(x) = \sum_{i \notin S_x} p_i$. The u(x) is the cost of the solution S_x corresponding to node x.

S(2) = {1}	M =1	$\hat{c}(2) = \sum_{\substack{i<m \\ i \neq sx}} p_i = 0$
$S_{(3)}$ = {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	$\hat{c}(6) = \sum_{\substack{i=1,2 \\ i \in s_x}} p_i = \sum_{\substack{i<1 \\ i \neq s(6)}} p_i = 0$
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	$\hat{c}(8) = \sum_{\substack{i<4 \\ i \notin s(8)}} p_i = p_2 + p_3 = 10 + 6 = 16$
S(9)={2,3}	m=3	$\hat{c}(9) = 5$
S(10)={2,4}	m=3	$\hat{c}(10) = 11$
S(11)={3,4}	m=4	$\hat{c}(11) = 15$

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 = $\propto$ 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 $\hat{c}(4)$ and $\hat{c}(5)$ 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 $\hat{c}(7) = 10$ > 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 $\hat{c}(10) = 11$ > 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 $\hat{c}(x)$ > 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 $\hat{c}(x)$ > upper are distributed in some random way in the queue. Instead, live nodes with $\hat{c}(x)$ > upper can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ 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 upper = ∞ 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 $\hat{c}(4)$ > upper and $\hat{c}(5)$ > upper.

Node 2 is the next E-node as $\hat{c}(2) = 0$ and $\hat{c}(3) = 5$. Nodes 6, 7 and 8 are generated and upper is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10$ > 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 $\hat{c}(6) = 0 < \hat{c}(3)$. 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 $\hat{c}(10) = 11$ 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 $= 1 \overset{2}{\rightarrow} 3 \overset{3}{\rightarrow} 9 = 5 + 3 = 8$

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 <i, j>. $c_{ij} = \infty$ if <i, j> $\notin$ 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, π, 1 | π is a

permutation of (2, 3, . . . , n)} and |S| = (n – 1)!. The size of S can be reduced by restricting S so that (1, i_1, i_2, ... i_{n-1}, 1) $\in$ S iff <i_j, i_{j+1}> $\in$ 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 $\propto$ 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 <i, j> 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 $\propto$.

   b) Set A (j, 1) to $\propto$.

   c) Reduce all rows and columns in the resulting matrix except for rows and column containing only $\propto$. Let r is the total amount subtracted to reduce the matrix.

   c) Find $\hat{c}(S) = \hat{c}(R) + A(i, j) + r,$ where 'r' is the total amount subtracted to reduce the matrix, $\hat{c}(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $\hat{c}(S)$ 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

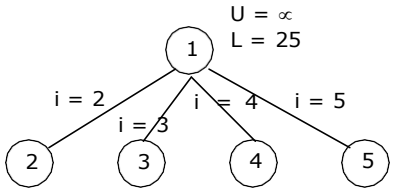
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.

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 $\propto$.

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 $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0 = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1, 2) + r$

$\hat{c}(S)$ = 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 $\bar{c}(S) = \bar{c}(R) + A(1, 3) + r$

$\bar{c}(S)$ = 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 $\hat{c}(S) = \hat{c}(R) + A(1, 4) + r$

$\hat{c}(S)$ = 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 ∞.

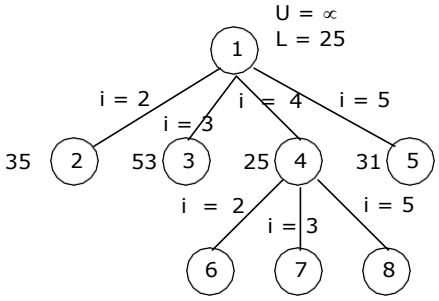
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction (r) = 5 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1, 5) + r$

$\hat{c}(S)$ = 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 $\hat{c}(S) = \hat{c}(R) + A(4,2) + r$

$\hat{c}(S)$ = 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 $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\bar{c}(S) = \bar{c}(R) + A(4,3) + r$

$\bar{c}(S)$ = 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 ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

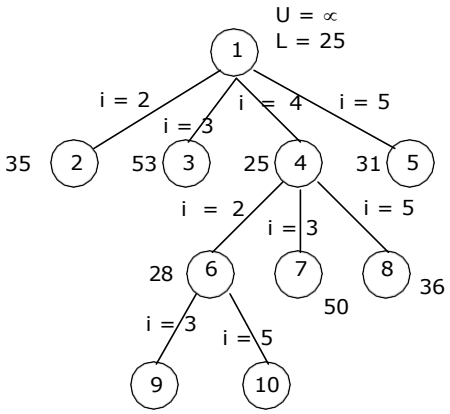
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$

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 $\hat{c}(S) = \hat{c}(R) + A(2,3) + r$

$\hat{c}(S)$ = 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 ∞.

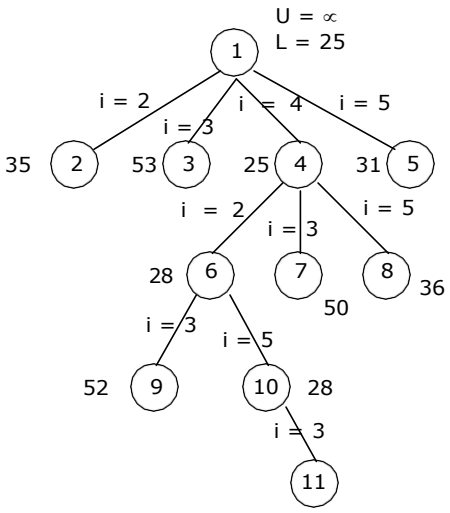
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2,5) + r$

$\hat{c}(S)$ = 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 ∞.

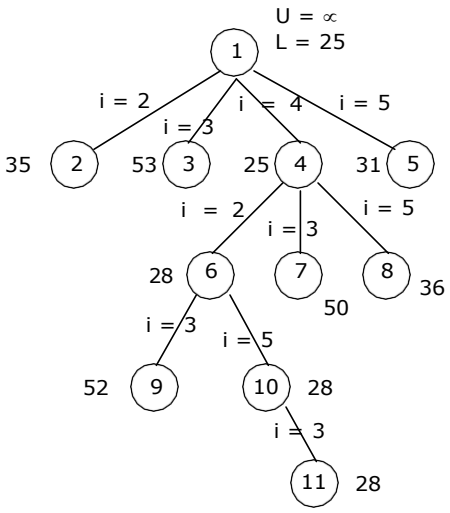
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(5, 3) + r$

$\hat{c}(S)$ = 28 + 0 + 0 = 28

The overall tree organization is as follows:



The path of traveling sale person problem is:

$1 \longrightarrow 4 \longrightarrow 2 \longrightarrow 5 \longrightarrow 3 \longrightarrow 1$

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.

Profit = $P_1 + P_2 + P_3$ = 10 + 10 + 12

So, Upper bound = 32

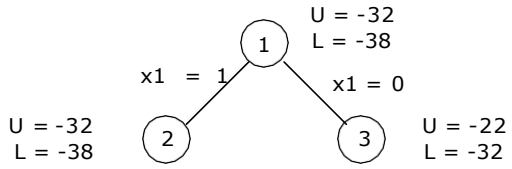
To calculate lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound = $10 + 10 + 12 + (\frac{3}{9} X\ 18) = 32 + 6 = 38$

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.

Therefore, Upper bound (U) = -32
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, make it as -32

L = 10 + 10 + 12 + $\frac{3}{9}$ x 18 = 32 + 6 = 38, make it as -38

For node 3, x_1 = 0, means we should not place first item in the knapsack.

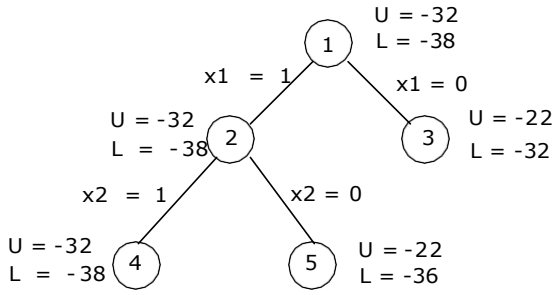
U = 10 + 12 = 22, make it as -22

L = 10 + 12 + $\frac{5}{9}$ x 18 = 10 + 12 + 10 = 32, make it as -32

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

For node 2, U – L = -32 + 38 = 6
For node 3, U – L = -22 + 32 = 10

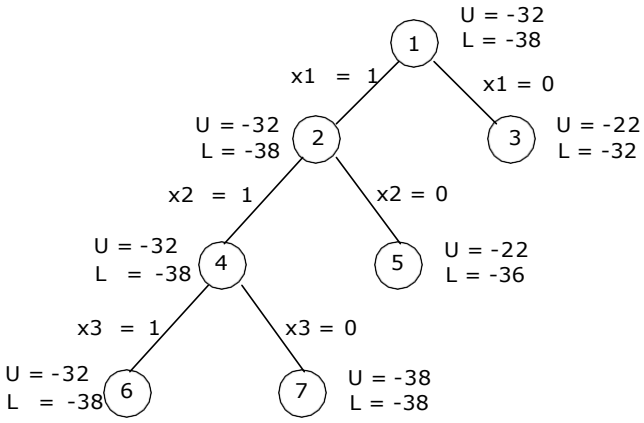
Choose node 2, since it has minimum difference value of 6.



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.

For node 4, U – L = -32 + 38 = 6
For node 5, U – L = -22 + 36 = 14

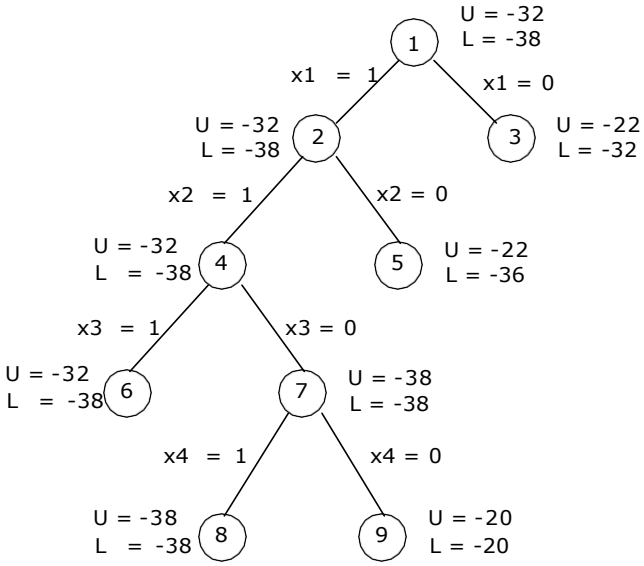
Choose node 4, since it has minimum difference value of 6.



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:

$$\sum P_i x_i = 10 \times 1 + 10 \times 1 + 12 \times 0 + 18 \times 1$$
$$= 10 + 10 + 18 = 38.$$

UNIT – III

I. **Internet Algorithms:** Strings and pattern matching algorithm – Tries: Text Compression – Text Similarity testing.

II. **Introduction to Parallelism models:** Simple Algorithms for parallel computers – CRCW and EREW algorithms.

III. **Probabilistic Algorithms:** Expected versus average time – Pseudorandom generation – Buffon's needle – Numerical Integration – Probabilistic Counting – Monte Carlo Algorithms.

PART – I: STRING AND PATTERN MATCHING ALGORITHMS

Introduction

- Document processing is rapidly becoming one of the dominant functions of computers.
- Computers are used to edit documents, to search documents, to transport documents over the Internet and to display documents on printers and computer screens.
- Web "surfing" and Web searching are becoming significant and important computer applications and many of the key computations in all of this document processing involve character strings and string pattern matching.

1. THE PATTERN MATCHING PROBLEM

- In the classic Pattern matching problem on strings, given a text string T of length n and a pattern string P of length m, to find whether P is a substring of T.
- The notion of a "match" is that there is a substring of T starting at some index i that matches P, character by character, so that **P[0] =T [i], P [1] =T [i+1], …, P [m-1]=T [i+m-1**], (i. e) the pattern **P = T [i…i+m-1].**
- Thus, the output from a pattern matching algorithm is either an indication that the pattern P does not exist in T or the starting index in T of a substring matching P.
- **Example:** Consider a test String **T = "abacaabaccabacabaabb"** and the pattern string **P = "abacab".** Then P is a substring of T. Namely, **P = T [10.. 15]** .

2. BRUTE-FORCE PATTERN MATCHING ALGORITHM (NAIVE ALGORITHM)

- The brute force algorithm design pattern is a powerful technique, typically enumerates all possible configurations of the inputs involved and picks the best of all these enumerated configurations.
- It simply test all the possible placements of pattern string P relative to the text string T.

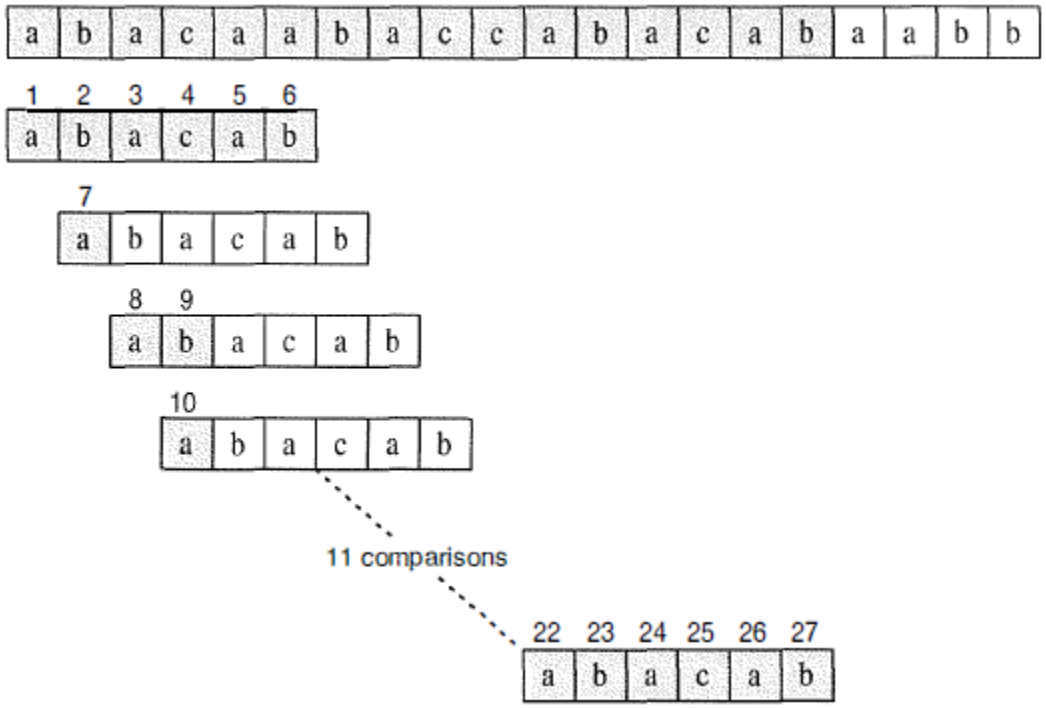
Algorithm ***Brute-Force Match*** (T, P)
Input: Strings T (text) with n characters and P (pattern) with m characters
Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

```
for i ← 0 to n-m do
{
```

```
    j ← 0
    while (j < m and T[i + J] = P[J] )  do j ← j+ 1
    if j = m then return i
    }
    return "There is no substring of T matching P."
```

- The Brute force algorithm consists of two nested loops, with the outer loop indexing through all possible starting indices of the pattern in the text and the inner loop indexing through each character of the pattern, comparing it to its potentially corresponding character in the text. Thus, the correctness of the brute-force pattern matching algorithm follows immediately.

