(1) Algorithm and Properties of Algorithm

Algorithm

- An algorithm is any well-defined computational procedure that takes some values or set of values as input and produces some values or set of values as output.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

Properties of Algorithm

All the algorithms should satisfy following five properties,

1) **Input:** There is zero or more quantities supplied as input to the algorithm.
2) **Output:** By using inputs which are externally supplied the algorithm produces at least one quantity as output.
3) **Definiteness:** The instructions used in the algorithm specify one or more operations. These operations must be clear and unambiguous. This implies that each of these operations must be definite; clearly specifying what is to be done.
4) **Finiteness:** Algorithm must terminate after some finite number of steps for all cases.
5) **Effectiveness:** The instructions which are used to accomplish the task must be basic i.e. the human being can trace the instructions by using paper and pencil in every way.

(2) Mathematics for Algorithmic Set

Set

Unordered collection of distinct elements can be represented either by property or by value.

Set Cardinality

- The number of elements in a set is called cardinality or size of the set, denoted |S| or sometimes n(S).
- The two sets have same cardinality if their elements can be put into a one-to-one correspondence.

It is easy to see that the cardinality of an empty set is zero i.e., | ∅ |.

Multiset

- If we do want to take the number of occurrences of members into account, we call the group a multiset.
- For example, {7} and {7, 7} are identical as set but {7} and {7, 7} are different as multiset.

Infinite Set

A set contains infinite elements. For example, set of negative integers, set of integers, etc…

Empty Set

Set contain no member, denoted as { } or ∅.

Subset

For two sets A and B, we say that A is a subset of B, written A B, if every member of set A is also a member of set B. Formally, A $\subseteq$ B if x $\in$ A implies x $\in$ B

Proper Subset

Set A is a proper subset of B, written A $\subset$ B, if A is a subset of B and not equal to B. That is, a set A is proper subset of B if A $\subseteq$ B but A $\neq$ B.

Equal Sets

The sets A and B are equal, written A = B, if each is a subset of the other.

Let A and B be sets. A = B if A $\subseteq$ B and B $\subseteq$ A.

Power Set

- Let A be the set. The power of A, written P(A) or 2^A, is the set of all subsets of A. That is, P (A) = {B: B $\subseteq$ A}.
- For example, consider A= {0, 1}.

  The power set of A is {{}, {0}, {1}, {0, 1}}.

Union of sets

- The union of A and B, written A $\cup$ B, is the set we get by combining all elements in A and B into a single set.
- That is, A $\cup$ B = {x: x $\in$ A or x $\in$ B}.

Disjoint sets

- Let A and B be sets. A and B are disjoint if A $\cap$ B = Φ .

Intersection sets

- The intersection of set A and B, written A $\cap$ B, is the set of elements that are both in A and in B. That is, A $\cap$ B = { x : x $\in$ A and x $\in$ B}.

Difference of Sets

- Let A and B be sets. The difference of A and B is A - B = {x : x $\in$ A and x $\notin$ B}.
- For example, let A = {1, 2, 3} and B = {2, 4, 6, 8}. The set difference A - B = {1, 3} while B-A = {4, 6, 8}.

Complement of a set

- All set under consideration are subset of some large set U called universal set.
- Given a universal set U, the complement of A, written A', is the set of all elements under consideration that are not in A.
- Formally, let A be a subset of universal set U.
- The complement of A in U is A' = A - U OR A' = {x: x $\in$ U and x $\notin$ A}.

Symmetric difference

- Let A and B be sets. The symmetric difference of A and B is,

  $A \oplus B = \{ x : x \in A \text{ or } x \in B \text{ but not in both} \}$

As an example, consider the following two sets A = {1, 2, 3} and B = {2, 4, 6, 8}. The symmetric difference, $A \oplus B = \{1, 3, 4, 6, 8\}$.

Sequences

- A sequence of objects is a list of objects in some order. For example, the sequence 7, 21, 57 would be written as (7, 21, 57). In a set the order does not matter but in a sequence it does.
- Repetition is not permitted in a set but repetition is permitted in a sequence. So, (7, 7, 21, 57) is different from {7, 21, 57}.

(3) Functions & Relations

Relation

- Let X and Y be two sets. Any subset ρ of their Cartesian product X x Y is a relation.
- When x ∈ X and y ∈ Y, we say that x is in relation with y according to ρ denoted as x ρ y if and only if (x, y) ∈ ρ.
- Relationship between two sets of numbers is known as function. Function is special kind of relation.
- A number in one set is mapped to number in another set by the function.

But note that function maps values to one value only. Two values in one set could map to one value but one value must never map to two values.

Function

- Consider any relation f between x and y.
- The relation is called a function if for each x ∈ X, there exists one and only one y ∈ Y such that (x, y) ∈ f.
- This is denoted as f : x → y, which is read as f is a function from x to y denoted as f(x).
- The set X is called domain of function, set Y is its image and the set $f(D) = \{f(x) \mid x \in D\}$ is its range.
- For example, if we write function as follow

$$f(x) = x^3$$

- Then we can say that $f(x)$ equals to x cube.
- For following values it gives result as,

$$f(-1) = (-1)^3$$

$$f(2) = (2)^3 = 8$$

$$f(3) = (3)^3 = 27$$

$$f(5) = (5)^3 = 125$$

This function $f(x)$ maps number to their cube.

- In general we can say that a relation is any subset of the Cartesian product of its domain and co domain.
- The function maps only one value from domain to its co domain while relation maps one value from domain to more than one values of its co domain.
- So that by using this concept we can say all functions are considered as relation also but not vice versa.

Properties of the Relation

- Different relations can observe some special properties namely reflexive, symmetric, transitive and Anti symmetric.

Reflexive:

- When for all values x: $x\ R\ x$ is true then relation R is said to be reflexive.
- E.g. the equality (=) relation is reflexive.

Symmetric:

- When for all values of x and y, $x\ R\ y \rightarrow y\ R\ x$ is true. Then we can say that relation R is symmetric. Equality (=) relation is also symmetric.

Transitive:

- When for all values of x, y and z, x R y and y R z then we can say that x R z, which is known as transitive property of the relation.
- E.g. the relation grater then > is transitive relation.
- If x>y and y>z then we can say that x>z i.e. x is greater than y and y is greater than z then x is also greater than z.

Anti-symmetric:

- When for all values of x and y if x R y and y R x implies x=y then relation R is Anti- symmetric.
- Anti-symmetric property and symmetric properties are lookalike same but they are different.
- E.g. consider the relation greater than or equal to $\geq$ if $x \geq y$ and $y \geq x$ then we can say that y = x.
- A relation is Anti-symmetric if and only if $x \in X$ and $(x, x) \in R$.

Equivalence Relation:

- Equivalence Relation plays an important role in discrete mathematics.
- Equivalent relation declares or shows some kind of equality or says equivalence.
- The relation is equivalent only when it satisfies all following property i.e. relation must be reflexive, symmetric and transitive then it is called Equivalence Relation.
- E.g. Equality '=' relation is equivalence relation because equality proves above condition i.e. it is reflexive, symmetric and transitive.
  - **Reflexive:** x=x is true for all values of x. so we can say that '=' is reflexive.
  - **Symmetric:** x=y and y=x is true for all values of x and y then we can say that '=' is symmetric.
  - **Transitive:** if x=y and y=z is true for all values then we can say that x=z. thus' =' is transitive.

(4) Quantifiers

- There are two basic quantifiers.
  - ***Universal Quantification:***
    P (a) is the preposition, if P (a) gives expected result for all values of a in the universe of discourse. The universal quantification of P (a) is denoted by, $\forall a\ P(a)$. So $\forall$ is called as for all i.e. it is the Universal Quantifier.

- ***Existential Quantification:***

  P (a) is the preposition, if there exits an element a in the universe of discourse such that P (a) is giving expected result. Here the Existential Quantification of P (a) is represented by $\exists a\ P(a) and\ \exists$ called as for some, i.e. it is Existential Quantifier.

(5) Linear Inequalities and Linear Equations.

Inequalities

The term inequality is applied to any statement involving one of the symbols <, >, ≤, ≥.

Examples of inequalities are:

i. $x \geq 1$
ii. $x + y + 2z > 16$
iii. $p^2 + q^2 \leq 1/2$
iv. $a^2 + ab > 1$

Fundamental Properties of Inequalities

1. If a≤b and c is any real number, then a + c≤b + c.

   For example, -3≤-1 implies -3+4≤-1 + 4.

2. If a≤b and c is positive, then ac≤bc.

   For example, 2≤3 implies 2(4) ≤3(4).

3. If a≤b and c is negative, then ac≥bc.

   For example, 3≤9 implies 3(-2) ≥9(-2).

4. If a≤b and b≤c, then a≤ c.

   For example, -1/2≤2 and 2≤8/3 imply -1/2≤8/3.

Solution of Inequality

- By solution of the one variable inequality 2x + 3≤7 we mean any number which substituted for x yields a true statement.
- For example, 1 is a solution of 2x + 3≤7 since 2(1) + 3 = 5 and 5 is less than and equal to 7.
- By a solution of the two variable inequality x - y≤5 we mean any ordered pair of numbers which when substituted for x and y, respectively, yields a true statement.
- For example, (2, 1) is a solution of x - y≤5 because 2-1 = 1 and 1≤5.
- By a solution of the three variable inequalities 2x - y + z≥3 we means an ordered triple of number which when substituted for x, y and z respectively, yields a true statement.
- For example, (2, 0, 1) is a solution of 2x - y + z≤3.
- A solution of an inequality is said to satisfy the inequality. For example, (2, 1) is satisfy x - y≤5.