Example: [Number of character comparisons = 27]



Analysis of Brute-Force Algorithm

- For each candidate index in T, it can perform up to m character comparisons to discover that P does not match T at the current index.
- The outer for-loop is executed at most n-m + 1 times, and the inner loop is executed at most m times.
- Thus, the running time of the brute-force method is O ((n - m + 1) m), which is **O(nm)** .
- Since this algorithm examines every character in T in order to locate a pattern P as a substring. Thus the running time is O (nm). Thus, in the worst case, when n and m are roughly equal, this algorithm has a quadratic running time.

3. BOYER-MOORE PATTERN MATCHING ALGORITHM

- The Boyer-Moore (BM) pattern matching algorithm can sometimes avoid comparisons between P and a sizable fraction of the characters in T.

- The BM algorithm assumes the alphabet is of fixed, finite size. It works the fastest when the alphabet is moderately sized and the pattern is relatively long.
- The main idea is to improve the running time of the brute-force algorithm by adding two potentially time-saving heuristics:
  1. **Looking-Glass Heuristic**: When testing a possible placement of P against T, begin the comparisons from the end of P and move backward to the front of P.
  2. **Character-Jump Heuristic** (also known as Bad Match table): During the testing of a possible placement of P against T, a mismatch of text character T [i] = c with the corresponding pattern character P[j] is handled as follows.
     a) If c is not contained anywhere in P, then shift P completely past T [i] (for it cannot match any character in P).
     b) Otherwise, shift P until an occurrence of character c in P gets aligned with T [i].
  - The looking-glass heuristic sets up the other heuristic to avoid comparisons between P and whole groups of characters in T.
  - The destination is reached faster by going backwards, if it encounters a mismatch during the consideration of P at a certain location in T, then it likely to avoid lots of needless comparisons by significantly shifting P relative to T using the character-jump heuristic (also called Bad-Match table).

The Character-Jump heuristic (Bad-Match table).

To implement this heuristic, define a function last (c) that takes a character c from the alphabet and specifies how far to shift the pattern P if a character equal to c is found in the text that does not match the pattern. In particular, the function last(c) is defined as follows,

Last (c) = m – i+1

where

m is length of the string and

'i' is the index of the character c.

Construction of Char-Jump Heuristic

Let i be the index of the character c of the pattern string P,

1. Last (c) = m – i+1
2. If i=m-1, (ie) the last character of the pattern P then Last (c) = m.
3. Set Last(c) = m, if c is not present in P, but present in T.

Example:

Consider a text String **T = "abacaabadcabacabaabb"** and the pattern string **P = "abacab".**

i	0	1	2	3	4	5
c	a	b	a	c	a	b

Last (a) = 6 – 0 – 1 = 5

Last (b) = 6 – 1 – 1 = 4

Last (a) = 6 – 2 – 1 = 3, Rewrite the entry for Last (a) with 3, from 5

Last (c) = 6 – 3 – 1 = 2

Last (a) = 6 – 4 – 1 = 1, Rewrite the entry for Last (a) with 1, from 3

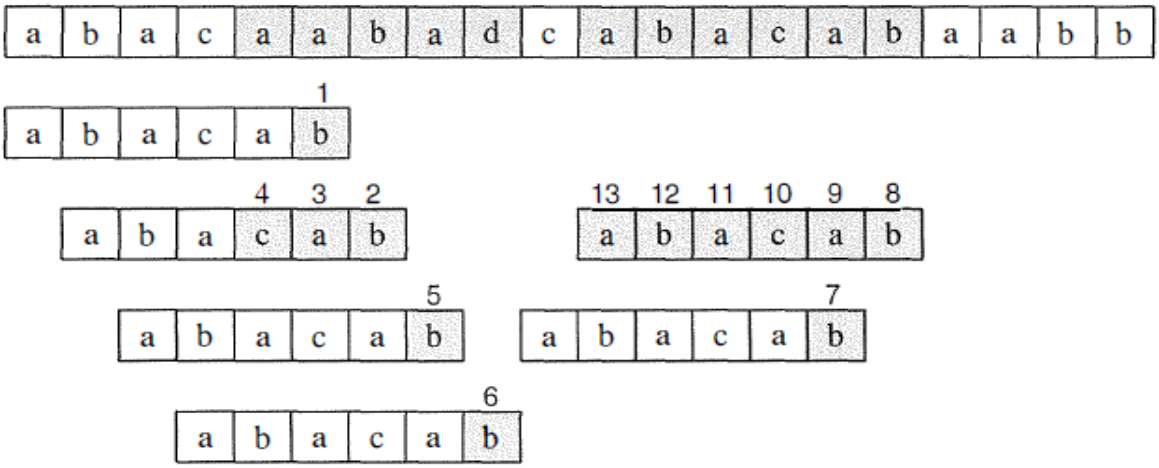
Last (b) = 6 – 5 – 1 = 0, since this is the last character, Rewrite the entry for Last (b) = 6

Last (*) = 6, * denotes other character not in P, but present in T.

Thus the **Char-Jump Heuristic table** is,

c	a	b	c	*
Last (c)	1	6	2	6

- If characters can be used as indices in arrays, then the Last () function can be easily implemented as a lookup table. The algorithm performs 13 character comparisons, which are indicated with numerical labels.
- The correctness of the BM pattern matching algorithm follows from the fact that each time the method makes a shift, it is guaranteed not to "skip" over any possible matches.
- For Last(c) is the location of the *last* occurrence of c in P.



ALGORITHM:

Algorithm Boyer-Moore Match (T, P)

Input: Strings T (text) with n characters and P (pattern) with m characters

Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

```
Compute Last() for P
i ←m – 1
j← m – 1
Repeat
        if P [j] = T [i] then
                if j = 0 then
                        return  i          //match found.
                else
                        i← i- 1
                        j ← j - 1
        else
                i← i + m – min(j, 1 + last(T [i]))
                j←m – 1
until i > n - I
return "There is no substring of T matching P"
```

Analysis of Boyer-Moore Pattern Matching Algorithm:

- The worst-case running time of the BM algorithm is O(nm + |∑|), where ∑ is the set of alphabets forming the pattern string P.
- The computation of the Last() function takes time O(m + |∑|) and the actual search for the pattern takes O(nm) time in the worst case, the same as the brute-force algorithm.

3. KMP (KNUTH – MORRIS – PRATT) PATTERN MATCHING ALGORITHM

- The KMP pattern matching algorithm incrementally processes the text string T comparing it to the pattern string P from left to right.
- Each time there is a match, it increments the current indices. On the other hand, if there is a mismatch then it consult the failure function f( ) to determine the new index in P where to continue checking P against T.
- Otherwise there was a mismatch and we are at the beginning of P then simply increment the index for T and keep the index variable for P at its beginning.
- Repeat this process until find a match of P in T or the index for T reaches n, the length of T indicating that the algorithm did not find the pattern P in T.

The KMP Failure-Function f()

- The main idea of the KMP algorithm is to preprocess the pattern string *P* so as to compute a *failure function f()* that indicates the proper shift of *P* so that, to the largest extent possible, it can reuse previously performed comparisons.
- Specifically, the failure function *f(j)* is defined as the length of the longest prefix of *P* that is a suffix of *P[1 .. j]* (note that we did *not* put *P[0.. j]* here).
- It also use the convention that *f(0)* = 0.
- The importance of this failure function is that it "encodes" repeated substrings inside the pattern itself.

Algorithm ***KMP Failure-Function (P)***
Input: String P (pattern) with *m* characters
Output: The failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of P[1 .. j]

```
i ← 0
j ← 0
f (0) ← 0
while i < m do
    if P[j] = P[i] then
    {we have matched j+ 1 characters}
            f(i) ←j + 1
            i ← i+ 1
            j ← j + 1
    else if j > 0 then
            {j indexes just after a prefix of P that must match}
```

```
            j ←f(j - 1 )
      else
            {we have no match here}
            f(i) ← 0
            i ←i + 1
```

Example: *Consider the pattern string* P = "abacab "*. The KMP failure function* f(j) *for the string P is as shown in the following table:*

pattern P[] = "**abacab**"

j = 0, i = 0.
f[0] is always 0, move to i = 1

j = 0, i = 1.
Since P[j] and P[i] do not match and j=0,
set f[i] = 0, move to i=2
Now, j = 0, **f[1] = 0**, i = 2

j = 0, i = 2.
Since P[j] and P[i] match, do j++,
store it in f[i] and do i++.
Now, j = 1, **f[2] = 1**, i = 3

j = 1, i = 3.
Since P[j] and P[i] do not match and j > 0,
set j = f[j-1] = f[0] = 0
j = 0, i = 3.
Since P[j] and P[i] do not match and j=0,
set f[i] = 0, move to i=4
Now, j = 0, **f[3] = 0**, i = 4

j = 0, i = 4.
Since P[j] and P[i] match, do j++,
store it in f[i] and do i++.
Now, j = 1, **f[4] = 1**, i = 5

j = 1, i = 5.
Since P[j] and P[i] match, do j++,
store it in f[i] and do i++.
Now, j = 2, **f[5] = 2**, i = 6

Stop here as we have constructed the whole f[].

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b

f(j)	0	0	1	0	1	2

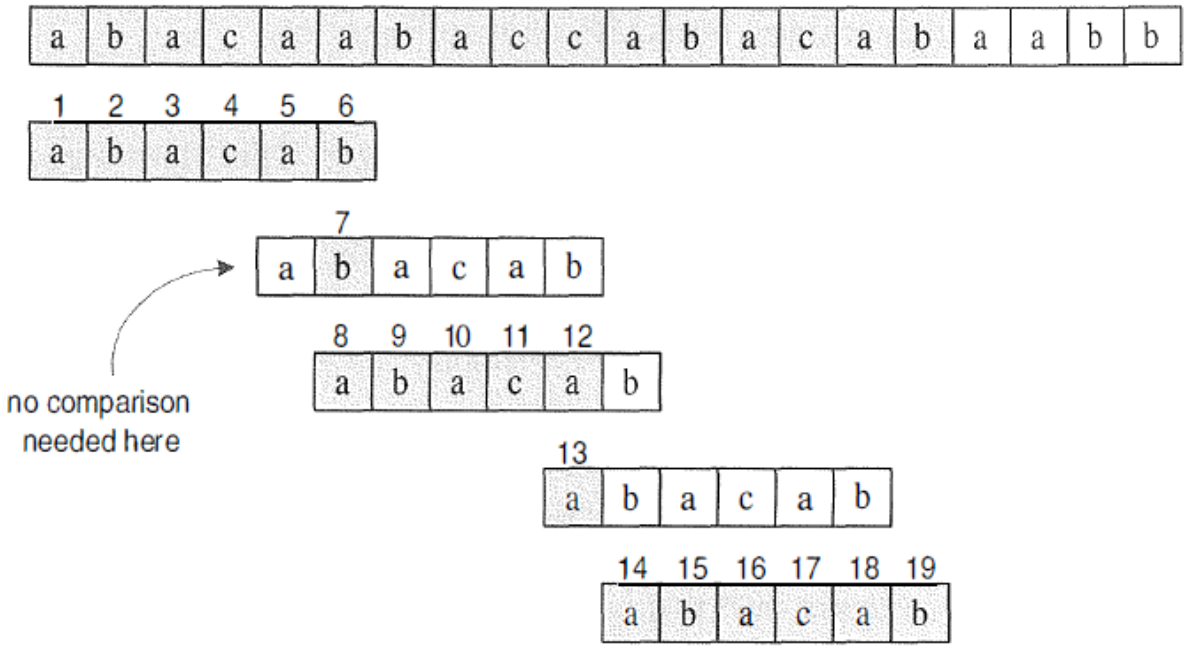
- This KMP Failure-Function f( ) algorithm compares the pattern to itself as in the KMP algorithm.
- Each iteration, if two characters are matched, then set f(i) ← j + 1 . Note that since i > j throughout the execution of the algorithm, f(j - 1) is always defined when we need to use it.

Algorithm KMP Pattern Matching

```
Algorithm KMP-Match (T, P)
Input: Strings T (text) with n characters and P (pattern) with m characters
Output: Starting index of the first substring of T matching P, or an indication that P is not a
substring of T
f ← KMP Failure-Function (P)                    {construct the failure function f for P}
i ←0
j ←0
while  i  <  n  do
      if P[j] = T [i] then
            if j = m - 1 then
              return (i - m + 1)                {a match !}
            i ← i + 1
            j ← j + 1
      else if j > 0 {no match , but we have advanced in P} then
            j ← f(j - 1)          {j indexes must after prefix of P that must match}
      else
            i ←  i + 1
return "There is no substring of T matching P."
```

Example:

Consider the pattern string P = "abacab " *and the text string T =* **“abacaabacabacabaabb”**



Analysis: Excluding the computation of the failure function, the running time of the KMP algorithm is clearly proportional to the number of iterations of the while-loop.

- For the sake of the analysis, let us define k = i- j. Intuitively, k is the total amount by which the pattern P has been shifted with respect to the text T. Note that throughout the execution of the algorithm, we have $k \leq n$.
- One of the following three cases occurs at each iteration of the loop.
  - If T[i] = P[j] , then i increases by 1, and k does not change, since j also increases by 1.
  - If T [i] ≠ P[j] and j > 0, then i does not change and k increases by at least 1, since in this case k changes from i- j to i- f(j - 1) , which is an addition of j- f(j - 1), which is positive because f(j - 1) < j.
  - If T [i] ≠ P[J] and j = 0, then i increases by 1 and k increases by 1, since j does not change.
- Thus, at each iteration of the loop, either i or k increases by at least 1 (possibly both); hence, the total number of iterations of the while-loop in the KMP pattern matching algorithm is at most 2n.
- **The KMP Failure-Function runs in O(m) time. Thus, *the Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length* n *and a pattern string of length m in* O(n + *m*) *time.***

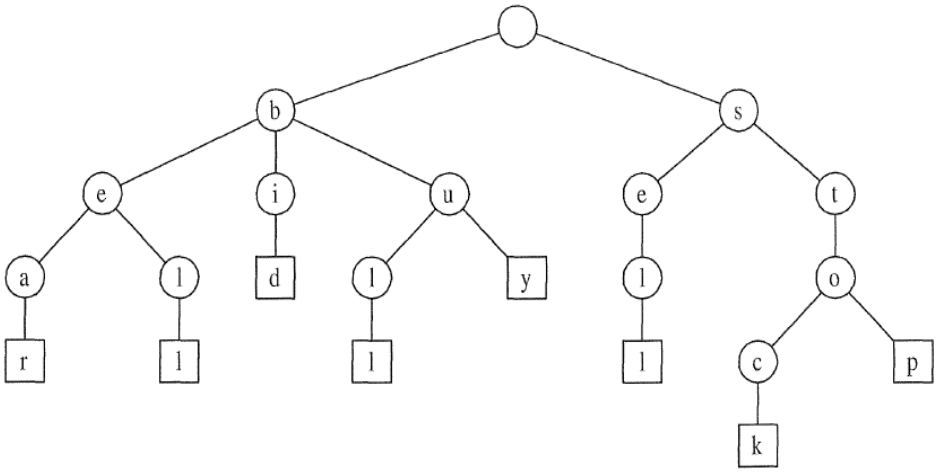
TRIES

- A trie (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching.
- The main application for tries is in information retrieval. Indeed, the name "trie" comes from the word "retrieval."

1. STANDARD TRIES

Let S be a set of s strings from alphabet L, such that no string in S is a prefix of another string,

- A *standard trie* for S is an ordered tree T with the following properties,
  1. Each node of T, except the root, is labeled with a character of L.
  2. The ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet L.
  3. T has n external nodes, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to an external node v of T yields the string of S associated with v.
- Thus, a trie *T* represents the strings of S with paths from the root to the external nodes of T.
- **Example:** Standard Trie for the strings S={bear, bell, bid, bull, buy, sell, stock, stop}.



The following are some important structural properties of a standard trie:

A standard trie storing a collection S of s strings of total length n from an alphabet of size d has the following properties:

1. ***Every internal node of T has at most d children***
2. ***T has s external nodes***
3. ***The height of T is equal to the length of the longest string in S***
4. ***The number of nodes of T is O(n)***

- In addition, a path from the root of *T* to an internal node v at depth i corresponds to an i^{th} character prefix X[0..i−1] of a string X of *S.*
- For each character c that can follow the prefix X[0..i−1] in a string of the set S, there is a child of v labeled with character c. In this way, a trie concisely stores the

common prefixes that exist among a set of strings.

- If there are only two characters in the alphabet, then the trie is essentially a binary tree, although some internal nodes may have only one child (that is, it may be an improper binary tree).
- In general, if there are d characters in the alphabet, then the trie will be a multi-way tree where each internal node has between 1 and d children.

Performance of Standard Trie

- The worst case for the number of nodes of a trie occurs when no two strings share a common non-empty prefix; that is, except for the root, all internal nodes have one child.
- A trie T for a set S of strings can be used to implement a dictionary whose keys are the strings of S.
- A search for a string X is performed in T by tracing down from the root the path indicated by the characters in X.
- If this path can be traced and terminates at an external node, then X is in the dictionary. If the path cannot be traced or the path can be traced but terminates at an internal node, then X is not in the dictionary.
- It is easy to see that the running time of the search for a string of size m is O (dm), where d is the size of the alphabet. Indeed, visiting at most m + 1 nodes of T and spending O (d) time at each node.
- For some alphabets, the time spent at a node to be improved to O (1) or O(log d) by using a dictionary of characters implemented in a hash table or lookup table.

Word Matching – an application of Tries

- A trie can be used to perform a special type of pattern matching, called *word matching,* where we want to determine whether a given pattern matches one of the words of the text exactly.
- Word matching differs from standard pattern matching since the pattern cannot match an arbitrary substring of the text, but only one of its words.
- Using a trie, word matching for a pattern of length m takes O(dm) time, where d is the size of the alphabet, independent of the size of the text.

Example: Consider a text:
"See a bear? Sell stock! See a bull? Buy stock! Bid stock! Bid stock! Hear the bell? Stop!"

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!		
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!			
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!				
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				

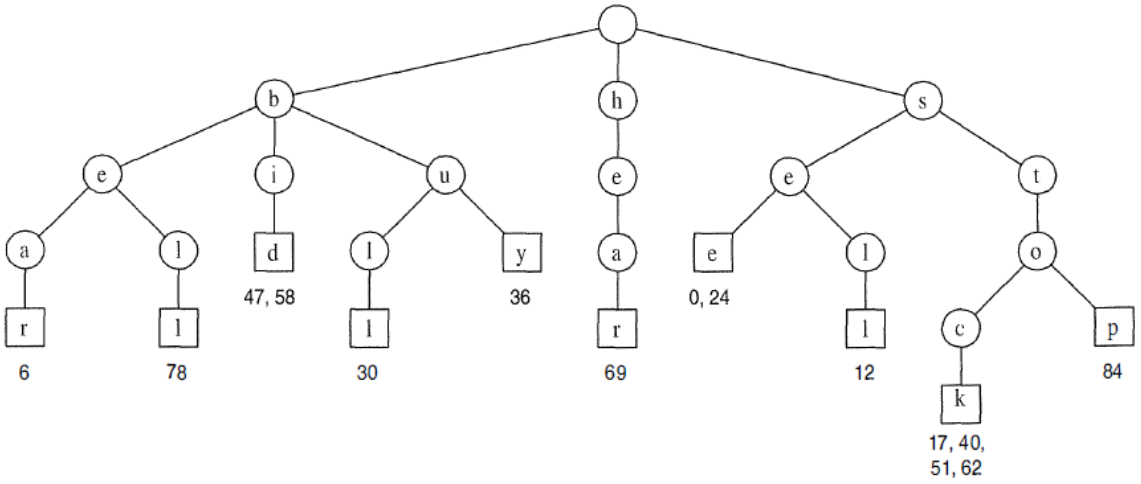


Fig: A standard Trie for the words in the text with articles and prepositions, which are known as stop words excluded.

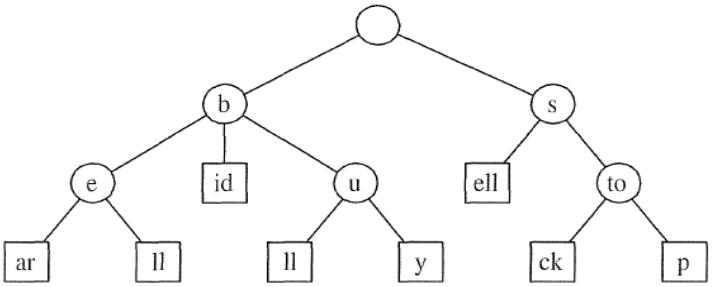
2. COMPRESSED TRIES

- A *compressed trie* is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges.
- Let *T* be a standard trie, then an internal node v of *T* is *redundant* if v has one child and is not the root.
- A Standard trie can be transformed into a compressed trie by replacing each redundant chain $(v_0 , v_1) \cdots (v_{k-1} , v_k)$ of $k \geq 2$ edges into a single edge (v_0 , v_k), relabeling v_k with the concatenation of the labels of nodes $v_1, \ldots , v_k$.

A compressed trie storing a collection S of s strings from an alphabet of size d has the following properties:

1. Every internal node of T has at least two children and at most d children
2. T has s external nodes
3. The number of nodes of T is O (s).

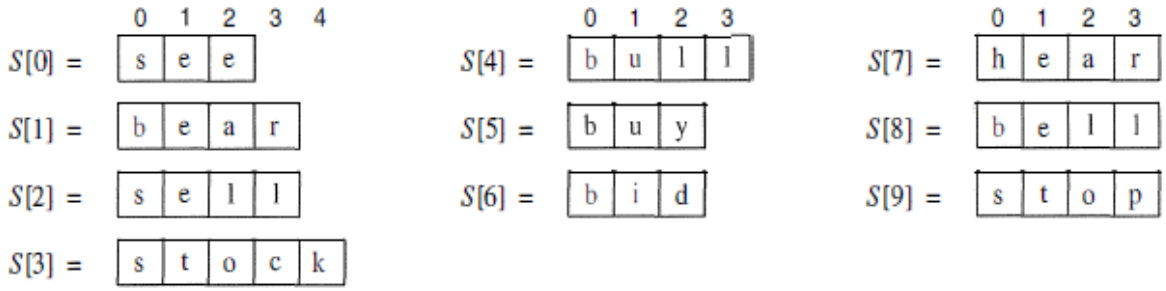
- Thus, nodes in a compressed trie are labeled with strings, which are substrings of strings in the collection, rather than with individual characters.
- The advantage of a compressed trie over a standard trie is that the number of nodes of the compressed trie is proportional to the number of strings and not to their total length.



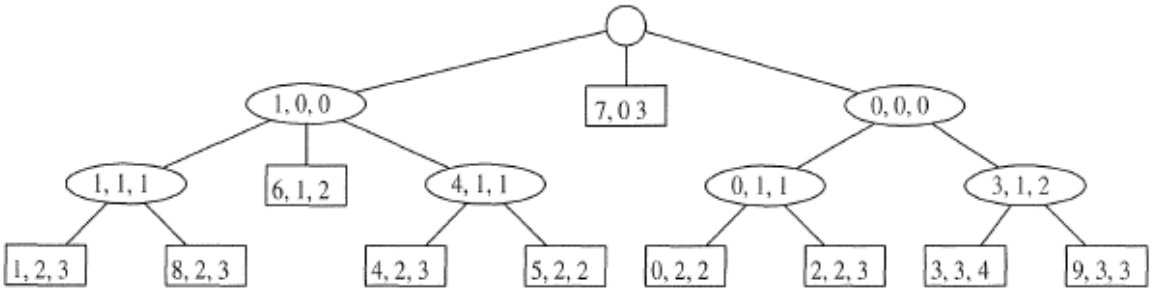
Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}

3. COMPACT REPRESENTATION OF COMPRESSED TRIES

- A compressed trie is truly advantageous only when it is used as an auxiliary index structure over a collection of strings already stored in a primary structure, and is not required to actually store all the characters of the strings in the collection.
- Let the collection S of strings is an array of strings S[0] , S[1], . . . , S[s - 1] . Instead of storing the label X of a node explicitly, it can be represented implicitly by a triplet of integers (i, j, k) ,, such that X = S[i] [j..k]; that is, X is the substring of S[i] consisting of the characters from the j^{th} to the k^{th} included.
- Compact representation of Trie



(a)



Performance

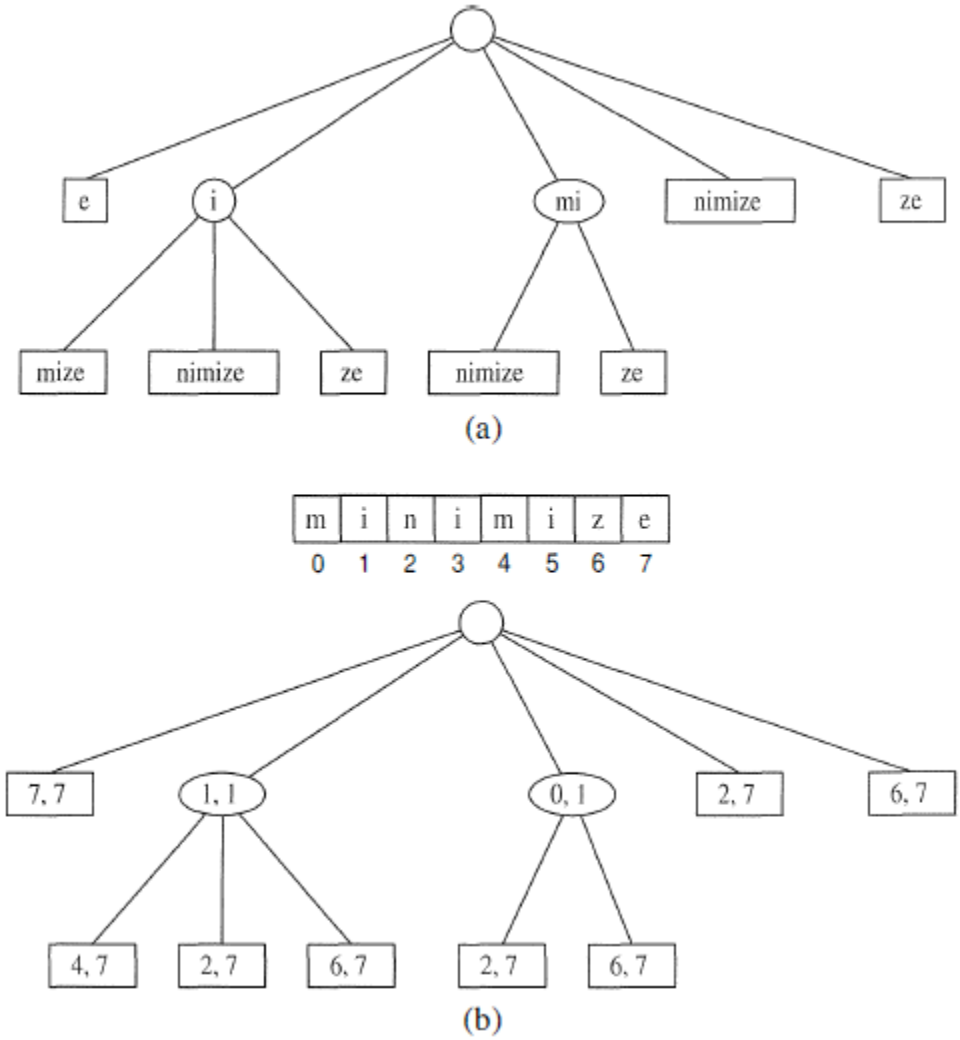
- This compression scheme reduces the total space for the trie itself from O(n) for the standard trie to O(s) for the compressed trie, where n is the total length of the strings in S and s is the number of strings in S.

4. SUFFIX TRIES

- One of the primary applications for tries is for the case when the strings in the collection S are all the suffixes of a string X. Such a trie is called the suffix trie (also known as a suffix tree or position tree) of string X.
- For a suffix trie, the compact representation can be further simplified by constructing the trie so that the label of each vertex is a pair (i, j) indicating the string X [i.. j] .
- To satisfy the rule that no suffix of X is a prefix of another suffix, a special character, denoted with $ is added, that is not in the original alphabet Σ at the end of X (and thus to every suffix).

- That is, if string X has length n, build a trie for the set of n strings X [i.. n-1]$, for i= 0, . . . , n - 1.

(a) Suffix Trie for the string “minimize” & (b) its compact representation



Performance

- The suffix trie T for a string X can be used to efficiently perform pattern matching queries on text X.
- Namely, we can determine whether a pattern P is a substring of X by trying to trace a path associated with P in T. P is a substring of X if and only if such a path can be traced.
- Using a suffix trie allows user to save space over a standard trie by using several space compression techniques, including those used for the compressed trie. The advantage of the compact representation of tries now becomes apparent for suffix tries. Since the total length of the suffixes of a string X of length n is

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

- Storing all the suffixes of X explicitly would take $O(n^2)$ space. Even so, the suffix trie represents these strings implicitly in O(n) space.

TEXT COMPRESSION

- Text compression is useful in any situation where we are communicating over a low-bandwidth channel, such as a slow modem line or wireless connection and we wish to minimize the time needed to transmit our text.
- Text compression is also useful for storing collections of large documents more efficiently, so as to allow for a fixed-capacity storage device to contain as many documents as possible.
- Standard encoding schemes, such as the ASCII and Unicode systems, use fixed- length binary strings to encode characters (with 7 bits in the ASCII system and 16 in the Unicode system).

1. HUFFMAN's ALGORITHM

- A Huffman code, on the other hand, uses a variable length encoding optimized for the string X.
- The optimization is based on the use of character *frequencies,* where we have, for each character c, a count f(c) of the number of times c appears in the string X.
- The Huffman code saves space over a fixed-length encoding by using short code-word strings to encode high-frequency characters and long code-word strings to encode low-frequency characters.
- To encode the string X, first convert each character in X from its fixed-length code word to its variable-length code word and then concatenate all these code words in order to produce the encoding Y for X.
- In order to avoid ambiguities, we insist that no code word in our encoding is a prefix of another code word in huffman encoding. Such a code is called a *prefix code* and it simplifies the decoding of Y in order to get back X.
- Huffman's algorithm for producing an optimal variable-length prefix code for X is based on the construction of a binary tree T that represents the code.
- Each node in T, except the root, represents a bit in a code word, with each left child representing a "0" and each right child representing a "1".
- Each external node v is associated with a specific character and the code word for that character is defined by the sequence of bits associated with the nodes in the path from the root of T to v.

The Huffman Coding Algorithm

- The Huffman coding algorithm begins with each of the d distinct characters of the string X to encode being the root node of a single-node binary tree.
- The algorithm proceeds in a series of rounds. In each round, the algorithm takes the two binary trees with the smallest frequencies and merges them into a single binary tree. It repeats this process until only one tree is left

Performance Analysis

- Each iteration of the while-loop in Huffman's algorithm can be implemented in O(log d) time using a priority queue represented with a heap.
- In addition, each iteration takes two nodes out of Q and adds one in, a process that will be repeated d - 1 times before exactly one node is left in Q.
- Thus, this algorithm runs in **O(n + d log d)** time.