Linear Equations

<u>One Unknown</u>

- A linear equation in one unknown can always be stated into the standard form

ax = b

- Where x is an unknown and a and b are constants. If a is not equal to zero, this equation has a unique solution

$$x = b/a$$

Two Unknowns

- A linear equation in two unknown, x and y, can be put into the form

$$ax + by = c$$

- Where x and y are two unknowns and a, b, c are real numbers. Also, we assume that a and b are no zero.

Solution of Linear Equation

- A solution of the equation consists of a pair of number, $u = (k_1, k_2)$, which satisfies the equation $ax + by = c$.
- Mathematically speaking, a solution consists of $u = (k_1, k_2)$ such that $ak_1 + bk_2 = c$.
- Solution of the equation can be found by assigning arbitrary values to x and solving for y or assigning arbitrary values to y and solving for x.

Two Equations in the Two Unknowns

- A system of two linear equations in the two unknowns x and y is

$$a_1x + b_1x = c_1$$
$$a_2x + b_2x = c_2$$

- Where a_1, a_2, b_1, b_2 are not zero. A pair of numbers which satisfies both equations is called a simultaneous solution of the given equations or a solution of the system of equations.
- Geometrically, there are three cases of a simultaneous solution
  1. If the system has exactly one solution, the graph of the linear equations intersects in one point.
  2. If the system has no solutions, the graphs of the linear equations are parallel.
  3. If the system has an infinite number of solutions, the graphs of the linear equations coincide.
- The special cases (2) and (3) can only occur when the coefficient of x and y in the two linear equations are proportional.

$$\frac{a_1}{a_2} = \frac{b_1}{b_2}$$

OR $\frac{a_1}{a_2} = \frac{b_1}{b_2}$ => $a_1b_2 - a_2b_1 = 0$ => $\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = 0$

The system has no solution when $\frac{a_1}{a_2} = \frac{b_1}{b_2} \neq \frac{c_1}{c_2}$

- The solution to the following system can be obtained by the elimination process, whereby reduce the system to a single equation in only one unknown.

$$a_1x + b_1x = c_1$$
$$a_2x + b_2x = c_2$$

(6) Algorithm Analysis Measures and Design Techniques.

Algorithm Analysis Measures

- Measuring Space complexity
- Measuring Time complexity
- Input size
- Computing Best case, Average case, Worst case
- Computing order of growth of algorithm.

Algorithm Design Techniques

- Divide and Conquer
- Greedy Approach
- Dynamic Programming
- Branch and Bound
- Backtracking

Randomized Algorithm

(7) Explain why analysis of algorithms is important

- When we have a problem to solve, there may be several suitable algorithms available. We would obviously like to choose the best.
- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- Generally, by analyzing several candidate algorithms for a problem, a most efficient one can be easily identified.
- Analysis of algorithm is required to decide which of the several algorithms are preferable.
- There are two different approaches to analyze an algorithm.
  1. **Empirical (posteriori) approach to choose an algorithm:** Programming different competing techniques and trying them on various instances with the help of computer.
  2. **Theoretical (priori) approach to choose an algorithm:** Determining mathematically the quantity of resources needed by each algorithm as a function of the size of the instances considered. The resources of most interest are computing time (time complexity) and storage space (space complexity). The advantage is this approach does not depend on programmer, programming language or computer being used.

(8) The Efficient Algorithm

- Analysis of algorithm is required to measure the efficiency of algorithm.
- Only after determining the efficiency of various algorithms, you will be able to make a well informed decision for selecting the best algorithm to solve a particular problem.
- We will compare algorithms based on their execution time. Efficiency of an algorithm means how fast it runs.
- If we want to measure the amount of storage that an algorithm uses as a function of the size of the instances, there is a natural unit available Bit.
- On the other hand, is we want to measure the efficiency of an algorithm in terms of time it takes to

arrive at result, there is no obvious choice.

- This problem is solved by the **principle of invariance**, which states that two different implementations of the same algorithm will not differ in efficiency by more than some multiplicative constant.
- Suppose that the time taken by an algorithm to solve an instance of size n is never more than cn seconds, where c is some suitable constant.
- Practically size of instance means any integer that in some way measures the number of components in an instance.
- Sorting problem: size is no. of items to be sorted.
- Graph: size is no. of nodes or edges or both involved.
- We say that the algorithm takes a time in the order of n i.e. it is a **linear time algorithm.**
- If an algorithm never takes more than cn^2 seconds to solve an instance of size n, we say it takes time in the order of cn^2 i.e. **quadratic time algorithm**.
- Some algorithms behave as **Polynomial**: (2^n or n^k) , **Exponential** : (c^n or n!), **Cubic**: (n^3 or $5n^3+n^2$), **Logarithmic:** (log n or n log n)

(9) Worst Case, Best Case & Average Case Complexity.

Worst Case Analysis

- In the worst case analysis, we calculate upper bound on running time of an algorithm.
- We must know the case that causes maximum number of operations to be executed.
- e.g. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array.
- When x is not present, the search () functions compares it with all the elements of arr[] one by one.

Average Case Analysis

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
- Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.
- For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array).
- So we sum all the cases and divide the sum by (n+1)

Best Case Analysis

- In the best case analysis, we calculate lower bound on running time of an algorithm.
- We must know the case that causes minimum number of operations to be executed.
- In the linear search problem, the best case occurs when x is present at the first location.

(10) Amortized Analysis

- In Amortize analysis finds the time required to perform a sequence of data structure operations is average of overall operation performed.
- It guarantees the average time per operation over worst case performance.
- It is used to show the average cost of an operation.

There are three methods of amortized analysis

1) Aggregate Method

- In this method we computes worst case time T(n) for n operations.
- Amortized cost is T(n)/n per operation.
- It gives the average performance of each operation in the worst case.

Example:

Operation:	PUSH (X,S)	POP (S)	MULTI-POP (S,k)
Worst Case:	O(1)	O(1)	O(min(s,k)=O(n)

MULTI-POP (S,k)

```
1. while not STACK-EMPTY(S) & k ≠ 0   This algorithm removes
2.       do POP (S)                   k top elements or pops
3.       K=k-1                        entire stack
```

- Total cost of sequence of n pop operations is $O(n^2)$. The average cost is O(n)/n = O(1).

2) Accounting Method

- In accounting method we assign different charges to different operations.
- The amount we charge in operation is called “amortized cost”.
- When amortized cost > actual cost then difference is assigned to “Credit”.
- This credit can be used when amortized cost < actual cost.
- The total amortized cost of a sequence of operations must be an upper bound on the total cost of sequence. C*i* = actual cost of i^{th} operation , $\widehat{Ci}$ = amortized cost of i^{th} operation

$$\sum_{i=1}^{n} \widehat{Ci} \geq \sum_{i=1}^{n} Ci$$

Example:

Operation	Actual Cost	Amortized Cost
PUSH	1	2
POP	1	0
MULTI-POP	Min(S,k)	0

Amortized cost = O(n)

3) Potential Method

- It is similar to the accounting method.
- Instead of representing prepaid work as credit it represents prepaid work as “Potential Energy”.
- Suppose initial data structure is D_0 for n operations D_0, D_1, D_2… D_n be the data structures let C_1, C_2… C_n denotes actual cost.
- ϕ is potential function that maps Data structure Di to a real number ϕ (Di)

$$\hat{C}_i = C_i + \Phi(D_i) - \Phi(D_{i-1})$$

(11) Asymptotic Notations.

Asymptotic notation is used to describe the running time of an algorithm.
It shows order of growth of function.

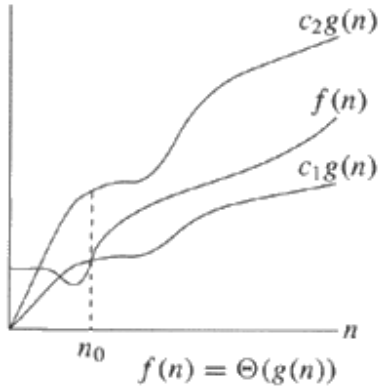
Θ-Notation (Same order)

- For a given function g(n), we denote by **Θ(*g*(*n*))** the set of functions

Θ(*g*(*n*)) = { f(n) : there exist positive constants c_1, c_2 and n_0 such that

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$ }

- Because **Θ(*g*(*n*))** is a set, we could write f(n) € **Θ(*g*(*n*))** to indicate that f(n) is a member of **Θ(*g*(*n*)).**
- This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.
- Figure a gives an intuitive picture of functions f(n) and g(n). For all values of n to the right of n_0, the value of f(n) lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \ge n_0$, the value of f(n) is equal to g(n) to within a constant factor.
- We say that g(n) is an asymptotically tight bound for f(n).



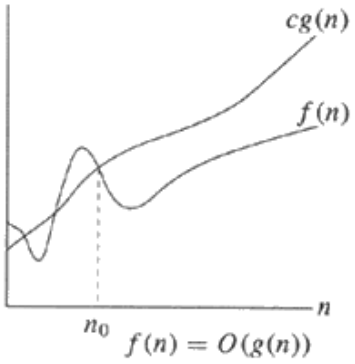
O-Notation (Upper Bound)

- For a given function g(n), we denote by **O(*g*(*n*))** the set of functions

O(*g*(*n*)) = { f(n) : there exist positive constants c and n_0 such that

$0 \le f(n) \le cg(n)$ for all $n \ge n_0$ }

- We use O notation to give an upper bound on a function, to within a constant factor. For all values of n to the right of n_0, the value of the function f(n) is on or below g(n).
- This notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.
- We say that g(n) is an asymptotically upper bound for f(n).



Example:

Let $f(n)=n^2$ and $g(n)=2^n$

n	$f(n)=n^2$	$g(n)=2^n$	
1	1	2	f(n) < g(n)
2	4	4	f(n) = g(n)
3	9	8	f(n) > g(n)
4	16	16	f(n) = g(n)
5	25	32	f(n) < g(n)
6	36	64	f(n) < g(n)
7	49	128	f(n) < g(n)

Here for n ≥ 4 we have behavior f (n) ≤ g(n)

Where $n_0=4$

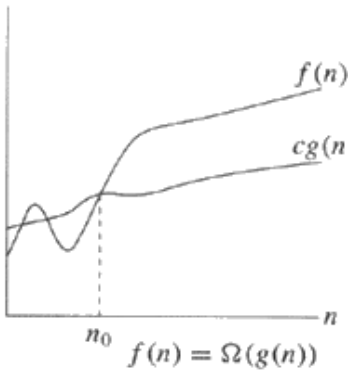
Ω-Notation (Lower Bound)

- For a given function g(n), we denote by **Ω(*g*(*n*))** the set of functions

Ω (*g*(*n*)) = { f(n) : there exist positive constants c and n_0 such that

0 ≤ cg(n) ≤ f (n)for all n ≥ n_0 }

- **Ω Notation provides an asymptotic lower bound.** For all values of n to the right of n_0, the value of the function f(n) is on or above cg(n).
- This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.



Example:

Let $f(n)= 2^n$ and $g(n)= n^2$

n	$f(n)= 2^n$	$g(n)= n^2$	
1	2	1	f(n) > g(n)
2	4	4	f(n) = g(n)
3	8	9	f(n) < g(n)
4	16	16	f(n) = g(n)
5	32	25	f(n) > g(n)
6	64	36	f(n) > g(n)
7	128	49	f(n) > g(n)

Here for $n \geq 4$ we have behavior $f(n) \geq g(n)$

Where $n_0=4$

(12) Analyzing Control Statement

Example: 1

```
C1:     b = a * c
```

Here cost of C1= O(1) as it executes once

Example: 2

```
for i=1 to n    C1: n+1 [executed n time + 1 time to check wrong
                condition]
    b = a * c   C2: n [executed n time]
```

T(n) =C1+C2
=n+1+n = 2n+1
=O(n)

Example: 3

```
for i=1 to n                    C1: n+1
   for j=1 to n    C2: n+1   }
       b = a * c   C3: n     }  n
   end
end
```

T(n) =C1+C2 +C3
=n+1+n(n+1)+n(n) = $2n^2$+2n+1
=O(n^2)

(13) Insertion sort

- Insertion Sort works by inserting an element into its appropriate position during each iteration.
- Insertion sort works by comparing an element to all its previous elements until an appropriate position is found.
- Whenever an appropriate position is found, the element is inserted there by shifting down remaining elements.

Algorithm

Procedure insert (T[1….*n*])	**cost**	**times**
for i ← 2 **to** *n* **do**	C1	n
x ← T[i]	C2	n-1
j ← i - 1	C3	n-1
while j > 0 **and** x < T[j] **do**	C4	$\sum_{i=2}^{n}$ Tj
T[j+1] ← T[j]	C5	$\sum_{i=2}^{n}$ Tj-1
j ← j - 1	C6	$\sum_{i=2}^{n}$ Tj-1
end		
T[j + 1] ← x	C7	n-1
end		

Analysis

- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
- A constant amount of time is required to execute each line of our pseudo code. One line may take a different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i, where c_i is a constant.
- The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and is executed n times will contribute $c_i n$ to the total running time.
- Let the time complexity of selection sort is given as T(n), then

  $T(n) = C_1n + C_2(n-1) + C_3(n-1) + C_4(\sum_{i=2}^{n} Tj) + (C_5 + C_6) \sum_{i=2}^{n} Tj - 1 + C_7(n-1)$.

  Where,

$$\sum_{i=2}^{n} Tj = \sum_{i=2}^{n} n - i$$

Best case:

Take $j = 1$

$T(n) = C_1n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5(n-1) + c_7(n-1)$

$= (C_1+C_2+C_3+C_4+C_7)\, n - (C_2+C_3+C_4+C_7)$

$= an - b$

Thus, $T(n) = \Theta(n)$

Worst case: Take j =n

T (n) = $C_1n+ C_2(n-1)+ C_3(n-1)+ C_4(\sum_{i=2}^{4} Tj)+(C_5 +C_6) \sum_{i=2}^{4} Tj - 1+ C_7(n-1)$.

= $C_1n+ C_2n+ C_3n+ C_4\frac{n}{2}+C_5\frac{n}{2}+ C_6\frac{n}{2}+ C_4\frac{n\^2}{2}+ C_5\frac{n\^2}{2}+ C_6\frac{n\^2}{2}+ C_7n-C_2-C_3- C_7$.

= $n^2(C_4\frac{1}{2}+ C_5\frac{1}{2}+ C_6\frac{1}{2})+n(C_1 +C_2 +C_3 +C_7 + C_4\frac{1}{2}+ C_5\frac{1}{2}+ C_6\frac{1}{2})-1(C_2 +C_3 +C_7)$.

= $an^2 + bn + c$.

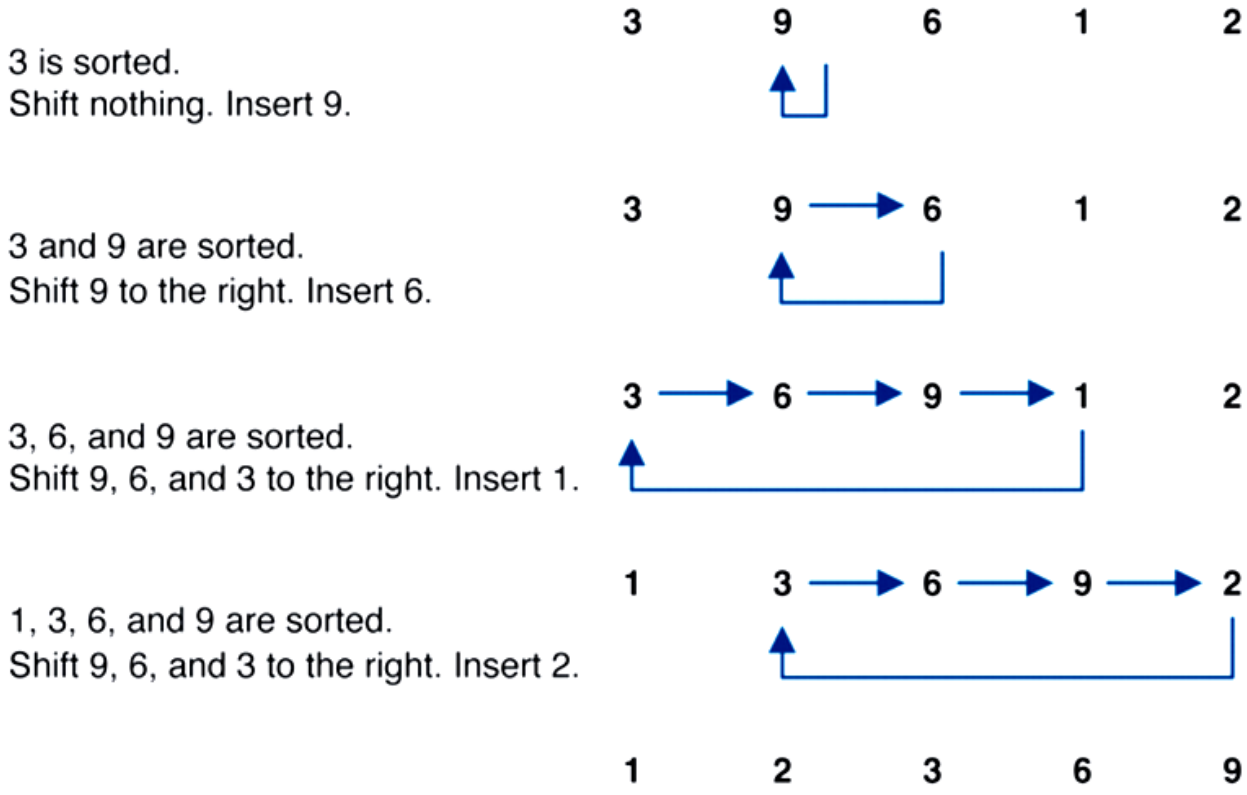
Thus, T(n) = $\Theta(n^2)$

Average case: Average case will be same as worst case T(n) = $\Theta(n^2)$

Time complexity of insertion sort

Best case	Average case	Worst case
O(n)	O (n^2)	O (n^2)

Example:



(14) Loop Invariant and the correctness of algorithm

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

1) **Initialization**: It is true prior to the first iteration of the loop.
2) **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
3) **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

- When the first two properties hold, the loop invariant is true prior to every iteration of the loop.
- Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step.
- Here, showing that the invariant holds before the first iteration is like the base case, and showing that the invariant holds from iteration to iteration is like the inductive step.
- The third property is perhaps the most important one, since we are using the loop invariant to show correctness.
- It also differs from the usual use of mathematical induction, in which the inductive step is used infinitely; here, we stop the "induction" when the loop terminates.