Algorithm

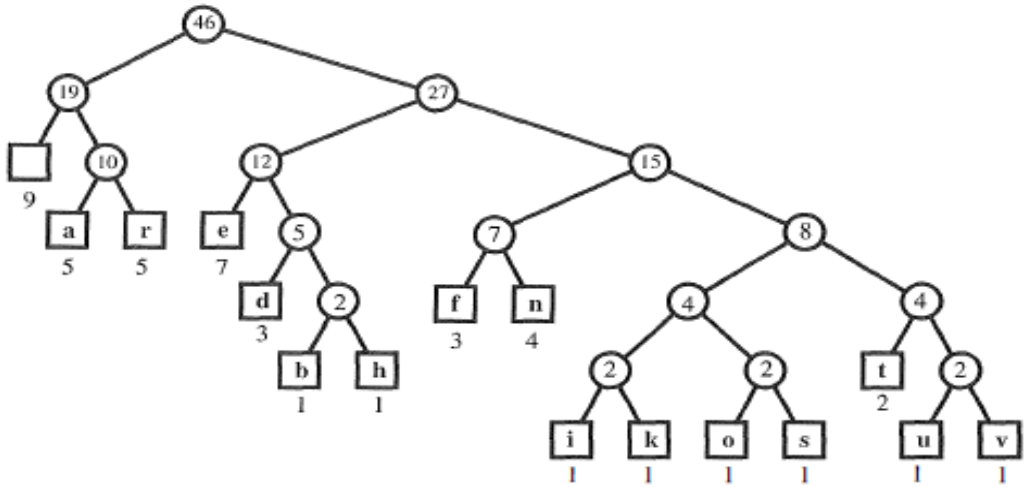
Algorithm ***Huffman (X)***
Input: String X of length n with d distinct characters
Output: Coding tree for X
Compute the frequency f(c) of each character c of X.
Initialize a priority queue Q.
for each character c in X do
    Create a single-node binary tree T storing c.
    Insert T into Q with key f(c) .
while Q.size() > 1 do
    f1 ← Q. min Key()
    T1 ← Q. removeM in()
    f2 ← Q.min Key()
    T2 ← Q. removeM in()
    Create a new binary tree T with left subtree T1 and right subtree T2.
    Insert T into Q with key fi + f2 .
return tree Q. removeMin ()

Example: Let X = " a fast runner need never be afraid of the dark " : (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for "a" is 010, and the code for "f" is 1 100.

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)



TEXT SIMILARITY TEST

1. String Subsequence

- Given a string X of size n, a *subsequence* of X is any string that is of the form,

  $$X [i_1] X [i_2] \ldots X [i_k], i_j < i_{j+1} \text{ for } j = 1, \ldots, k$$

  that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from X.
- For example, the string AAAG is a subsequence of the string CGATAATTGAGA. Note that the concept of *subsequence* of a string is different from the one of *substring* of a string.

2. LCS - LONGEST COMMON SUBSEQUENCE PROBLEM

- Given two character strings, X of size n and Y of size m, over some alphabet and to find a longest string S that is a subsequence of both X and Y.

3. Brute Force Method to find LCS

- One way to solve the longest common subsequence problem is to enumerate all subsequences of X and take the largest one that is also a subsequence of Y.
- Since each character of X is either in or not in a subsequence, there are potentially 2^n different subsequences of X, each of which requires 0(m) time to determine whether it is a subsequence of Y.
- Thus, the brute-force approach yields an exponential algorithm that runs in $0(2^n m)$ time, which is very inefficient.

Applying Dynamic Programming to the LCS problem

The LCS problem can be solved much faster than exponential time by using dynamic programming approach. One of the key components of the dynamic programming technique is the definition of simple subproblems that satisfy the subproblem optimization and subproblem overlap properties.

- In the LCS problem, we are given two character strings, X and Y, of length n and m respectively, and are asked to find a longest string S that is a subsequence of both X and Y.
- Since X and Y are character strings, we have a natural set of indices with which to define subproblems-indices into the strings X and Y.
- Let us define a subproblem, therefore, as that of computing the length of the longest common subsequence of X [0... i] and Y [0... j], denoted L[i, j].
- This definition allows us to rewrite L[i, j] in terms of optimal subproblem solutions with the following two cases,

  Case 1: X [i] = Y [j] ➔ Let c = X [i] = Y [j].
  - We claim that a longest common subsequence of X[0…i] and Y[0...j] ends with c.

- To prove this claim, assume that it is not true. There has to be some longest common subsequence X $[i_1]$X $[i_2]$ ... X $[i_k]$ = Y$[j_1]$Y $[j_2]$ . . . Y $[j_k]$.
- If X $[i_k]$ = c or Y $[j_k]$ = c, then we get the same sequence by setting i_k = i and j_k = j. alternately, if X $[j_k] \neq$ c, then we can get an even longer common subsequence by adding c to the end.
- Thus, a longest common subsequence of X [0…i] and Y [0... j] ends with c = X [i] = Y [j]. Therefore, we set **L [i, j] = L[i - 1, j - 1] + 1, if X [i] = Y [j] .**

Case 2: X [i] ≠ Y [j]

- In this case, we cannot have a common subsequence that includes both X[i] and Y[j].
- That is, a common subsequence can end with X [i] , Y [j] , or neither, but not both. Therefore, we set **L [i, j] = MAX {L[i - 1 , j] , L[i, j - 1] } if X [i] ≠Y [j]** .
- In order to make sense in the boundary cases when i = 0 or j = 0, we define

L [i, -1] = 0 for i = -1, 0, 1, . . . , n – 1

and

L [-1, j] = 0 for j = -1, 0, 1 , ... , m – 1

The LCS Algorithm

The definition of L [i, j] (in case 1 & 2) satisfies subproblem optimization, for with LCS and also without having longest common subsequences for the subproblems. Additionally, it uses subproblem overlap, because a subproblem solution L[i, j] can be used in several other problems (namely, the problems L[i + 1, j], L[i, j + 1], and L[i + 1, j+ 1]).

```
Algorithm LCS (X,Y)
Input: Strings X and Y with n and m elements, respectively
Output: For i = 0, . . . , n - 1, j = 0, . . . ,m - 1, the length L[i, j] of a longest common
subsequence of X [0... i] and Y [0 ... j]
for i ← -1 to n - 1 do
        L [i, -1] ← 0
for j ← 0 to m - 1 do
        L[-1, j] ← 0
for i ← 0 to n - 1 do
    for j ← 0 to m - 1 do
        if X [i] = Y [j] then
            L[i, j] ← L[i - 1, j - 1 ] + 1
        else
            L[i, j] ← max{L[i - 1, j] , L[i, j - 1] }
return array L
```

Analysis:

The running time of LCS Algorithm is easy to analyze, for it is dominated by two nested for-loops, with the outer one iterating n times and the inner one iterating m times. Since the if-statement and assignment inside the loop each requires O (1) primitive operations, this

algorithm runs in **O(nm)** time.

Construction of Longest Common Subsequence

A simple post-processing step can extract the longest common subsequence from the array L returned by the algorithm is follows:

1. **Start from L [n - 1, m - 1]** and work back through the table, reconstructing a longest common subsequence from back to front.
2. At any **position L[i, j],**
   3. if **X [i] = Y [j]** then

      Take X [i] as the next character of the subsequence

      Move next to L [i - 1, j - 1] .
   4. If **X [i] ≠ *Y* [j]** then

      Move to the larger of L[i, j - 1] and L[i - 1, j].
5. **Stop when** a boundary entry is reached (with **i = -1 or j = -1**).

Example: Let X = ACBDEA and Y = ABCDA find the longest Common Subsequence.

By the algorithm, {for construction of table please refer the class work note book}

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

LCS - "ACDA"

Thus the **Longest Common Subsequence is "ACDA" and length is 4.**

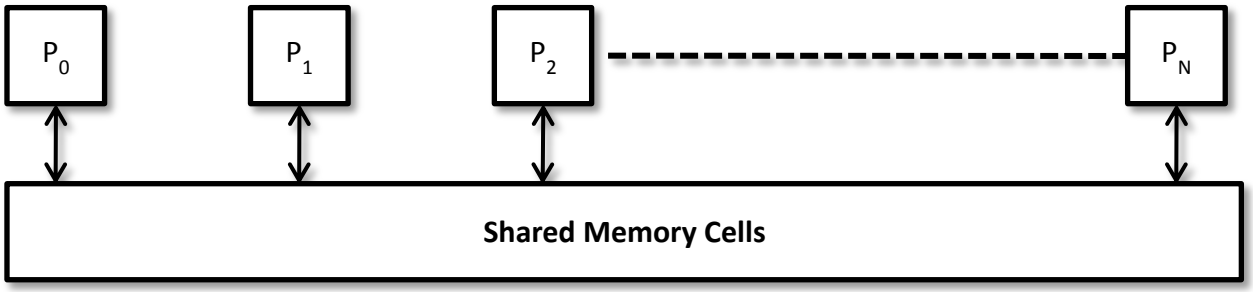
PART – II: PARALLELISM MODELS & ALGORITHMS

PARALLEL ALGORITHM

- A Parallel algorithm is an algorithm which can be executed a piece at a time on many processing devices and then combined together again at the end to get the correct result.
- Parallel algorithms become interesting from a computational complexity point of view when the number of processors is very large, larger than the input size for many of the actual cases for which a program is used, by which computations could be speed up significantly.

PRAM Architecture

- **Parallel Random Access Machine** is a straightforward and natural generalization of RAM. It is an idealized model of a **shared memory SIMD machine**.
- PRAM is one of the major classes of models for general purpose parallel computers. It serves as an effective tool for implementing parallel computers.
- PRAM is an abstract machine for designing the algorithms applicable to parallel computers.



- Its main features are:
  1. **Unbounded** collection of numbered RAM processors P_0, P_1, P_2,....
  2. **Unbounded** collection of **shared** memory cells *M[0], M[1], M[2],....*
  3. Each P_i has its own (unbounded) **local memory (registers)** and knows its index *i*.
  4. Each processor can access **any** shared memory cell (unless there is an access conflict, see further) in **unit time**.
  5. Input of a PRAM algorithm consists of *n* items stored in (usually the first) *n* shared memory cells.
  6. Output of a PRAM algorithm consists of *n'* items stored in *n'* shared memory cells.
  7. PRAM instructions execute in **3-phase cycles**.
     1. **Read** (if any) from a shared memory cell.
     2. **Local computation** (if any).
     3. **Write** (if any) to a shared memory cell.
  8. Processors execute these 3-phase PRAM instructions **synchronously**.
  9. **Special assumptions** have to be made about R-R and W-W shared memory **access conflicts**.
  10. The only way processors can exchange data is by writing into and reading from memory cells.
  11. P_0 has a special **activation register** specifying the maximum index of an active processor. Initially, only P_0 is active, it computes the number of required active

processors and loads this register, and then the other corresponding processors start executing their programs.

12. Computation proceeds until P_0 halts, at which time all other active processors are halted.
13. **Parallel time complexity** = the time elapsed for P_0's computation.
14. Space complexity = the number of shared memory cells accessed.

PRAM is an attractive and important model for designers of parallel algorithms. Why?

1. It is **natural**: the number of operations executed per one cycle on *p* processors is at most *p*.
2. It is **strong**: any processor can read or write **any** shared memory cell in unit time.
3. It is **simple**: it abstracts from any communication or synchronization overhead, which makes the complexity and correctness analysis of PRAM algorithms easier. Therefore,
4. It can be used as a **benchmark**: If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution on any parallel machine.
5. It is **useful**: it is an idealization of existing (and nowaday more and more abundant) shared memory parallel machines.

PRAM MODELS

To make the PRAM model realistic and useful, some mechanism has to be defined to resolve read and write access conflicts to the same shared memory cell. PRAM are classified based on their Read/Write abilities (realistic and useful)

<table>
<tr><td rowspan="4">Writes into the same memory cell / location</td><td colspan="4">Reads from same memory location / cell</td></tr>
<tr><td></td><td>EXCLUSIVE</td><td colspan="3">CONCURRENT</td></tr>
<tr><td>EXCLUSIVE</td><td>EREW
(Less powerful & more realistic)</td><td colspan="3">CREW
(Default)</td></tr>
<tr><td>CONCURRENT</td><td>ERCW
(Not useful)</td><td colspan="3">CRCW
(More powerful & less realistic)
has Further sub-models</td></tr>
<tr><td colspan="2"></td><td>P–CRCW</td><td>A–CRCW</td><td>C–CRCW</td></tr>
</table>

The operation of a synchronous PRAM can result in simultaneous access by multiple processors to the same location in shared memory. There are several variants of PRAM model, depending on whether such simultaneous access is permitted (concurrent access) or prohibited (exclusive access). As accesses can be *reads* or *writes*, we have the following four possibilities:

1. **Exclusive Read Exclusive Write (EREW)**:

- This PRAM variant does not allow any kind of simultaneous access to a single memory location.
- All correct programs for such a PRAM must insure that no two processors access a common memory location in the same time unit.

2. **Concurrent Read Exclusive Write (CREW)**:

- This PRAM variant allows concurrent reads but not concurrent writes to shared memory locations.

- All processors concurrently reading a common memory location obtain the same value.

3. **Exclusive Read Concurrent Write (ERCW)**:

- This PRAM variant allows concurrent writes but not concurrent reads to shared memory locations.
- This variant is generally not considered independently, but is subsumed within the next variant.

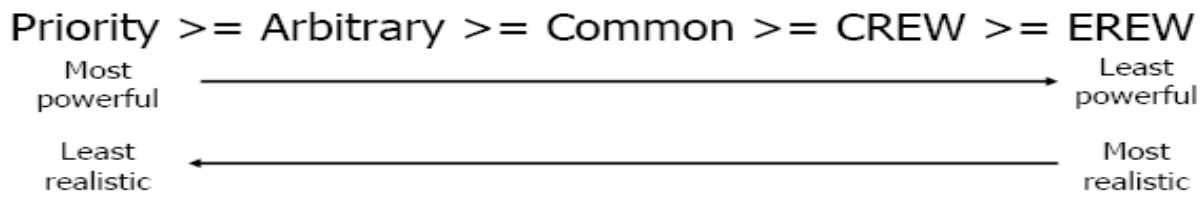
4. **Concurrent Read Concurrent Write (CRCW)**:

- This PRAM variant allows both concurrent reads and concurrent writes to shared memory locations.
- There are several sub-variants within this variant, depending on how concurrent writes are resolved.

CRCW – Sub models

1. **Common CRCW**: This model allows concurrent writes if and only if all the processors are attempting to write the same value (which becomes the value stored).
2. **Arbitrary CRCW**: In this model, a value arbitrarily chosen from the values written to the common memory location is stored.
3. **Priority CRCW**: In this model, the value written by the processor with the minimum processor id writing to the common memory location is stored.
4. **Combining CRCW**: In this model, the value stored is a combination (usually by an associative and commutative operator such as + or **max**) of the values written.
5. **REDUCTION CRCW:** All the values are reduced to one value using some reduction function such as SUM, MINIMUM, MAXIMUM, and so on.

The most powerful model is of course CRCW where everything is allowed but that's the most unrealistic in practice too. The weakest model is EREW where concurrency is limited, closer to real architectures although still infeasible in practice



SIMULATING MULTIPLE ACCESS ON AN EREW PRAM

- The EREW PRAM model is considered the most restrictive among the four subclasses of PRAM models. Only one processor can read from or write to a given memory location at any time.
- An algorithm designed for such model must not rely on having multiple processor access the same memory location simultaneously on order to improve its performance.
- An algorithm designed for an EREW can run on a CRCW PRAM. The algorithm simply will not use the concurrent access features in the CRCW PRAM. However, the contrary is not true, an algorithm designed for CRCW cannot run on an EREW PRAM.
- The simulation of concurrent read/write access on an EREW PRAM can be done using a broadcasting procedure.

Let a memory location x is needed by all processors at a given time in the EREW PRAM, then

1. P1 reads x and makes it known to P2
2. P1 and P2 make x known to P3 and P4 respectively in parallel.
3. P1, P2, P3, and P4 make x known to P5, P6, P7, and P8 respectively in parallel.
4. These eight processors will make x known to another eight processors and so on.