(15) Bubble sort

- In the bubble sort, the consecutive elements of the table are compared and if the keys of the two elements are not found in proper order, they are interchanged.
- It starts from the beginning of the table and continue till the end of the table. As a result of this the element with the largest key will be pushed to the last element's position.
- After this the second pass is made. The second pass is exactly like the first one except that this time the elements except the last are considered. After the second pass, the next largest element will be pushed down to the next to last position.

Algorithm

	cost	times
Procedure bubble (T[1….*n*])		
for i ← 1 **to** n **do**	C1	n+1
for i ← 1 **to** n-i **do**	C2	$\sum_{i=1}^{n}(n+1-i)$
if T[i] > T[j]	C3	$\sum_{i=1}^{n}(n-i)$
T[i] ↔ T[j]	C4	$\sum_{i=1}^{n}(n-i)$
end		
end		

Analysis

$$T(n)=C1\,(n+1) + C2\ \sum_{i=1}^{n}(n+1-i) + C3\ \sum_{i=1}^{n}(n-i) + C4\ \sum_{i=1}^{n}(n-i)$$

Best case:

Take i = 1

$T(n) = C_1n + C_1 + C_2n + C_3n - C_3 + C_4 n - C_4$

$= (C_1+C_2+C_3+C_4)\, n - (C_2,C_3,C_4,C_7)$

$= an - b$ Thus, $T(n) = \Theta(n)$

Worst Case:

$= C_1n + C_1 + C_2 n + C_2 - C_2\left(\frac{n(n+1)}{2}\right) + C_3n - C_3\left(\frac{n(n+1)}{2}\right) + C_4n - C_4\left(\frac{n(n+1)}{2}\right)$

$= [-C_2/n^2 - C_3/n^2 - C_4/n^2] + [-C_2/n - C_3/n - C_4/n] + C_1 + C_2 + C_3 + C_4$

$= an^2 + bn + c$

$= \Theta(n^2)$

Average case: Average case will be same as worst case $T(n) = \Theta(n^2)$

Example:

Consider the following numbers are stored in an array:

Original Array: 32,51,27,85,66,23,13,57

Pass 1 : 32,27,51,66,23,13,57,85

Pass 2 : 27,33,51,23,13,57,66,85

Pass 3 : 27,33,23,13,51,57,66,85

Pass 4 : 27,23,13,33,51,57,66,85

Pass 5 : 23,13,27,33,51,57,66,85

Pass 6 : 13,23,27,33,51,57,66,85

(16) Selection sort

- Selection Sort works by repeatedly selecting elements.
- The algorithm finds the smallest element in the array first and exchanges it with the element in the first position.

Then it finds the second smallest element and exchanges it with the element in the second position and continues in this way until the entire array is sorted.

Algorithm

Procedure select (T [1….n])	**Cost**	**times**
for i←1 **to** n-1 **do**	C1	n
minj ← i ; minx ← T[i]	C2	n-1
for j←i+1 **to** n **do**	C3	$\sum_{i=1}^{n-1}(i+1)$
if T[j] < minx **then** minj ← j	C4	$\sum_{i=1}^{n-1} i$
minx ← T[j]	C5	$\sum_{i=1}^{n-1} i$
T[minj] ← T[i]	C6	n-1
T[i] ← minx	C7	n-1

Analysis

$T(n)=C_1n + C_2(n-1) + C_3(\sum_{i=1}^{n-1}(i+1)) + C_4(\sum_{i=1}^{n-1}(i)) + C_5(\sum_{i=1}^{n-1}(i)) + C_6(n-1) + C_7(n-1)$

$= C_1n + C_2n + C_6n + C_7n + C_3\frac{n}{2} + C_4\frac{n}{2} + C_5\frac{n}{2} + C_3\frac{n\hat{}2}{2} + C_4\frac{n\hat{}2}{2} + C_5\frac{n\hat{}2}{2} - C_2 - C_6 - C_7$

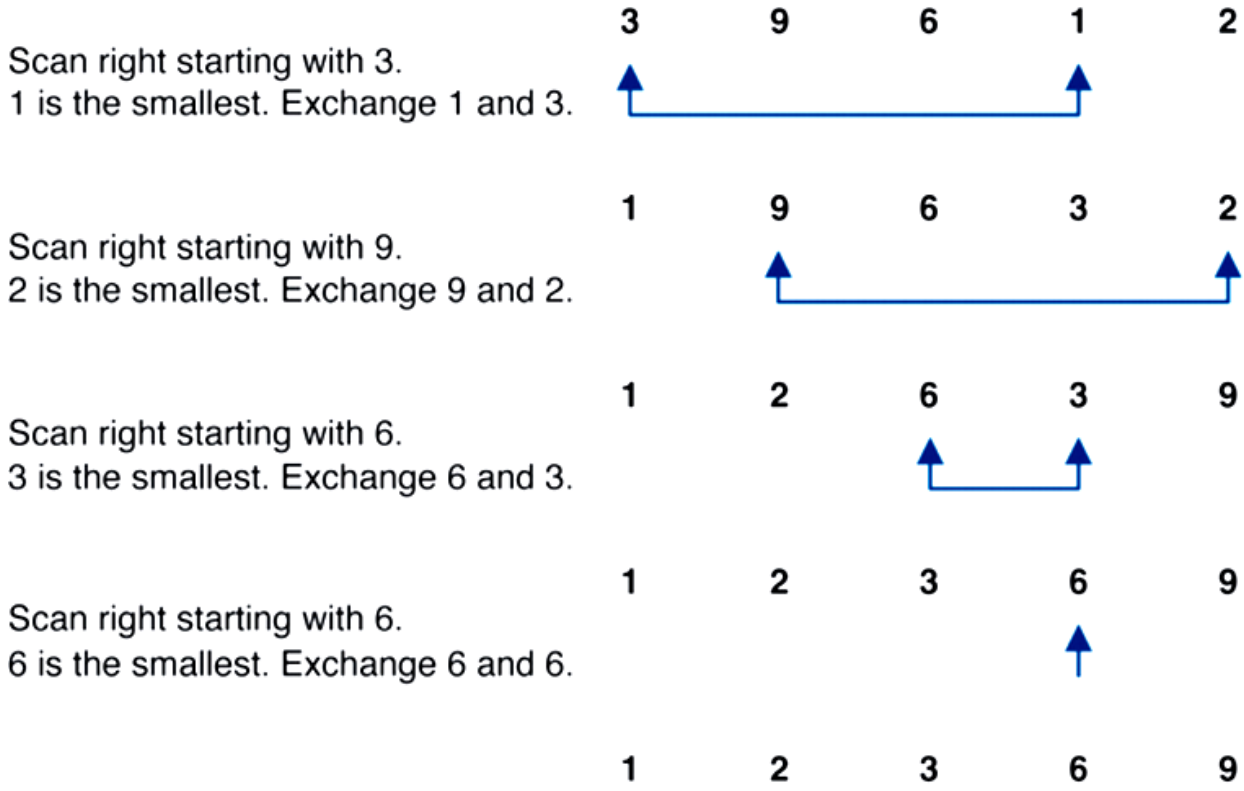
$= n(C_1 + C_2 + C_6 + C_7 + C_3\frac{1}{2} + C_4\frac{1}{2} + C_5\frac{1}{2}) + n^2(C_3\frac{1}{2} + C_4\frac{1}{2} + C_5\frac{1}{2}) - 1(C_2 + C_6 + C_7)$

$= an^2 + bn + c$

Time complexity of insertion sort

Best case	Average case	Worst case
$O(n^2)$	$O(n^2)$	$O(n^2)$

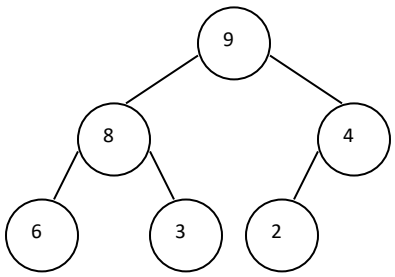
Example:



(17) Heap

- A heap data structure is a binary tree with the following properties.
  1. It is a complete binary tree; that is each level of the tree is completely filled, except possibly the bottom level. At this level it is filled from left to right.
  2. It satisfies the heap order property; the data item stored in each node is greater than or equal to the data item stored in its children node.

Example:

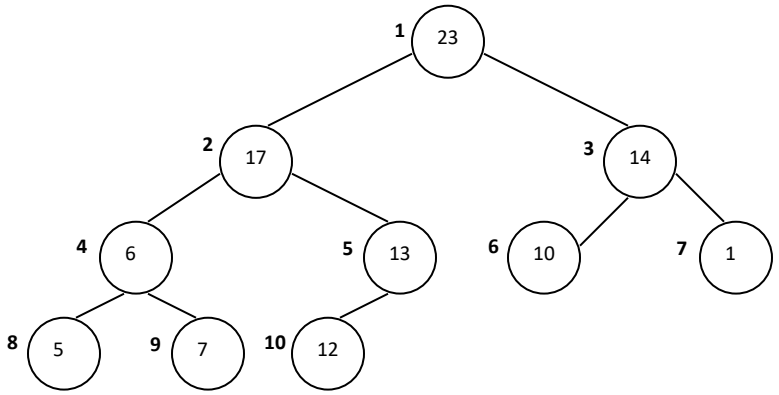


- Heap can be implemented using an array or a linked list structure. It is easier to implement heaps using arrays.

- We simply number the nodes in the heap from top to bottom, numbering the nodes on each level from left to right and store the i^{th} node in the i^{th} location of the array.
- An array *A* that represents a heap is an object with two attributes:
  i. *length*[*A*], which is the number of elements in the array, and
  ii. *heap-size*[*A*], the number of elements in the heap stored within array *A*.
- The root of the tree is *A*[1], and given the index *i* of a node, the indices of its parent PARENT(*i*), left child LEFT(*i*), and right child RIGHT(*i*) can be computed simply:

```
PARENT(i)
    return ⌊i/2⌋
LEFT(i)
    return 2i
RIGHT(i)
    return 2i + 1
```

Example:



- The array form for the above heap is,

23	17	14	6	13	10	1	5	7	12

- There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***.
- In a ***max-heap***, the ***max-heap property*** is that for every node *i* other than the root, ***A*[PARENT(*i*)] ≥ *A*[*i*] ,**
- That is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the sub-tree rooted at a node contains values no larger than that contained at the node itself.
- A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node *i* other than the root, ***A*[PARENT(*i*)] ≤ *A*[*i*] .**
- The smallest element in a min-heap is at the root.
- For the heap-sort algorithm, we use max-heaps. Min-heaps are commonly used in priority Queues.

- Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the Heap to be the height of its root.
- Height of an n element heap based on a binary tree is lg *n*
- The basic operations on heap run in time at most proportional to the height of the tree and thus take *O*(lg *n*) time.
- Since a heap of *n* elements is a complete binary tree which has height k; that is one node on level k, two nodes on level k-1 and so on…
- There will be 2^{k-1} nodes on level 1 and at least 1 and not more than 2^k nodes on level 0.

Building a heap

- For the general case of converting a complete binary tree to a heap, we begin at the last node that is not a leaf; apply the "percolate down" routine to convert the subtree rooted at this current root node to a heap.
- We then move onto the preceding node and percolate down that subtree.
- We continue on in this manner, working up the tree until we reach the root of the given tree.
- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array *A*[1,…,*n*], where *n* = *length*[*A*], into a max-heap.
- The elements in the sub-array *A*[(⌊*n*/2⌋+1),…, *n*] are all leaves of the tree, and so each is a 1-element heap to begin with.
- The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAXHEAPIFY on each one.

Algorithm

```
BUILD-MAX-HEAP(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
            do MAX-HEAPIFY(A, i)
```

Analysis

- Each call to Heapify costs O(lg *n*) time, and there are O(*n*) such calls. Thus, the running time is at most O(*n* lg n)

Maintaining the heap property

- One of the most basic heap operations is converting a complete binary tree to a heap. Such an operation is called Heapify.
- Its inputs are an array *A* and an index *i* into the array. When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but that *A*[*i*] may be smaller than its children, thus violating the max-heap property.

- The function of MAX-HEAPIFY is to let the value at *A*[*i*] "float down" in the max-heap so that the sub-tree rooted at index *i* becomes a max-heap.

Algorithm

MAX-HEAPIFY(*A*, *i*)
l ← LEFT(*i*)
r ← RIGHT(*i*)
if *l* ≤ *heap-size*[*A*] and *A*[*l*] > *A*[*i*]
    then *largest* ← *l*
else *largest* ← *i*
if *r* ≤ *heap-size*[*A*] and *A*[*r*] > *A*[*largest*]
    then *largest* ← *r*
if *largest* ≠ *i*
    then exchange *A*[*i*] ↔ *A*[*largest*]
MAX-HEAPIFY(*A*, *largest*)

- At each step, the largest of the elements *A*[*i*], *A*[LEFT(*i*)], and *A*[RIGHT(*i*)] is determined, and its index is stored in *largest*.
- If *A*[*i*] is largest, then the sub-tree rooted at node *i* is a max-heap and the procedure terminates.
- Otherwise, one of the two children has the largest element, and *A*[*i*] is swapped with *A*[*largest*], which causes node *i* and its children to satisfy the max-heap property.
- The node indexed by *largest*, however, now has the original value *A*[*i*], and thus the sub-tree rooted at *largest* may violate the max-heap property. therefore, MAX-HEAPIFY must be called recursively on that sub-tree.

Analysis

- The running time of MAX-HEAPIFY on a sub-tree of size *n* rooted at given node *i* is the Θ(1) time to fix up the relationships among the elements *A*[*i*], *A*[LEFT(*i*)], and *A*[RIGHT(*i*)], plus the time to run MAX-HEAPIFY on a sub-tree rooted at one of the children of node *i*.
- The children's sub-trees can have size of at most 2*n*/3 and the running time of MAX-HEAPIFY can therefore be described by the recurrence

$$T(n) \le T(2n/3) + \Theta(1).$$

- The solution to this recurrence is $T(n) = O(\lg n)$.

Heap sort

- The heap sort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array *A*[1,…, *n*], where *n* = *length*[*A*].
- Since the maximum element of the array is stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*].
- If we now "discard" node *n* from the heap (by decrementing *heap-size*[*A*]), we observe that *A*[1,…, (*n* - 1)] can easily be made into a max-heap.
- The children of the root remain max-heaps, but the new root element may violate the max-heap property.

- All that is needed to restore the max heap property, however, is one call to MAX-HEAPIFY(*A*, 1), which leaves a max-heap in *A*[1,…, (*n* - 1)].
- The heap sort algorithm then repeats this process for the max-heap of size *n* – 1 down to a heap of size 2.

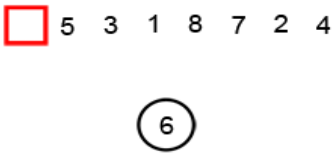
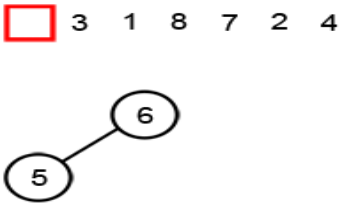
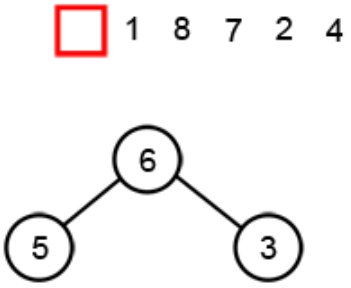
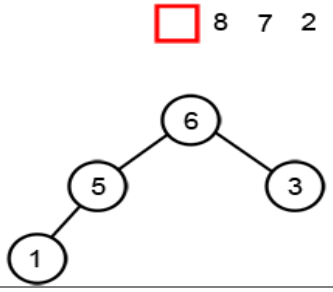
Algorithm

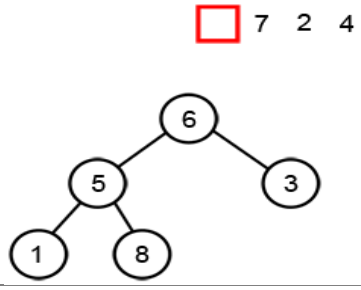
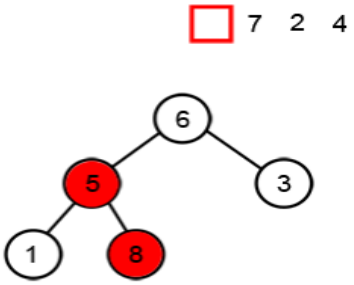
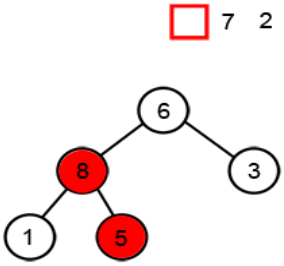
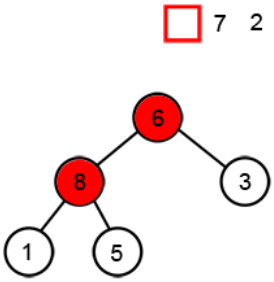
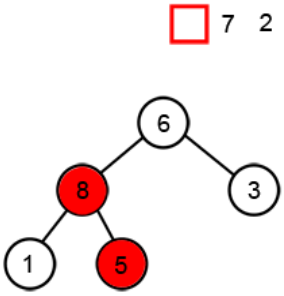
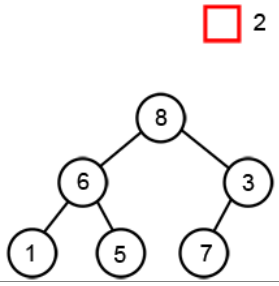
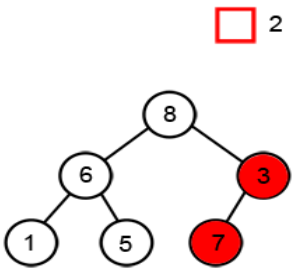
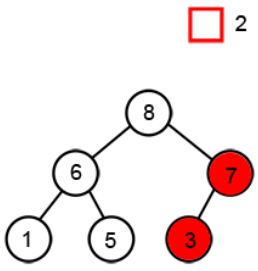
```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i ← length[A] downto 2
                do exchange A[1] ↔ A[i]
                heap-size[A] ← heap-size[A] - 1
                MAX-HEAPIFY(A, 1)
```