Parallel Algorithm Conventions

- The processors are named arbitrarily either 0, 1, . . . , $p - 1$ or 1, 2, . . . , p.
- The input to a particular problem would reside in the cells of the shared memory. In order to simplify the exposition of the algorithms, that a cell is wide enough (in bits or bytes) to accommodate a single instance of the input (eg. a key or a floating point number).
- If the input is of size n, the first n cells numbered 0, . . . , n − 1 store the input.
- Assume that the number of processors of the PRAM is n or a polynomial function of the size n of the input. Processor indices are 0, 1, . . . , n − 1.
- PRAM Inputs/Outputs are placed in the shared memory (designated address)
- Memory cell stores an arbitrarily large integer
- Each instruction takes unit time
- Instructions are synchronized across the processors
- PRAM Instruction Set: Accumulator architecture - memory cell M_0 accumulates results
- multiply/divide instructions take only constant operands
- Communication between processors: any pair of processor P_i and P_j can communicate in constant time. P_i writes the message in cell x at time t & P_j reads the message in cell x at time t+1.

Computational Power

- Model A is computationally stronger than model B, *if and only if* any algorithm written for B will run unchanged on A in the same parallel time and same basic properties.

Design Issues for PRAM

There are two technical issues for PRAM models,

1. How processors are activated?
2. How shared memory is accessed? – by concurrent / exclusive.

(i) Processor Activation in PRAM Model

- The processor P_0 places the number of processors (p) in the designated shared-memory cell.
  - each active P_i, where i < p, starts executing
  - O(1) time to activate.
  - all processors halt when P_0 halts.
- Active processors explicitly activate additional processors via FORK instructions
  - tree-like activation: *i* processor will activate a processor *2i* and a processor *2i+1*
  - O(log p) time to activate

SIMPLE PRAM ALGORITHMS

(1a) Parallel MAX – finding maximum of 'n' numbers

The Tournament method is commonly used technique for PRAM computation. In the tournament method, elements are paired off and compared in rounds. In succeeding rounds, the winners from the preceding round are paired off and compared. All comparisons in one round can be performed at the sa

Input: Keys x[0], x[1], …, x[n-1], initially in memory cells M[0], M[1],…,M[n-1].
Output: The largest key will be let in M[0].
Remarks: Each processor carries out the algorithm Using its own index number(pid) for a unique Offset into M.
The algorithm initializes cells M[n],…, M[2*n-1] with -∞, because some of these cells will enter the process.
Example: Please refer class work note book.

```
EREW-Maximum (n)
 int incr;
 key BIG, TEMP0, TEMP1;
 incr=1;
while(incr<n)
   READ M[pid] into TEMP0;
   READ M[pid + incr] into TEMP1;
   BIG = max (TEMP0, TEMP1);
   WRITE BIG into M[pid]
    incr = incr * 2;
```

(1b) PARALLEL ALGORITHM FOR FINDING MAXIMUM IN CONSTANT TIME. [CRCW]

An algorithm for finding the maximum of n numbers using CRCW / Common CRCW PRAM model, taking less time than the EREW PRAM model.

This algorithm uses n^2 processors to simplify indexing, although only n(n-1)/2 processors do any work. The strategy is to compare all pairs of keys in parallel, then communicate the results through the shared memory. It uses an array called LOSER which can occupy the memory cells M[n], Mn+1], …, M[2*n-1]. Initially all entries of LOSER array are zero. If xi loses a comparison, then LOSER[i] will be assign

Input: Keys x_0, x_1, …, x_{n-1}, initially in memory cells M[0], M[1],…,M[n-1].
Output: The largest key will be let in M[0].
Remarks:
The processors will be numbered
$P_{i,j}$ for $0 \le i,j \le n-1$.
Each processor computes its i and j from its Index by
i=ceiling(pid/n)
&
j =pid – ni,
if i≥j, then the processor does no work.
Example: Please refer classwork notebook.

CRCW-Maximum (n)
Compute i and j from pid. If i ≥ j then return.
$P_{i,j}$ reads x_i from M[i] & reads x_j from M[j]
$P_{i,j}$ compares x_i and x_j
(Let k be the index of the smallest key)
$P_{i,j}$ writes 1 into LOSER[k]
$P_{i,i+1}$ reads LOSER[i] and $P_{0,n-1}$ reads LOSER [n-1]
The processor that reads a 0 writes x_i in M[0] & $P_{0,n-1}$ would writes x_{n-1}.
End

(1c) PARALLEL ALGORITHM FOR SUM OF N NUMBERS

Input: An array A = A(1) . . .A(n) of n numbers, stored in M[0], m[1]…, m[n-1]
The problem is to compute A(1) + . . . + A(n).
The summation algorithm works in rounds.

Each round: add, in parallel, pairs of elements: add each odd-numbered element and its successive even-numbered element.
If n = 8, outcome of 1st round is: A(1) + A(2), A(3) + A(4), A(5) + A(6), A(7) + A(8)
Outcome of 2nd round: A(1) + A(2) + A(3) + A(4), A(5) + A(6) + A(7) + A(8) and the outcome of 3rd (and last) round: A(1) + A(2) + A(3) + A(4) + A(5) + A(6) + A(7) + A(8)
Let pid() returns the index of the processor issuing the call.

PARALLEL SUM(n) ➔ EREW	**CRCW – PARALLEL SUM (n)**
Begin **i=1** **j=pid();** **while(j mod 2^i ==0)** **a=m[j]** **b=m[j+2^{i-1}]** **m[j] = a+b** **i=i+1** end	Begin **i=1** **j=pid();** **while(j < a/2^i)** **a=m[2j]** **b=m[2j+1]** **m[j] = a+b** **i=i+1** end

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]	
x0	x1	x2	x3	x4	x5	x6	x7	time=0
x0+x1		x2+x3		x4+x5		x6+x7		time=1
x0+...+x3				x4+...+x7				time=2
x0+...+x7								time=3

EREW – Parallel Sum

The EREW PRAM algorithm consists of log *n* steps. In step *i*, if *j* can be exactly divisible by 2^i, processor *j* reads shared-memory cells *j* and $j + 2^{i-1}$ combines (sums) these values and stores the result into memory cell *j*. After log *n* steps the sum resides in cell 0. Algorithm Parallel Sum has T = O(lg n), P = n and W = O(n lg n), W2 = O(n).

Processing node used:

P0, p2, p4, p6 t=1
P0, p4 t=2
P0 t=3

Example: Please refer the class work notebook.

CRCW – Parallel Sum

In step *i*, processor $j < n / 2^i$ reads shared-memory cells 2*j* and 2*j* +1 combines (sums) these values and stores the result into memory cell *j*. After log *n* steps the sum resides in cell 0. Algorithm Parallel Sum has T = O(log n), P = n and W = O(n log n), W2 = O(n).

M[0] M[1] M[2] M[3] M[4] M[5] M[6] M[7]

x0	x1	x2	x3	x4	x5	x6	x7	t=0
x0+x1	x2+x3	x4+x5	x6+x7					t=1
x0+...+x3	x4+...+x7							t=2
x0+...+x7								t=3

(1d) PARALLEL BOOLEAN OR ON N BITS

- Each processor performs an OR operation on a pair of bits at each round, and the problem is solved in O(log n) time. But in all the Concurrent Write models this algorithm in constant time.
- Find the OR of n bits $x_0, x_2,\ldots x_{n-1}$ input as 0's and 1's in M[0], M[1]….M[n-1].

Input: Bits x0,x1,…,xn-1 in M[0], M[1],… M[n-1]

Output: x0 OR x1 OR…xn-1 in M[0]

Common-Write OR (M,n)
Each Pi reads xi from M[i]
If Xi is 1, then Pi writes 1 into M[0]
End

- Since all the processors that write into M[0] write the same value '1', this is a legal program for Common Write model, and therefore also for the all CRCW models.
- The Boolean OR of n bits can be computed in one step on these models with n processors.

Example: Please refer class work notebook.

Boolean AND on n Bits

- Each processor performs an AND operation on a pair of bits at each round, and the problem is solved in O(log n) time. But in all the Concurrent Write models this algorithm in constant time.
- Find the AND of n bits $x_0, x_2,\ldots x_{n-1}$ input
- as 0's and 1's in M[0], M[1]….M[n-1].

Input: Bits x0,x1,…,xn-1 in M[0], M[1],… M[n-1]

Output: x0 AND x1 AND…AND xn-1 in M[0]

Common-Write OR (M,n)
Set M[0] to 1.
Each Pi reads xi from M[i]
If Xi is 0, then Pi writes 0 into M[0]
End

- Since all the processors that write into M[0] write the same value '1', this is a legal program for Common Write model, and therefore also for the all CRCW models.
- The Boolean OR of n bits can be computed in one step on these models with n processors.
- The result stored in cell 0 is 1 (TRUE) unless a processor writes a 0 in cell 0; then one of the X_i is 0 (FALSE) and the result X should be FALSE, as it is.

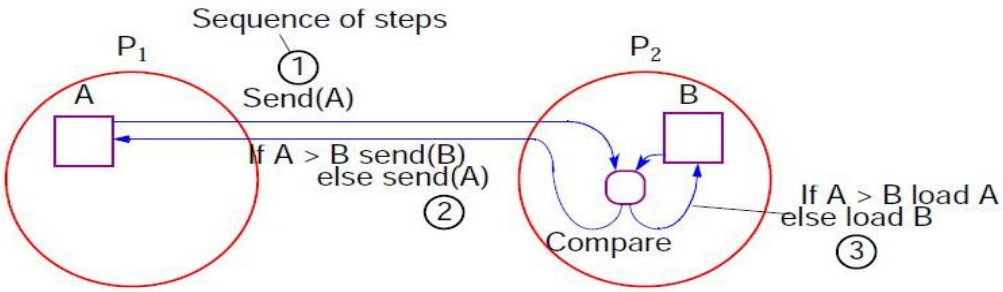
Example: Please refer class work notebook.

(1e) PARALLEL SORTING ALGORITHM

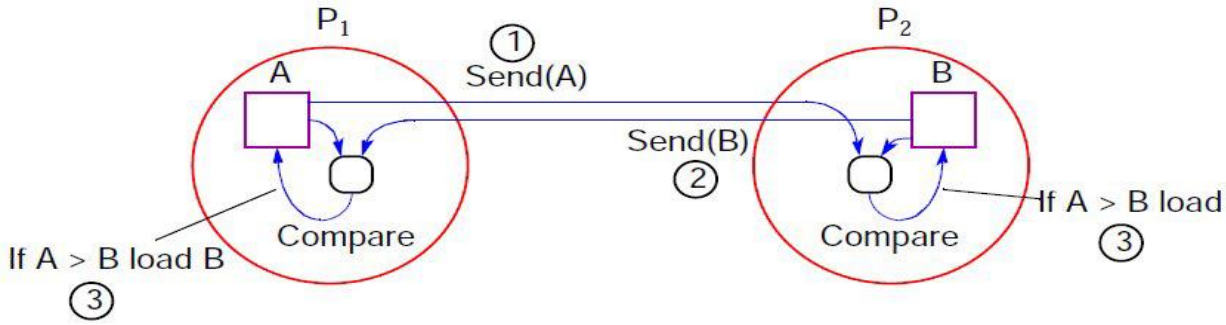
Goal: sorting a sequence of values in increasing order using n processors

Sequential sorting requires the comparison of values and swapping in the positions they occupy in the sequence. And, if it is in parallel? And, if the memory is distributed? There are two cases,

Case (1): The processor P1 sends A to P2. This compares B with A and sends to P1 the Min(A,B)



Case 2: The processor *P*1 sends *A* to *P*2; *P*2 sends *B* to *P*1; *P*1 does *A* = *min*(*A, B*) and *P*2 does *B* = *max*(*A, B*).



Bubble Sort: A Sequential bubble sort algorithm compares two consecutive values at a time and swaps them if they are out of order. Greater values are being moved towards the end of the list. Number of comparisons and swaps: (n(n-1))/2 which corresponds to a time complexity O(n^2).

- The bubble sort can be implemented parallel using odd-even transposition with two phases,
  Phase-even: Even numbered processors exchange values with right neighbors.
  Phase-odd: Odd numbered processors exchange values with right neighbors.

Algorithm

```
procedure ODD-EVEN TRANSPOSITION (S)
  for j = 1 to ⌈n/2⌉ do
    (1) for i = 1, 3, ..., 2⌊n/2⌋ − 1 do in parallel
          if x_i > x_{i+1}
          then x_i ↔ x_{i+1}
          end if
        end for
    (2) for i = 2, 4, ..., 2⌊(n − 1)/2⌋ do in parallel
          if x_i > x_{i+1}
          then x_i ↔ x_{i+1}
          end if
        end for
  end for. □
```

Example:

Odd-Even Transposition Sort - example

Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4 ↔	2	7 ↔	8	5 ↔	1	3 ↔	6
1	2	4 ↔	7	8 ↔	1	5 ↔	3	6
2	2 ↔	4	7 ↔	1	8 ↔	3	5 ↔	6
3	2	4 ↔	1	7 ↔	3	8 ↔	5	6
4	2 ↔	1	4 ↔	3	7 ↔	5	8 ↔	6
5	1	2 ↔	3	4 ↔	5	7 ↔	6	8
6	1 ↔	2	3 ↔	4	5 ↔	6	7 ↔	8
7	1	2 ↔	3	4 ↔	5	6 ↔	7	8

Time ↓

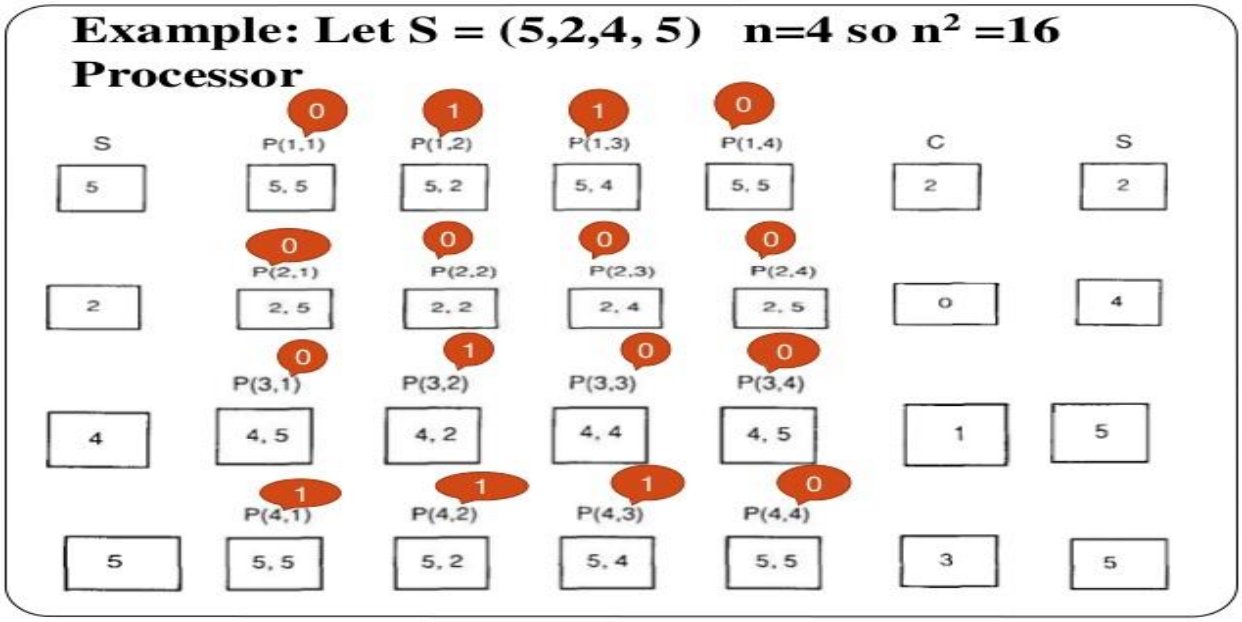
Parallel time complexity: $T_{par} = O(n)$ (for P=n)

FAST PARALLEL SORTING – CRCW SORT

CRCW Sort (S)

```
Begin
Step 1: For i=1 to n do in parallel
        For j=1 to n do in parallel
          if xi > xj OR (xi = xj AND i>j) then
            P i,j writes 1 in Ci
          Else
            P i,j writes 0 in Ci
Step 2: For i=1 to n do in parallel
          Pi,1 stores xi in position 1+Ci into M.
End
```

Example:



From the step – 2

P1,1 stores X1 into position 1 + 2 = 3: Writes 5 into M[3]
P2,1 stores X2 into position 1 + 0 = 1: Writes 2 into M[1]
P3,1 stores X3 into position 1 + 1 = 2: Writes 4 into M[2]
P4,1 stores X4 into position 1 + 3 = 4: Writes 5 into M[4]

Final Memory cells M[1], M[2], M[3] and M[4] after sorting.