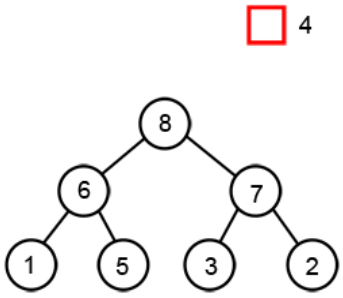
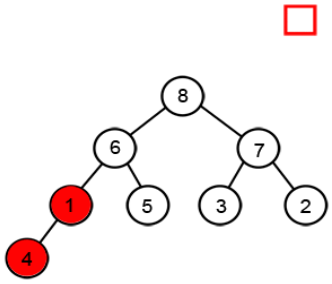
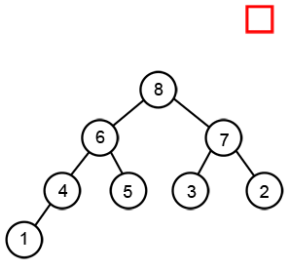
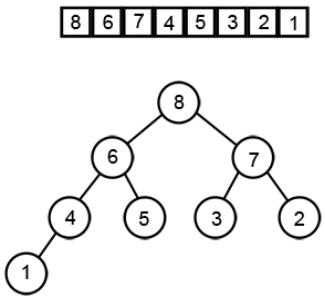
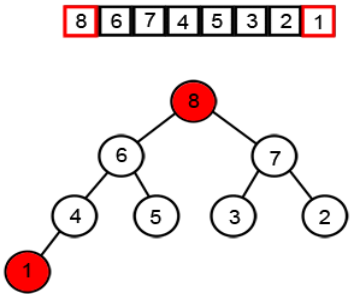
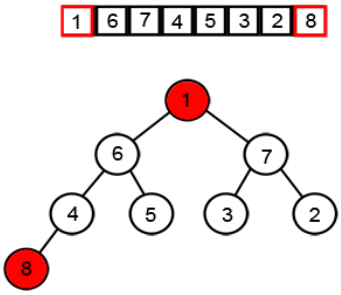
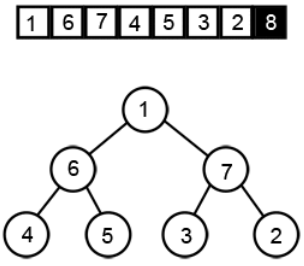
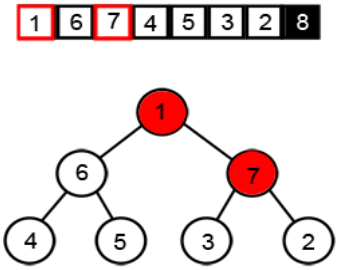
- The HEAPSORT procedure takes time *O*(*n* lg *n*), since the call to BUILD-MAX-HEAP takes time *O*(*n*) and each of the *n* - 1 calls to MAX-HEAPIFY takes time *O*(lg *n*).

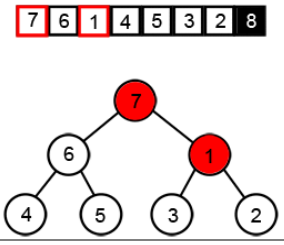
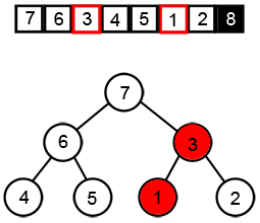
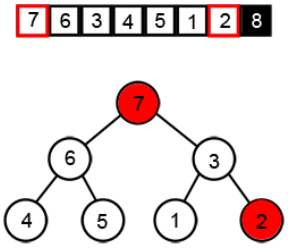
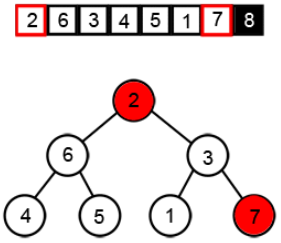
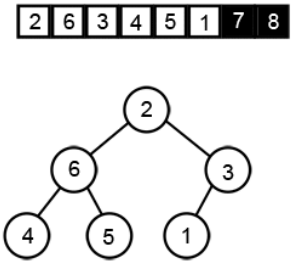
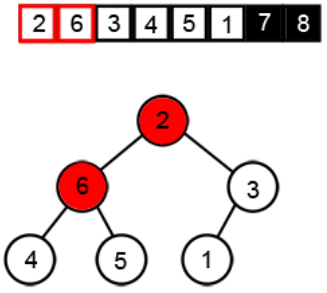
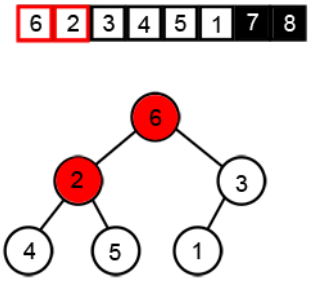
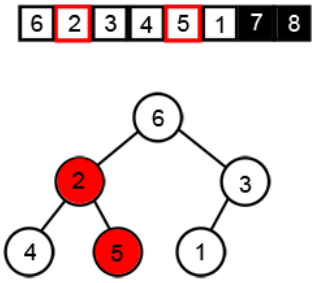
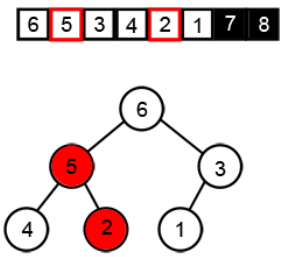
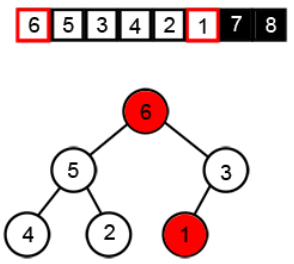
Example

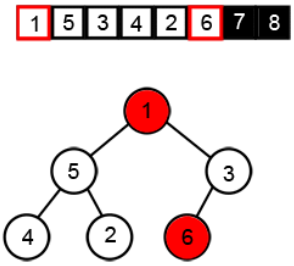
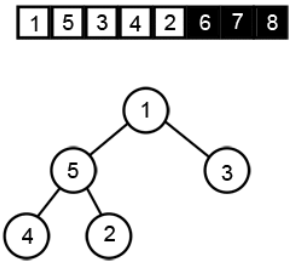
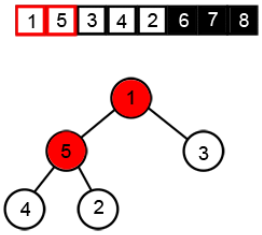
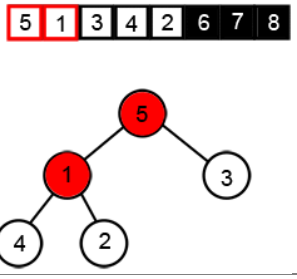
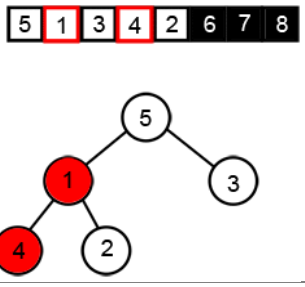
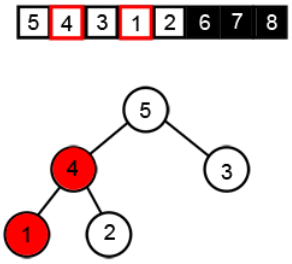
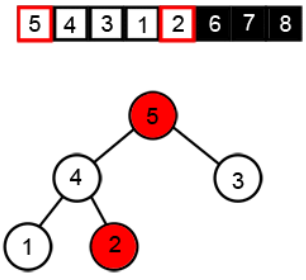
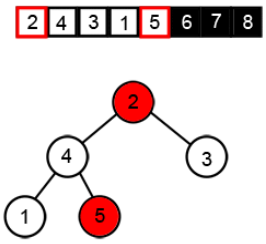
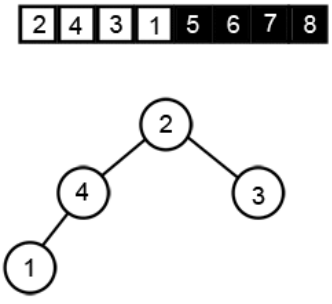
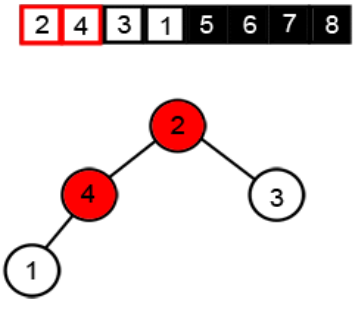
Here is the array: 6, 5, 3, 1, 8, 7, 2, 4