2	**4**	**5**	**5**

PART – III: PROBABILISTIC ALGORITHMS

PROBABILISTIC ALGORITHMS

- A probabilistic algorithm is an algorithm where the result and/or the way the result is obtained depend on chance. These algorithms are also sometimes called randomized algorithms.
- A probabilistic algorithm A (⋅ , ⋅) is an algorithm that takes two inputs x and r, where x is an instance of some problem to be solved and r is the output of a random source.
- A random source is an idealized device that outputs a sequence of bits that are uniformly and independently distributed. For example the random source could be a device that tosses coins, observes the outcome, and outputs it.

Characteristics of Probabilistic Algorithms

- At least once during the algorithm, a random number is used to make a decision instead of spending time to work out which alternative is best.
- The worst-case running time of a randomized algorithm is almost always the same as the worst-case running time of the non-randomized algorithm.
- A probabilistic algorithm A is good if it is efficient and if, say, for every x,

$$\mathbf{Pr_r[\ A(x,r) = \text{correct answer for } x) \geq \frac{3}{4}}$$

- A probabilistic algorithm runs quickly but occasionally makes an error. The probability of error can, however, be make negligibly small.
- Any purported solution can be verified efficiently for correctness. This algorithm may give probabilistic answers which are not necessarily exact.
- The same algorithm may behave differently when it is applied twice to the same instance. Its execution time, and even the result obtained, may vary considerable from one use to the next.
- If the algorithm gets stuck (e.g., core dump), simply restart it on the same instance for a fresh chance of success.
- If there is more than one correct answer, several different ones may be obtained by running the probabilistic algorithm more than once. An expected running time bound is somewhat stronger than an average-case bound, but is weaker than the corresponding worst-case bound.

Types of Probabilistic algorithms - There is a variety of behaviours associated with probabilistic algorithms:

1. Monte Carlo algorithms
2. Las Vegas algorithms
3. Sherwood algorithms
4. Numerical Approximation algorithms

1. Monte Carlo algorithms

- Algorithms which always return a result, but the result may not always be correct. Attempt to minimize the probability of an incorrect result, and using the random element, multiple runs of the algorithm will reduce the probability of incorrect results.
- These are called Monte Carlo algorithms. **Example:** Primality testing

2. Las Vegas algorithms

- Algorithms that never return an incorrect result, but may not produce results at all on some runs. To minimize the probability of no result, and because of the random element, multiple runs will reduce the probability of no result. These are called Las Vegas algorithms.
- Example: factoring

3. Sherwood algorithms

- Algorithms which always return a result and the correct result, but where a random element increases the efficiency, by avoiding or reducing the probability of worst-case behaviour.
- Useful for algorithms which have a poor worst-case behaviour but a good average-case behaviour. These are called Sherwood algorithms

Example: Quicksort

4. Numerical Approximation algorithms

- Here the random element allows us to get approximate numerical results, often much faster than direct methods, and multiple runs will provide increasing approximation.

COMPLEXITY CLASSES FOR PROBABILISTIC ALGORITHMS

1. Randomized Polynomial time (RP):

- A language is in RP if there is a polynomial-time (Monte Carlo) algorithm that never accepts words that are not in the language, and has probability at least 1/2 of accepting words in the language

2. Zero-error Probabilistic Polynomial (ZPP)

- Like RP, except the algorithm is Sherwood, so it always accepts if the word is in the language

3. Bounded-error Probabilistic Polynomial (BPP)

- Like RP, except that the algorithm has probability at least 3/4 of accepting if the word is in the language, and probability at most 1/4 of accepting if the word is not in the language.

PROBABILISTIC ALGORITHMS VERSUS DETERMINISTIC ALGORITHMS

1. In some applications the use of probabilistic algorithms is natural, e.g. simulating the behaviour of some existing or planned system over time. In this case the result by nature is stochastic.
2. In some cases the problem to be solved is deterministic but can be be transformed into a stochastic one and solved by applying a probabilistic algorithm (eg. numerical integration, optimization).
3. For these numerical applications the result obtained is always approximate, but its expected precision improves as the time available to use the algorithm increases.
4. There are also a number of discrete problems for which only an exact result is acceptable (eg. sorting and searching) and where the introduction of randomness influences only on the ease and efficiency in finding the solution.
5. For some problems where trivial exhaustive search is not feasible probabilistic algorithms can be applied giving a result that is correct with a probability less than one (eg. primality testing, string equality testing).
6. The probability of failure can be made arbitrary small by repeated applications of the algorithm.
7. One incentive for using probabilistic algorithms is that their application does not normally require sophisticated mathematical knowledge. Further, the programming is often rather trivial which means that an acceptable approximation can be obtained quickly.
8. One can say that the use of probabilistic algorithms sometimes allow that theoretical knowledge and analytical work is compensated for by making extensive simple machine computations.
9. In some other cases the probabilistic algorithms are the simplest and even the most efficient available and for some problems no other feasible algorithm is known to exist (eg. primality testing).

PSEUDO RANDOM GENERATORS

- A pseudo-random number generator (PRNG) is a program written for, and used in, probability and statistics applications when large quantities of random digits are needed. Most of these programs produce endless strings of single-digit numbers, usually in base 10, known as the decimal system. When large samples of pseudo-random numbers are taken, each of the 10 digits in the set {0,1,2,3,4,5,6,7,8,9} occurs with equal frequency, even though they are not evenly distributed in the sequence.
- Many algorithms have been developed in an attempt to produce truly random sequences of numbers, endless strings of digits in which it is theoretically impossible to predict the next digit in the sequence based on the digits up to a given point.
- But the very existence of the algorithm, no matter how sophisticated, means that the next digit can be predicted! This has given rise to the term pseudo-random for such machine-generated strings of digits.
- They are equivalent to random-number sequences for most applications, but they are not truly random according to the rigorous definition.
- Most of the time pseudorandom generators are deterministic procedures able to generate long sequence of values that appears to have the properties of a random sequence.
- To start a sequence, an initial value is needed called SEED. The same seed always gives rise to the same sequence. To obtain different sequence, may choose a seed that depends on the date or time of the system.

The linear congruential generator: $xi+1 = Axi \% M$, where $x0$ is the seed and $1 \le x0 < M$. If M is prime, xi is never 0. After M-1 numbers, the sequence repeat (period of M-1). Some choices of A gets shorter period than M-1.

1. If M is chosen to be a large, 31-bit prime, the period should be significantly large for most applications. $M = 231 - 1 = 2{,}147{,}483{,}647$ and $A = 48{,}271$.
2. Same sequence occurs all the time for easy debugging, and input seed (e.g., use system clock) for real runs.
3. Usually a random real number in the open interval (0,1), which can be done by dividing by M.
4. Multiplication overflow prevention: let $Q = M / A = 44{,}488$ and $R = M \% A = 3{,}399$,
   $xi+1 = Axi \% M = A(xi \% Q) - R(xi / Q) + M(xi)$, where $(xi) = xi / Q - Axi / M = 1$ iff the remaining terms evaluate to less than zero, 0 otherwise.

$$
\begin{aligned}
xi+1 &= Axi \% M = Axi - M(Axi / M) \\
&= Axi - M(xi / Q) + M(xi / Q) - M(Axi / M) \\
&= Axi - M(xi / Q) + M(xi / Q - Axi / M) \\
&= A(Q(xi / Q) + xi \% Q) - M(xi / Q) + M(xi / Q - Axi / M) \\
&= (AQ - M)(xi / Q) + A(xi \% Q) + M(xi / Q - Axi / M) \\
&= -R(xi / Q) + A(xi \% Q) + M(xi / Q - Axi / M) \\
&= A(xi \% Q) - R(xi / Q) + M(xi)
\end{aligned}
$$

NUMERICAL PROBABILISTIC ALGORITHMS

- For certain real-life problems, computation of an exact solution is not possible even in principle, e.g., uncertainties in the experimental data, digital computers handle only binary values, etc.
- For other problems, a precise answer exists but it would take too long to figure it out exactly. Numerical algorithms yield a confidence interval, and the expected precision improves as the time available to the algorithm increase.
- The error is usually inversely proportional to the square root of the amount of work performed.

(i) **Buffon's Needle:**

- Throw a needle at random on a floor made of planks of constant width, if the needle is exactly half as long as the planks in the floor and if the width of the cracks between the planks are zero, the probability that the needle will fall across a crack is $1/\prod$.
- The probability that a randomly thrown needle will fall across a crack is $2\lambda /w\prod$, where λ is needle length and w is plank width.
- The result estimate will be between $\prod - \varepsilon$ and $\prod + \varepsilon$ with probability at least p (desired reliability).

(ii) **Numerical integration (Monte Carlo integration)**:

- Deterministic integration algorithms are easy to be fooled, and very expensive when evaluating a multiple integral.
- The hybrid techniques that partly systematic and partly probabilistic is called quasi Monte Carlo integration.

(iii) **Probabilistic Counting:**

- Counting twice as far to up to $2^{n+1} - 2$ by initialize to 0, each time tick is called, flip a fair coin. If it comes up head, add 1 to the register, otherwise, do nothing.
- When count is called, return twice the value stored in the register.
- Counting exponentially farther from 0 to $2(2^2-1) - 1$.
- Keep in the register an estimate of the logarithm of the actual number of ticks and count(*c*) returns 2^c-1.
- Keep the relative error in control instead of absolute.

MONTE CARLO ALGORITHMS

- Monte Carlo algorithms give exact answer with high probability whatever the instance considered, although sometimes they provide a wrong answer.

- Generally we cannot tell if the answer is correct, but we can reduce the error probability arbitrarily by allowing the algorithm more time (amplifying the stochastic).
- A Monte Carlo algorithm is ***p-correct*** if the probability that it outputs a correct solution is at least p for every input. Advantage = p - 0.5.
- A Monte Carlo algorithm is ***consistent*** if its all executions to the same input result in the same solution (irrespective of whether the solution is correct).
- **Confidence Amplification:** If a Monte Carlo algorithm is p-correct (p > 0.5) and consistent, its advantage can be amplified arbitrarily close to 1 by repeating execution of the algorithm.
- **Fast Amplification:** A Monte Carlo algorithm is ***y_0-biased*** if there is a subset X of inputs such that
  1. It always outputs a correct solution to every input not in X, and
  2. It outputs y_0 to every input in X, where y_0 may or may not be correct.
- If an Monte Carlo algorithm is y_0-biased, p-correct (even if p < 0.5) and consistent, then k repetitions yields a y_0-biased, $(1 - (1-p)^k)$-correct and consistent algorithm.

(i) Monte Carlo Algorithm for Verifying Matrix Multiplication

- Consider three n X n matrices A, B and C. To verify that C = A.B, the obvious approach is to multiply A and B and then compare the result with C.
- The traditional matrix multiplication algorithm takes $O(n^3)$ time. Strassen's algorithm is faster and even faster algorithm exists for very large n but takes $\Omega(n^{2.37})$. But it can be solved $O(n^2)$ time with error probability at most ε where $\varepsilon > 0$.

Verifying Matrix Multiplication

- Let $D = AB - C$, consider any subset $S \subseteq \{1,2, .., n\}$, and $\sum_S(D)$ denote the vector of length n obtained by adding point-wise the rows of D indexed by the elements of S.
- $\sum_S(D)$ is always 0 if AB equal C, otherwise, assume i be an integer such that the i^{th} row of D contains at least one nonzero element.
- The probability that $\sum_S(D) \neq 0$ is at least one-half.
- Let X be a binary vector of length of n such that $X_j = 1$ if $j \in S$ and $X_j = 0$ otherwise.
- Then $\sum_S(D) = XD$, and we want to verify if $XAB = XC$, where $(XA)B$ need $O(n^2)$. Getting the answer false just once allows you conclude that $AB \neq C$.
- The probability that k successive calls each return the wrong answer is at most 2^{-k}, so it is $(1 - 2^{-k})$ correct.
- Alternatively, Monte Carlo algorithms can be given an explicit upper bound on the tolerable error probability in $O(n^2 \log \varepsilon^{-1})$.

```
Function Frievalds (A, B, C, n)
    for j=1 to n do Xj = Uniform(0..1)
      if (XA)B = XC then return true
      else return false.
```

- The above algorithm returns a correct answer with probability at least one-half on every instance. It is not p-correct for any p smaller than ½, however, because its error probability is exactly ½ when C differs from AB in precisely one row. An incorrect answer is returned iff $X_i=0$ where it is row i that differs between C and AB.

Example:

Consider the following 3 X 3 matrices,

$$A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \qquad B = \begin{matrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{matrix} \qquad \text{and} \qquad C = \begin{matrix} 11 & 29 & 37 \\ 29 & 65 & 91 \\ 47 & 99 & 45 \end{matrix}$$

- A Call on Frievalds(A, B, C, 3) could choose X = (1, 1, 0) in which case XA is obtained by adding the first and second rows of A, thus XA= (5, 7, 9). Continuing, we calculate (XA)B = (40, 94, 128). This is compared to XC = (40, 94, 128), which is obtained by adding the first and second rows of C.
- With this choice of X, Frievalds might choose X= (0, 1, 1) instead. This time calculations are, XA = (11,13, 15); XA(B) = (76, 166, 236) and XC = (76, 164, 136)
- Frievalds () returns false. So the fact that XAB ≠ XC is conclusive proof that AB ≠ C.
- To handle the Frievalds() running several times on the same instatnce, a new parameter K is introduced, so

```
Function Repeat_Freivalds (A, B, C, n, K)
  for I = 1 to K do
        if Freivalds(A, B, C, n) = false then
              return true
        else
              return false.
```

(ii) Monte Carlo Algorithm for Primality testing

Problem: Deciding whether the given odd integer is prime of composite.

- When the number to be tested has more than few hundred decimal digits, then testing primality of such a large number is more than a mathematical recreation.
- Takes $O(2^d/2)$ to test whether a *d*-digit number is a prime

Fermat's Little Theorem:
Let n be prime.
Then $a^{n-1} \bmod n = 1$
for any integer a such that $1 \le a \le n - 1$.

Randomized polynomial-time algorithm: If the algorithm declares that the number is not prime, then it is certainly not a prime. If the algorithm declares that the number is a prime, then with high probability but not 100% sure, the number is prime.

Fermat's Lesser Theorem: If P is prime, and $0 < A < P$, then $A^{P-1} = 1 \% P$. Pick $1 < A < N-1$ at random. If $A^{N-1} = 1 \% N$, declare that N is probably prime, otherwise declare that N is definitely not prime.

False witness of primality: Carmichael numbers are not prime but satisfy $A^{N-1} = 1 \% N$ for all $0 < A < N$ that are relatively prime to N.

If P is prime and $0 < X < P$, the only solutions to $X^2 = 1 \% P$ are $X = 1, P\text{-}1$.
Whenever the Fermat's algorithm tells the given integer n is composite, it provides no clue concerning its divisors. And more factorization is much harder than primality.
The revised fermat's algorithm for testing primality as

Fermat's Theorem:
Let n be prime.
Then $a^t \bmod n = 1$, or
there exists an integer i, $0 \le i \le s$, such that $a^{2^i.t} \bmod n = n-1$.

Applications of Monte Carlo Method

- Evaluating integrals of arbitrary functions of 6+ dimensions
- Predicting future values of stocks
- Solving partial differential equations
- Sharpening satellite images
- Modeling cell populations
- Finding approximate solutions to NP-hard problems

LAS VEGAS ALGORITHMS

- Las Vegas algorithms make probabilistic choices to help guide them more quickly to a correct solution, they never return a wrong answer.
- Two main categories of Las Vegas algorithms:
  - It take longer time to solve a problem when unfortunate choice are made (e.g., Quicksort), and alternatively,
  - They allow themselves go to a dead end and admit that they cannot find a solution in this run of the algorithm.
- A Las Vegas algorithm has the Robin Hood effect, with high probability, instances that took a long time deterministically are now solved much faster, but instances on which the deterministic algorithm was particularly good are slowed down to average. Let $p(x)$ be the probability of success of the algorithm, then the expected time $t(x)$ is $1/p(x)$. However, a correct analysis must consider separately the expected time taken by LV(x) in case of success $s(x)$ and in case of failure $f(x)$. $t(x) = s(x) + ((1-p(x))/p(x))f(x)$.

The Eight Queens Problem

- Combine backtracking with probabilistic algorithm, first places a number of queens on the board in a random way, and then uses backtracking to try and add the remaining queens without reconsidering the positions of the queens that were placed randomly.
- The more queens we place randomly, the smaller the average time needed by the subsequent backtracking stage, whether it fails or succeeds, but the greater the probability of failure. This is the fine-tuning knob.

Probabilistic Quickselect and Quicksort

Las Vegas hashing allows us to retain the efficiency of hashing on the average, without arbitrarily favoring some programs at the expense of others. Choose the hash function randomly at the beginning of each compilation and again whenever rehashing becomes necessary, ensure that collision lists remain reasonably well-balanced with high probability.

Universal hashing: Let U = {1,2, .., a-1} be the universe of potential indexes for the associative table, and let B = {1,2, ..,N-1} be the set of indexes in the hash table. Let two distinct x and y in U, a set H of functions from U to B, and h:$U \rightarrow B$ is a function chosen randomly from H, H is a universal2 class of hash functions if the probability that $h(x) = h(y)$ is at most $1/N$. Let p be a prime number at least as large as a, and i, j be two integers ($1 \leq i < p$ and $0 \leq j < p$), then $h_{ij}(x) = ((ix + j)\%p)\%N$, and H is universal.

Factorizing Large Integers

The factorization problem consists of finding the unique decomposition of n into a product of prime factors. The splitting consists of finding one nontrivial divisor of n, provided n is composite. Factorizing reduces to splitting and primality testing. An integer is k-smooth if all its prime divisors are among the k smallest prime numbers. k-smooth integers can be factorized efficiently by trial division if k is small. A hard composite number is the product of two primes of roughly equal size.

Let n be a composite integer, Let a and b be distinct integers between 1 and n-1 such that $a + b$ n. If $a^2 \% n = b^2 \% n$, then gcd (a+b, n) is a nontrivial divisor of n.

CLASS P AND CLASS NP

Polynomial Problem / Algorithm

- An algorithm is said to be polynomially bounded if its worst-case complexity is bounded by a polynomial function of the input size.
- A problem is said to be polynomially bounded if there is a polynomially bounded algorithm for it.

Class P

- P is the class of all decision problems that are polynomially bounded. The implication is that a decision problem X in P can be solved in polynomial time on a deterministic computation model (or can be solved by deterministic algorithm).
- A deterministic machine, at each point in time, executes an instruction. Depending on the outcome of executing the instruction, it then executes some next instruction, which is unique.
- *The class P consists of problems that are solvable in polynomial time. These problems are also called* ***tractable****. The advantages in considering the class of polynomial-time algorithms is that all reasonable* ***deterministic single processor model of computation*** *can be simulated on each other.*

Class NP (**NP** stands for **N**on-deterministically **P**olynomial**)**

- NP represents the class of problems which can be solved in polynomial time by a Non-deterministic model of computation (or by a non-deterministic algorithm). That is, a problem X in NP can be solved in polynomial-time on a non-deterministic computation model. A non-deterministic model can make the right guesses on every move and race towards the solution much faster than a deterministic model.
- A non-deterministic machine has a choice of next steps. It is free to choose any step that it wishes. For example, it can always choose a next step that leads to the best solution for the problem. A non-deterministic machine thus has the power of extremely good, optimal guessing.
- *The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct. Every problem in this class can be solved in exponential time using exhaustive search.*

OPTIMIZATION PROBLEM

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

- Finding the minimum number of colors needed to color a given graph.
- Finding the shortest path between two vertices in a graph.

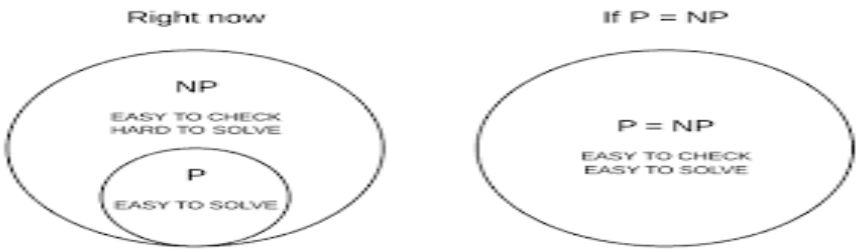
DECISION PROBLEM

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

- Whether a given graph can be colored by only 4-colors.
- Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

CLASS - P AND CLASS - NP

- Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.
- All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP - P* are intractable.
- It is not known whether ***P = NP***. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.
- If ***P ≠ NP***, there are problems in NP that are neither in P nor in NP-Complete.
- The problem belongs to class **P** if it's easy to find a solution for the problem. The problem belongs to **NP**, if it's easy to check a solution that may have been very tedious to find.



NON-DETERMINISTIC ALGORITHMS

The deterministic algorithms has the property that the result of every operation is uniquely defined, whereas the non-deterministic algorithm contains operations whose outcomes are not uniquely defined but are limited to specified set of possibilities.

To specify such algorithms some new functions are introduced as follows:

Choice (S) – arbitrarily chooses one of the elements of set S. There is no rule specifying how this choice is to be made.

Failure () – signals an unsuccessful completion. The Non-deterministic algorithm terminates unsuccessfully iff there exists no set of choices leading to a success signal.

Success () – signals a successful completion. The computing time of all these three functions are taken to be O(1), Constant.

Sample Non-deterministic Algorithms

Algorithm ND_**Search** (A, n, Key)
j= Choice(1..n)
ifA[j] == Key then Write (j); Success();
else Write(0); Failure();
End
	The **Non-deterministic search algorithm** taking an array A[1..n] containing n elements and the element to be searched, Key.
	The ND function, Choice() will return the position of Key in A[1..n]. Time Complexity=O(1)

Algorithm ND **Sort**(A, n)
For i=1 to n do
j=Choice(1..n);
B[j] = A[i];
For i=1 to n do // verify the order
If B[i] > B[i+1] then Failure();
Write (B[1..n])
Success();
End
	The function Choice() will read the i^{th} element from the array A[1..n] and return the proper position where it has to be placed in the sorted sequence.

Algorithm ND **Knapsack** (n, p, w, m)
p=0; w=0;
for i= 1 to n do
x[i] = Choice(0,1);
w=w+x[i].w[i]; p=p+x[i].p[i];
If w>m then Failure();
Success();
End
	Knapsack decision problem-the Choice () function assigns 0 or 1 value to the solution vector x[1..n] at every step.
	Time Complexity= O(n), Linear.

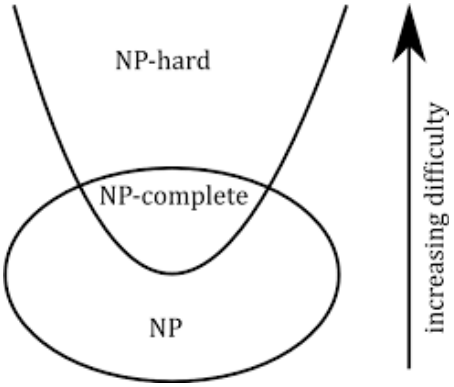
Algorithm ND **Clique** (G, k)
S=∅
For i= 1 to k do
t=Choice(1..n);
if t ∈S then Failure();
S= S ∪ t
For all pairs (i,j) such that i,j ∈S & i≠ j do
If (i,j) $not\ in$ E(G) then Failure();
Write(S); Success();
End.
	Clique Decision problem the nondeterministic function Choice() will determine the subset of k out of n vertices that are forming a clique in linear time.
	Time Complexity = O(n), Linear.

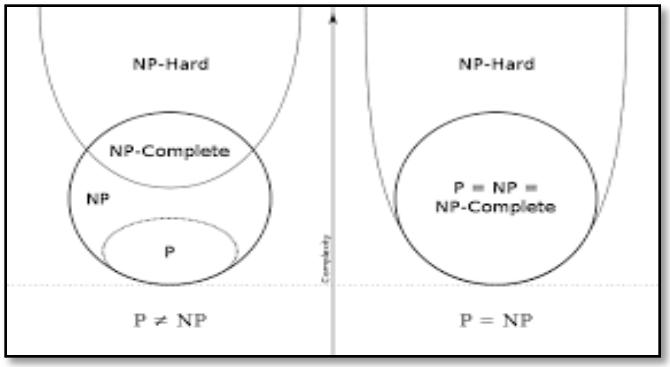
NP – HARD & NP – COMPLETE

A problem / language **B** is ***NP-complete*** if it satisfies the following two conditions:

1. The problem **B** is in Class NP
2. Every problem **A** in NP is reducible to **B** in polynomial time.

If a problem or a language satisfies the second property, but not necessarily the first one, then the problem / language **B** is known as **NP-Hard**.





- Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that reduces to **B**.
- The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NP-Complete, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

Polynomially equivalent problems:

- Two problems L_1 and L_2 are said to be polynomially equivalent if and only if $L_1 \propto L_2$ and $L_1 \propto L_2$.
- To show that a problem L_2 is NP-hard, it is adequate to show $L_1 \propto L_2$ where L_1 is some problem already known to be NP – Hard.
- Since $\propto$ is transitive relation, it follows that if $L_1 \propto L_2$ and $L_2 \propto L_3$, then L_3 also NP –Hard.
- To show that an NP-hard problem A is NP-Complete, just exhibit a polynomial time *Non-Deterministic Algorithm* for A.

REDUCTION ($\propto$)

How to prove some problems are computationally difficult?

Consider the statement: “Problem X is at least as hard as problem Y”,

To prove such a statement: Reduce problem Y to problem X.

(i) If problem Y can be reduced to problem X, we denote this by $Y \leq_P X$ or $X \propto Y$.

(ii) This means “Y is polynomal-time reducible to X.”

(iii) It also means that X is at least as hard as Y because if we can solve X, we can solve Y.

NP-HARD GRAPH PROBLEMS

1. CDP – CLIQUE DECISION PROBLEM

Problem Description: Given a graph G = (V, E) and a positive integer K, the Clique Decision Problem is to determine whether the graph G contains a CLIQUE of size K or not.

To Show CDP is in Class – NP: [Verify the result in polynomial time]

For the given graph G=(V, E), use the subset V'⊆ V of vertices in the clique as a certificate for G. Checking V' is a clique in polynomial time for each pair u ∈ V', v ∈ V' and the edge (u,v) ∈ E, requires $O(n^2)$.

To Show CDP is NP-hard: [Reduce an instance of known NP-hard problem into CDP Instance]

(i) Pick a known NP-hard problem: CNF – Satisfiability Problem.
(ii) Transform (Reduce) the instance of CNF – SAT into an instance of CDP problem in polynomial time.
(iii) If CNF – SAT ∝ CDP, then CDP is also NP – Hard.

Proof: Let F = $\bigwedge_{i=1}^{k} Ci$ can be a propositional calculus formula in CNF having K clauses and let x_i for 1≤i≤n be the n boolean variables or literals used in F. Construct from F, a graph G=(V,E) such that G has a clique of size atleast K if and only if F is satisfiable.

For any F, the graph G = (V, E) is defined as follows:

The vertices V = {<a, i> | a is a literal / variable in the clause Ci (i.e) a ∈ Ci}

The edges E = {(<a,i>,<b,j>) | a and b belongs to different Clauses (i.e) i≠ j & a≠b'}

Claim: F is satisfiable if and only if G has a Clique of size atleast K.

Proof of Claim: If F is satisfiable, then there is a set of truth values for all xi such that each clause is true with this assignment.

Let S = { <a,i> | a is true in Ci} be a set containing exactly one <a,i> for each i. Between any two nodes <a,i> and <b,j> in S there is an edge in G since i≠j and both a and b have the true value. Thus S forms a clique of size K.

Example: Please refer Class-work note-book.

2. NCDP – NODE COVER DECISION PROBLEM

Problem Description: A set S ⊆ V is a node cover for a graph G=(V,E) if and only if all edges in E are incident to atleast one vertex in the set S.

To Show NCDP is in Class – NP: [Verify the vertex cover in polynomial time]

- For the given graph G=(V, E), use the subset S ⊆ V of vertices in the vertex cover as a certificate for G. Checking vertices in S is covering all the edges of E requires O(|E|), Linear
- Hence the verification of node cover is done in polynomial time, NCDP is Class NP.

To Show NCDP is NP-hard: [Reduce an instance of known NP-hard problem into NCDP Instance]

(i) Pick a known NP-hard problem: CDP – Clique Decision Problem.
(ii) Transform (Reduce) the instance of CDP into an instance of NCDP problem in polynomial time.
(iii) If CDP ∝ NCDP, then NCDP is also NP – Hard.

Proof: Let G = (V, E) and K defines an instance of CDP. Let |V| = n, Construct from G, a new graph G'=(V,E') such that G' has a node cover of atmost n-K vertices if and only if G has a clique of size K.

The graph G' = (V, E') is defined as follows:

The edges E' = {(u,v)|u ∈ V, v ∈ V and (u,v)∉ E}. The graph G' is the complement of G.

Claim: The graph G has a clique of size K if and only if G' has a vertex cover of atmost n – K.

Proof of Claim:

- Let K be any clique in G, since there are no edges in E' connecting vertices in K, the remaining n - |K| vertices in G' must cover all edges in E'.
- Hence, G' can be obtained from G in polynomial time, CDP can be solved in polynomial time if we have a polynomial time deterministic algorithm for NCDP.
- So, NCDP is also NP-Hard.

Example: Please refer Classwork note-book.

3. CNDP – Chromatic Number Decision Problem

Problem Description: A colouring of a graph G=(V,E) is a function f: V = {1, 2, …k} defined for every i∈V. If any edge (u,v) is in E then f(u) ≠ f(v). The Chromatic number decision problem is to determine whether G has a colouring for a given k.

To Show CNDP is in Class – NP: [Verify the coloured graph in polynomial time]

- Given a graph G=(V,E) and a positive integer m is it possible to assign one of the numbers(colours) 1, 2, ..., m to vertices of G so that for no edge in E is it true that the vertices on that edge have been assigned the same colour.
- The verification of coloured grpah is done in polynomial time and then CNDP is Class NP.

To Show CNDP is NP-hard: [Reduce an instance of known NP-hard problem into CNDP Instance]

(i) Pick a known NP-hard problem: CNF – Satisfiability Problem.
(ii) Transform (Reduce) the instance of CNF – SAT problem into an instance of CNDP in polynomial time.
(iii) If CNF – SAT ∝ CNDP, then CNDP is also NP – Hard.

Proof: Let F be a propositional calculus formula having atmost three literals in each clause and having 'r' clauses C_1, C_2, …C_r. Let x_i , $1 \le i \le n$ be the 'n' boolean variables or literals used in F. Construct a polynomial time graph G that is n+1 colourable if and only if F is satisfiable.