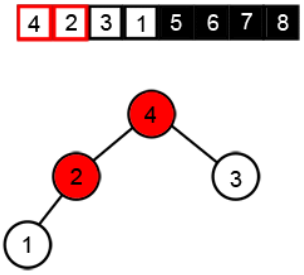
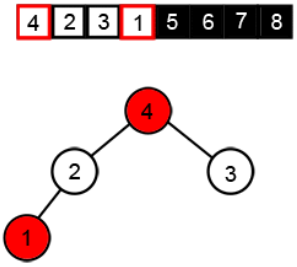
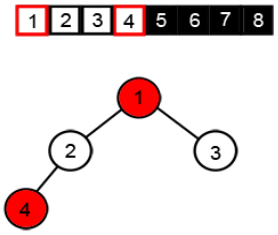
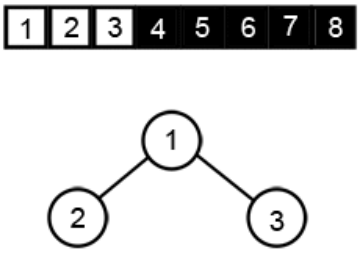
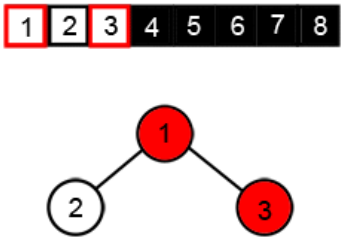
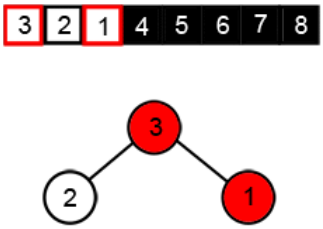
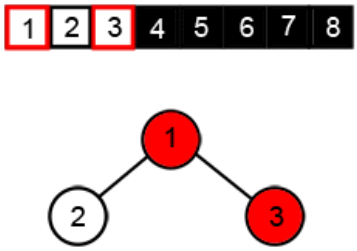
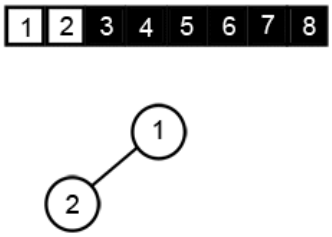
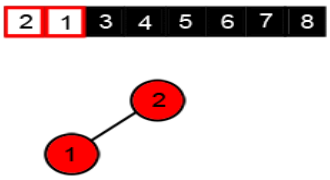
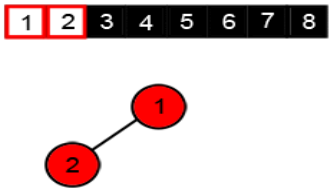
Creating Max Heap. Inserting Node 6 in left to right top to bottom manner 	Inserting Node 5 in left to right top to bottom manner 
Inserting Node 3 in left to right top to bottom manner 	Inserting Node 7 in left to right top to bottom manner 

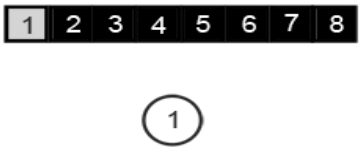
Inserting Node 8 in left to right top to bottom manner 	Heapify node 8 
Heapify node 8 and swap with node 5 	Heapify node 8 
Heapify node 8 and swap with node 6 	Inserting Node 7 in left to right top to bottom manner 
Heapify node 7 	Heapify node 7 and swap with node 3. 

Inserting Node 2 in left to right top to bottom manner 	Inserting Node 4 in left to right top to bottom manner 
Heapify node 4 	Array representation of MAX heap 
Select first node and swap with last node 	Select first node 8 and swap with last node 1 
Decrease size of heap by one 	Heapify new array 

Heapify new array swap node 7 with node 1. 	Heapify new array swap node 7 with node 1. 
Select first node and swap with last node 	Select first node and swap with last node 
Decrease size of heap by one 	Heapify node 2 and node 6 
Swap node 6 and node 2 	Heapify node 5 and node 2 
Swap node 5 with node 2 	Select first node 6 and last node 1 

Swap first node 6 with last node 1 	Decrease size of heap by one 
Heapify node 5 and node 1 	Swap node 5 with node 1 
Heapify node 1 with node 4 	Swap node 4 with node 1 
Select first node 5 and last node 2 	Swap first node 5 and last node 2 
Decrease size of heap by one 	Heapify node 4 and node 2 

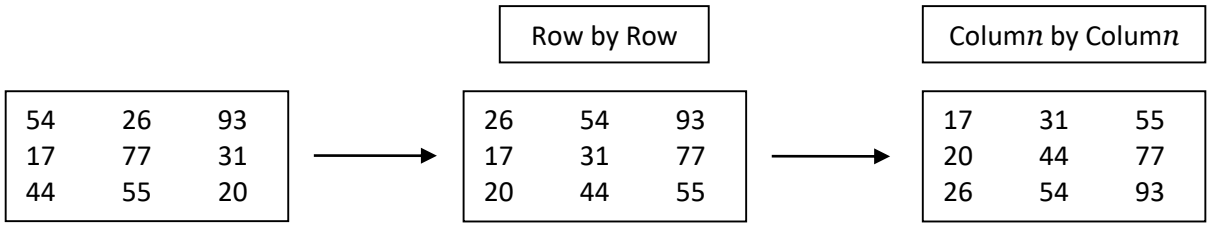
Swap node 4 with node 2 	Select first node 4 and last node 1 
Swap first node 4 with last node 1 	Decrease size of heap by one 
Heapify node 1 and node 3 	Swap node 3 with node 1 
Select first node 3 and last node 1 and swap it 	Decrease size of heap by one 
Heapify node 2 and node 1 	Select first and last element and swap it 

Decrease size of heap by one	Sorted array
	

(18) Shell sort

- Shell sort works by comparing elements that are distant rather than adjacent elements in an array.
- Shell sort uses a sequence h1, h2… ht called the increment sequence. Any increment sequence is fine as long as h1 = 1 and some other choices are better than others.
- Shell sort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.

Example:1 54,26,93,17,77,31,44,55,20



Sorted array: 17,20,26,31,44,54,55,77,93

Example:2 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2

It is arranged in an array with 7 columns (left), then the columns are sorted (right):

Step-1

3 7 9 0 5 1 6 → 3 3 2 0 5 1 5
8 4 2 0 6 1 5 → 7 4 4 0 6 1 6
7 3 4 9 8 2 → 8 7 9 9 8 2

Step-2

3 3 2 → 0 0 1
0 5 1 → 1 2 2
5 7 4 → 3 3 4
4 0 6 → 4 5 6
1 6 8 → 5 6 8
7 9 9 → 7 7 9
8 2 → 8 9

(19) Bucket sort

- Bucket sort runs in linear time when the input is drawn from a uniform distribution.
- Like counting sort, bucket sort is fast because it assumes something about the input.
- Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1)

Assumption: the keys are in the range [0, N)

Basic idea:

1. Create N linked lists (buckets) to divide interval [0,N) into subintervals of size 1
2. Add each input element to appropriate bucket
3. Concatenate the buckets

Expected total time is O(n + N), with n = size of original sequence if N is O(n) -> sorting algorithm in O(n) !

Algorithm

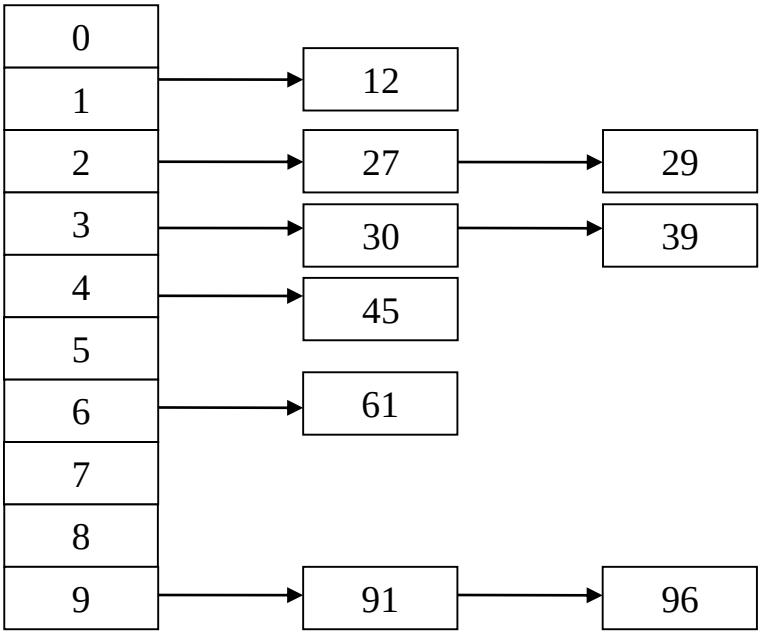
```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3     do insert A[i] into list B[⌊n A[i]⌋]
4  for i ← 0 to n - 1
5     do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . ., B[n - 1]  together in
    order
```

***Example*:** 45,96,29,30,27,12,39,61,91

Step-1 Add keys one by one in appropriate bucket queue as shown in figure

Step-2 sort each bucket queue with insertion sort

Step-3 Merge all bucket queues together in order



After sorting : 12,27,29, 30, 39, 45, 61, 91, 96

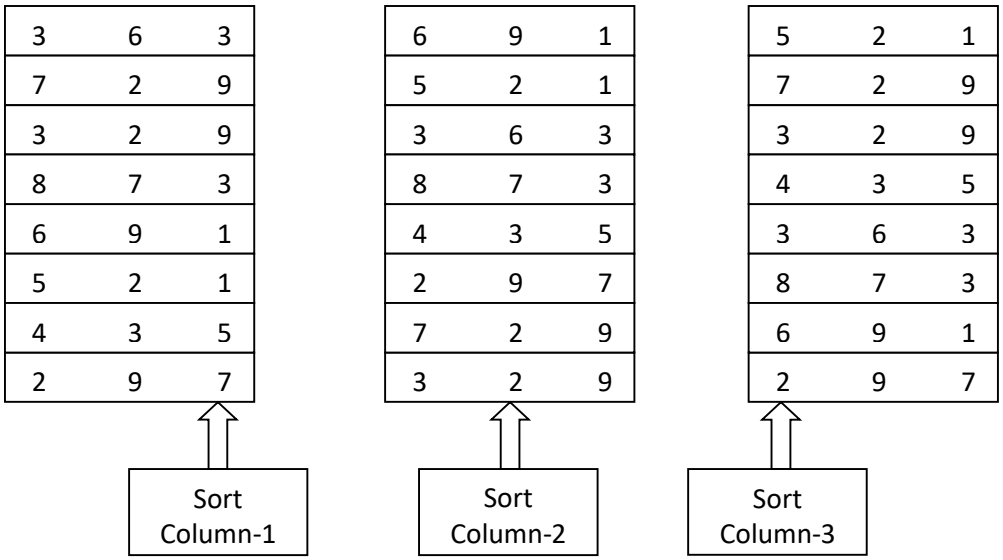
(20) Radix sort

- The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions.
- Shading indicates the digit position sorted on to produce each list from the previous one.
- The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest order digit.
- Given n d-digit numbers in which each digit can take on up to k possible values, RADIXSORT correctly sorts these numbers in Θ(d(n+ k)) time.