For any F, the graph G = (V, E) is defined as:
$V = \{x_1,x_2,\ldots x_n\} \cup \{x'_1,x'_2,\ldots x'_n\} \cup \{ y_1,y_2,\ldots y_n\} \cup \{C_1, C_2, \ldots,C_r\}$, where $y_1, y_2,\ldots, y_n$ are new variables.
$E = \{(x_i, x_i'),1 \le i \le n\} \cup \{(y_i, y_j) | i \ne j\} \cup \{(y_i, x_j) | i \ne j\} \cup \{(y_i, x'_j) | i \ne j\} \cup \{(x_i, C_j) | xi \notin Cj\} \cup \{(x'_i, C_j) | x'_i \notin Cj\}$.

Claim: A graph G is n+1 colorable if and only if F is satisfiable.

Proof of Claim:

- First observe all yi form a complete sub-graph on n vertices. Since yi is connected to all the xj and xj' except xi and xi', the colour 'i' can be assigned to only xi and xi'. However (xi, xi') is in E and also a new colour n+1 is needed for one of these vertices. The vertex assigned a new colour n+1 is called a false vertex.
- Each Clause has atmost three literals, each Ci is adjacent to a pair of vertices xj, xj' for atleast one j. so no Ci can be assigned the colour n+1. This imply that only colours that can be assigned to Ci correspond to vertices xj or xj' that are in clause Ci and are true vertices. hence G is n+1 colourable if and only if there is a true vertex corresponding to each Ci.
- So G is n+1 colourable iff F is satisfiable.

NP HARD SCHEDULING PROBLEMS

1. SCHEDULING IDENTICAL PROCESSORS

Problem Description:

- There are m identical processors (or) machines, P_1, P_2, …, P_m
- There are n different jobs J_1, J_2, …, J_n to be processed.
- Each job J_i requires some t_i processing time.
- A schedule S is an assignment of jobs to processors, which specifies the time interval and the processor on which the job J_i is to be processed.
- A Schedule can be either a non-preemptive schedule (the processing of a job is not terminated until the job is complete) or a preemptive schedule.
- Constraint: A job can't be processed by more than one processor at any given time.
- The problem is obtaining a minimum finish time non-preemptive schedule.

The Mean Finish Time (MFT) of a schedule S is:

MFT (S) = $\frac{1}{n}\sum_{1\le i\le n} fi$, where fi is the time at which the processing of job J_i is completed.

The Weighted Mean Finish Time (WMFT) of a schedule S is:

WMFT (S) = $\frac{1}{n}\sum_{1\le i\le n} wi.fi$, where w_i is the weight associated with each job J_i.

The overall Finish Time (FT) of a schedule S is:

FT (S) = $\mathbf{MAX}_{1\le i\le n}\, Ti$, where Ti is the time at which the processor Pi finishes processing all the jobs assigned to it.

To show the Minimum finish time non-preemptive schedule is NP – Hard

(i) Choose a known NP – hard problem: Partition problem.

(ii) Reduce Partition ∝ MFT Non-Preemptive Schedule Instance.

Proof: Let a_i, $1 \le i \le n$, be an instance of the Partition problem. Define n jobs with processing time requirements $t_i = a_i$, $1 \le i \le n$. There is a non-preemptive MFT schedule for this set of jobs on two processors with finish time at most $\frac{1}{2}\sum_{1\le i\le n} ti$, if and only if there is a partition of the a_i's.

Claim: There is a minimum WMFT non-preemptive schedule with n jobs on two processors iff there is a partition of the a_i's.

Proof of Claim: With the instance of the partition problem, construct a two-processor scheduling problem with n jobs and $w_i = t_i = a_i$, $1 \le i \le n$.

For this set of jobs the schedule S with weighted MFT atmost ½ $\sum ai^2$ + ¼ $(\sum ai)^2$ iff the a_i's have a partition.

Let the weights and times of jobs on Processor P1 be (w_1, t_1), (w_2, t_2),…(w_k, t_k) and on the processor P2 be (w_1', t_1'), (w_2', t_2'),…(w_j', t_j') and the order the in which the jobs are processed.

Then the schedule S is:

WMFT (S) = $\{w_1t_1 + w_2 (t_1+t_2)+ \ldots+ w_k(t_1+t_2+\ldots+ t_k)\} + \{w_1't_1'+ w_2' (t_1'+t_2')+ \ldots+w_k'(t_1'+t_2'+\ldots+ t_k')\}$

$$= ½ \sum wi^2 + ¼ (\sum wi')^2 + ½ (\sum wi - \sum wi')^2$$

This is obtainable if and only if wi' s and so also ai's have a partition.

Thus the MWFT Non-preemptive Schedule problem is NP – Hard.

2. FLOW SHOP SCHEDULING

Problem Description:

- There are m processors $P_1, P_2, P_3 \ldots P_m$ & there are n jobs $J_1, J_2, J_3 \ldots J_n$
- Each job J_i, $1 \le i \le n$ requires processing on every processor P_j, $1 \le j \le m$ in sequence.
- Each job requiring m tasks $T_{1,i}, T_{2,i}, \ldots T_{m,i}$ for $1 \le i \le n$ to be performed and task $T_{j,i}$ must be assigned to processor P_j.
- A Schedule for the n jobs is an assignment of tasks to time intervals on the processors.
- For any job J_k the processing of task $T_{j,k}$, $k>1$ can not be started until task $T_{j-1,k}$ has been completed.
- The problem of flow shop sequencing is to assign jobs to each assigned processors in a manner that the every processors are engaged all the time without being left ideal.
- Obtain a minimum finish time preemptive schedule is NP – hard.

To show the Minimum finish time preemptive FS schedule is NP – Hard

(i) Choose a known NP – hard problem: Partition problem.

(ii) Reduce Partition $\propto$ Minimum FT Preemptive FS Schedule Instance.

Proof: Let a_i, $1 \le i \le n$, be an instance of the Partition problem.

Let m= 3, construct the following preemptive FS instance with n+2 jobs with atmost two non-zero tasks per job.

$t_{1,i} = a_i$	$t_{2,i} = 0$	$t_{3,i} = a_i$, $1 \le i \le n$.
$t_{1,n+1} = \frac{T}{2}$	$t_{2,n+1} = T$	$t_{3,n+1} = 0$
$t_{1,n+2} = 0$	$t_{2,n+2} = T$	$t_{3,n+2} = \frac{T}{2}$, where $T = \sum_{i=1}^{n} a_i$

Claim: The constructed FS instance has a preemptive schedule with finish time at most 2T if and only if A has a partition.

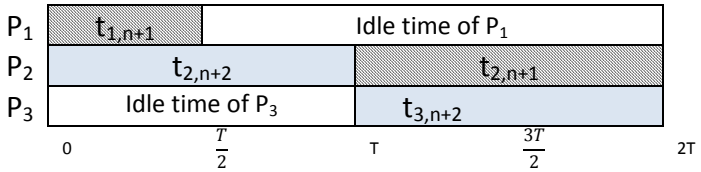
Proof of claim: If a partition problem instance A has a partition u, then there is a non-preemptive schedule with finish time 2T. If A has no partition then all preemptive schedule for JS must have a finish time greater than 2T. It can be shown by contradiction.

Let assume that there is a preemptive schedule for FS with finish time at most 2T:

(a) Task $t_{1,n+1}$ must finish by time T as $t_{2,n+1} = T$ and can not start until $t_{1,n+1}$ finishes.

(b) Task $t_{3,n+2}$ can not start before T units of time have elapsed as $t_{2,n+2}=T$.

From the above observations,



Let V be the set of indices of tasks completed on processor P_1 by time T excluding task $t_{1,n+1}$, $\sum_{i\in V} t_{1,i} < \frac{T}{2}$ as A has no partition. hence, $\sum_{i\notin V} t_{3,i} > \frac{T}{2}$

Thus the processing of jobs not included in V can not commence on processor P3 until after time T since their processor P1 processing is not completed until after T. So total amount of processing time left for processor P3 at time T is $t_{3,n+2} + \sum_{i\notin V} t_{3,i} > T$.
The schedule length must therefore be more than 2T.

3. JOB SHOP SCHEDULING

Problem Description:

- There are m processors P_1, P_2, P_3…..P_m & there are n jobs J_1, J_2, J_3…..J_n
- The time of the j^{th} task of J_i is denoted as $t_{k,i,j}$ and the task is to be processed by P_k
- The task for any job are to be carried out in the order 1, 2, 3, and so on. Task j can not begin until task j-1 has been completed
- Obtaining a minimum finish time preemptive or non-preemptive schedule is NP hard when m=2.

To show the Minimum finish time preemptive JS schedule is NP – Hard

- (i) Choose a known NP – hard problem: Partition problem.
- (ii) Reduce Partition ∝ Minimum FT Preemptive JS Schedule Instance.

Proof: Let a_i, $1 \le i \le n$, be an instance of the Partition problem. Construct the following JS instance with n+1 jobs and m=2 processors.

Jobs 1…n : $t_{1,i,1} = t_{2,i,2} = a_i$ for $1 \le i \le n$.

Job n+1 : $t_{1,n+1,1} = t_{2,n+1,2} = t_{2,n+1,3} = t_{1,n+1,4} = \frac{T}{2}$

Claim: The job shop instance has a preemptive schedule with finish time at most 2T if and only if A has a partition.

Proof of claim:

- If A has a partition u then there is a schedule with finish time 2T.
- If A has no partition then all schedules for JS must have a finish time greater than 2T.
- Let assume that there is a schedule S with finish time at most 2T.
- There can be no idle time on either P_1 or P_2.
- Let R be the set of jobs scheduled for P1 in the interval $[0 - \frac{T}{2}]$. Let R' be the subset of R representing jobs whose first task is completed on P1 in this interval.
- Since A has no partition $\sum_{j\in R'} t_{i,j,1} < \frac{T}{2}$, consequently $\sum_{j\in R'} t_{2,j,2} < \frac{T}{2}$
- Since only the second task of jobs in R' can be scheduled on P2 in the interval $[\frac{T}{2}$, T]. It follows that there is some idle time on P2 in this interval.

Hence S must have a finish time greater than 2T.

4. APPROXIMATION ALGORITHMS

- An algorithm that runs in polynomial time and yields a solution close to the optimal solution is called an approximation algorithm.
- We will explore polynomial-time approximation algorithms for several NP-Hard problems.

Formal Definition:

- Let P be a minimization problem and I be an instance of P.
- Let A be an algorithm that finds feasible solution to instances of P.
- Let A(I) is the cost of the solution returned by A for instance I and OPT(I) is the cost of the optimal solution for I. Then, A is said to be an α-approximation algorithm for P if,

$$\forall_I, \frac{A(I)}{OPT\ (I)} \leq \alpha, \text{ where } \alpha \geq 1.$$

- So any minimum optimization problem A(I) ≥ OPT(I). Therefore, 1-approximation algorithm produces an optimal solution.
- An approximation algorithm with a large α may return a solution that is much worse than optimal. So the smaller α is, the better quality of the approximation the algorithm produces.

For instance size n, the most common approximation classes are:

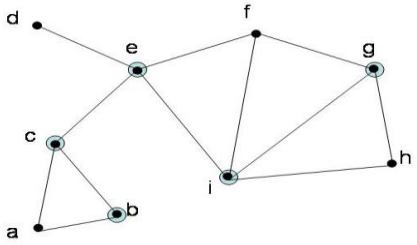
a. α = O(n c) for c < 1, e.g. Clique.
b. α = O(log n), e.g. Set Cover.
c. α = O(1), e.g. Vertex Cover.

1. VERTEX COVER PROBLEM

Problem Description:

- Given a graph G = (V, E), find a minimum subset C⊆V, such that C covers all edges in E, i.e., every edge ∈ E is incident to at least one vertex in C.
- The optimum vertex cover must cover every edge in E (G). So, it must include at least one of the endpoints of each edge ∈ E (G).

Example: Consider the following graph G = (V, E)
V = {a, b, c, d, e, f, g, h, i}
E = {(a,b), (a,c), (b,c), (c,e), (d,e), (e,f), (e,i), (f,g), (f,i), (g,h), (g,i), (h,i)}



An optimal ***vertex cover*** for the graph is
C = {b, c, e, i, g}.

Algorithm APPROX – VERTEX – COVER (G)

C = ∅; E' = E
Mark the degree of every vertex in V
while C does not cover all edges in E'
    pick a vertex u with highest degree
    C = C ∪ {u}
    Delete a vertex u and all the edges incident on u, from E'
return C

2. TRAVELING SALESPERSON PROBLEM

Problem Description:

- A salesman wants to visit each of n cities exactly once each, minimizing total distance travelled and returning to the starting point.
- ***Input:*** A complete undirected graph G = (V,E), with edge weights (costs) w, and |V| = n.
- ***Output:*** A tour (cycle that visits all n vertices exactly once and returning to starting vertex) of minimum cost.

Approximation Algorithm for metric TSP

A metric space is a pair (S, d), where S is a set and d is a distance function that satisfies, for all u, v, w ∈ S, the following conditions.

1. d (u, v) = d (v, u)
2. d (u, v) + d(v, w) ≥ d (u, w) (***triangle inequality*** – a least distance to reach a vertex w from vertex u is always to reach w directly from u, rather than through some other vertex v).

The approximation algorithm works only if the problem instance satisfies Triangle – Inequality.

Algorithm APPROX – TSP - TOUR (G)

1. Select a starting vertex r ∈ V, to be a root vertex.
2. Compute a weighted MST T for G from r.
3. Traverse the MST T, in pre-order: $v_1, v_2, \ldots, v_n$.
4. Return tour: $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow v_1$.

3. SET COVER PROBLEM

- An Instance (X, F) of the set-covering problem consists of a finite set X and a family F of subset of X, such that every element of X belongs to at least one subset of F,

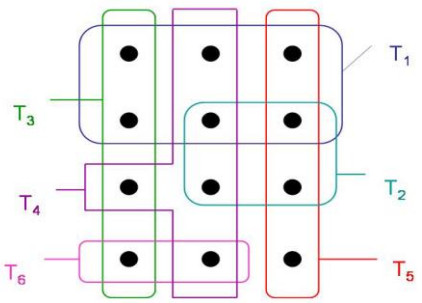
$$X = \bigcup_{S \in F} S.$$

- Let say that a subset S ∈ F covers all elements in X. *The goal is to find a minimum size subset C∈ F whose members cover all of X.*

$$X = \bigcup_{S \in C} S.$$

- The cost of the set-covering is the size of C, which defines as the number of sets it contains, and we want |C| to be minimum.

Example 1: Consider an instance (X, F) of set-covering problem. Here, X consists of 12 vertices and F = {T1, T2, T3, T4, T5, T6}. The minimum size set cover is C = {T3, T4, T5} and it has the size of 3.



Greedy Approximation Algorithm for Set Cover

- Idea: At each stage, the greedy algorithm picks the set S ∈ F that covers the greatest numbers of elements not yet covered.

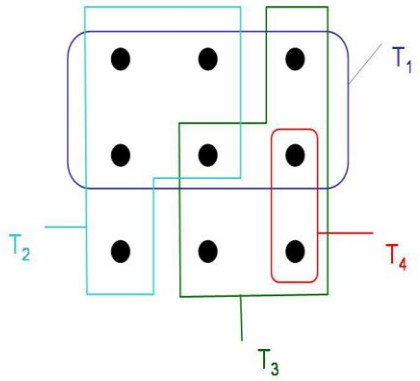
The description of this algorithm as following:

1. First, start with an empty set C.
2. Let K contain, at each stage, the set of remaining uncovered elements.
3. While there exists remaining uncovered elements,
    a. choose the set S from F that covers as many uncovered elements as possible,
    b. put that set in C and
    c. Remove these covered elements from K.
4. When all element are covered, C contains a subfamily of F that covers X, return C

Algorithm GREEDY – SET COVER (X, F)

K = X
C =Ø
While K ≠ Ø do
    Select a subset S ∈ F that maximizes |S ∩ K|
    K = K – S
    C = C ∪ {S}
Return C

Example 2: An Instance (X, F) of set-covering problem where X consists of 9 vertices and F = {T1, T2, T3, T4}.



The greedy algorithm produces set cover of size 3 by selecting the sets T1, T3 and T2 in order.