Algorithm

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2      do use a stable sort to sort array A on digit i
```

***Example*:** 363, 729, 329, 873, 691, 521, 435, 297



2	9	7
3	2	9
3	6	3
4	3	5
5	2	1
6	9	1
7	2	9
8	7	3

(21) Counting sort

- Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.
- When k = O(n), the sort runs in Θ(n) time.
- The basic idea of counting sort is to determine, for each input element x, the number of elements less than x.
- This information can be used to place element x directly into its position in the output array.

Algorithm

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2     do C[i] ← 0
 3  for j ← 1 to length[A]
 4     do C[A[j]] ← C[A[j]] + 1
 5 //C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7     do C[i] ← C[i] + C[i - 1]
 8 //C[i] now contains the number of elements less than or equal to
    i.
 9  for j ← length[A] down to 1
10     do B[C[A[j]]] ← A[j]
11        C[A[j]] ← C[A[j]] – 1
```

Example

Step-1 Given input array A[1…8]

1	2	3	4	5	6	7	8
3	6	4	1	3	4	1	4

Step-2 Determine size of array C as maximum value in array A, here its ‘6’
Initialize all elements intermediate array C[1…6] = 0

1	2	3	4	5	6
0	0	0	0	0	0

Step-3 Update array C with occurrences of each value of array A

1	2	3	4	5	6
2	0	2	3	0	1

Step-4 In array C from index 2 to n add value with previous element

1	2	3	4	5	6
2	2	4	7	7	8

Create output array B[1…8]
Start positioning elements of Array A to B shown as follows

Step-5 Array A

1	2	3	4	5	6	7	8
3	6	4	1	3	4	1	4

Array C

1	2	3	4	5	6
2	2	4	7	7	8

Array B

1	2	3	4	5	6	7	8
						4	

Repeat above from element n to 1 we can have

Array B

1	2	3	4	5	6	7	8
1	1	3	3	4	4	4	0

(1) Divide & Conquer Technique and the general template for it.

Divide and conquer technique

- Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub problems.
- These algorithms typically follow a ***divide-and-conquer*** approach:
- The divide-and-conquer approach involves three steps at each level of the recursion:
- **Divide:** Break the problem into several sub problems that are similar to the original problem but smaller in size,
- **Conquer:** Solve the sub problems recursively, and If the sub problem sizes are small enough, just solve the sub problems in a straightforward manner.
- **Combine:** Combine these solutions to create a solution to the original problem.

The general template

- Consider an arbitrary problem, and let adhoc be a simple algorithm capable of solving the problem.
- We call adhoc as basic sub-algorithm such that it can be efficient in solving small instances, but its performance on large instances is of no concern.
- The general template for divide-and-conquer algorithms is as follows.

function DC(x)

__if__ x is sufficiently small or simple __then return__ adhoc(x)
decompose x into smaller instances x_1 , x_2, …, x_l
__for__ i = 1 to l __do__ yi ← DC(x_i)
recombine the Y_L 's to obtain a solution y for x
__return__ y

- The running-time analysis of such divide-and-conquer algorithms is almost automatic.
- Let *g(n)* be the time required by *DC* on instances of size *n,* not counting the time needed for the recursive calls.
- The total time t*(n)* taken by this divide-and-conquer algorithm is given by Recurrence equation,

$$t(n) = lt(n \div b) + g(n)$$

Provided n is large enough.

- The solution of equation is given as, $if\ there\ exists\ an\ integer\ k\ such\ that\ g(n) \in\ \theta(n^k)$ *then*

$$t(n) \in \begin{cases} \theta(n^k) & if\ l < b^k \\ \theta(n^k logn) & if\ l = b^k \\ \theta\left(n^{\log_b l}\right) & if\ l > b^k \end{cases}$$

- The recurrence equation and its solution are applicable to find the time complexity of every problem which can be solved using Divide & Conquer Technique.

(2) Linear Search

Sequential (Linear) search algorithm

```
Function sequential ( T[1,…,n], x )
    for i = 1 to n do
        if T [i] ≥ x then return index i
    return n + 1
```

Analysis

- Here we look sequentially at each element of T until either we reach to end of the array or find a number no smaller than x.
- This algorithm clearly takes time in Θ (r), where r is the index returned. Θ (n) in worst case and O (1) in best case.

(3) Divide and Conquer Technique and the use of it for Binary Searching Method.

Binary Search Method

- Binary Search is an extremely well-known instance of divide-and-conquer approach.
- Let T[*1 . . . n*] be an array of increasing sorted order; that is $T[i] \leq T[j]$ whenever $1 \leq i \leq j \leq n$.
- Let x be some number. The problem consists of finding x in the array T if it is there.
- If x is not in the array, then we want to find the position where it might be inserted.

Binary Search Algorithm (Iterative)

- The basic idea of binary search is that for a given element we check out the middle element of the array.
- We continue in either the lower or upper segment of the array, depending on the outcome of the search until we reach the required (given) element.
- Here the technique Divide & Conquer applies. Total number of elements to be searched is divided in half size every time.

```
Function biniter ( T[1,…,n], x )
    i ← 1; j ← n
    while i < j do
        k ← (i + j ) ÷ 2
        if x ≤ T [k] then j ← k
                     else i ← k + 1
    return i
```

Analysis

- To analyze the running time of a while loop, we must find a function of the variables involved whose value decreases each time round the loop.
- Here it is ***j – i + 1***
- Which we call as d. d represents the number of elements of T still under consideration.

- Initially $d = n$.
- Loop terminates when $i \geq j$, which is equivalent to $d \leq 1$
- Each time round the loop there are three possibilities,
  I. Either j is set to k – 1
  II. Or i is set to k + 1
  III. Or both i and j are set to k
- Let d and d' stand for the value of j – i +1 before and after the iteration under consideration. Similarly i, j , i' and j'.
- Case I : if x < T [k]
  So, j ← k – 1 is executed.
  Thus i' = i and j' = k -1 where k = (i + j) ÷ 2
  Substituting the value of k, j' = [(i + j) ÷ 2] – 1
  d' = j' – i' + 1
  Substituting the value of j', d' = [(i + j) ÷ 2] – 1 – i + 1
  d' ≤ (i + j) /2 – i
  d' ≤ (j – i)/2 ≤ (j – i + 1) / 2
  d' ≤ d/2
- Case II : if x > T [k]
  So, i ← k + 1 is executed
  Thus i' = K + 1 and j' = j where k = (i + j) ÷ 2
  Substituting the value of k, i' = [(i + j) ÷ 2] + 1
  d' = j' – i' + 1
  Substituting the value of I', d' = j - [(i + j) ÷ 2] + 1 + 1
  d' ≤ j - (i + j -1) /2 ≤ (2j – i – j + 1) / 2 ≤ (j – i + 1) / 2
  d' ≤ d/2
- Case III : if x = T [k]
  i = j → d' = 1
- We conclude that whatever may be case, **d' ≤ d/2** which means that the value of d is at least getting half each time round the loop.
- Let d_k denote the value of j – i + 1 at the end of k^{th} trip around the loop. $d_0 = n$.
- We have already proved that $d_k = d_{k-1} / 2$
- For n integers, how many times does it need to cut in half before it reaches or goes below 1?
- $n / 2^k \leq 1 \rightarrow n \leq 2^k$
- ***k = lgn*** , search takes time.

The complexity of biniter is Θ (lg n).

Binary Search Algorithm (Recursive)

Function binsearch (T[1,…,n], x)
  if *n = 0* ***or*** *x > T[n]* ***then return*** *n + 1*
  else return *binrec (T[1,…,n], x)*
Function binrec(T[i,…,j], x)
  if *i = j* ***then return*** *i*

```
    k ← (i + j) ÷ 2
    if x ≤ T [k] then return binrec( T[i,…,k], x )
            else return binrec( T[k + 1,…,j], x )
```

Analysis

- Let t(n) be the time required for a call on binrec(T[i,…,j], x), where n = j – i + 1 is the number of elements still under consideration in the search.
- The recurrence equation is given as,
  t(n) = t(n/2) + Θ(1)
  Comparing this to the general template for divide and conquer algorithm, l = 1, b = 2 and k = 0.
  So, t(n) Є Θ(lg n)
- **The complexity of binrec is Θ (lg n).**

(4) Merge Sort and Analysis of Merge Sort

- The divide & conquer approach to sort n numbers using merge sort consists of separating the array T into two parts where sizes are almost same.
- These two parts are sorted by recursive calls and then merged the solution of each part while preserving the order.
- The algorithm considers two temporary arrays U and V into which the original array T is divided.
- When the number of elements to be sorted is small, a relatively simple algorithm is used.
- Merge sort procedure separates the instance into two half sized sub instances, solves them recursively and then combines the two sorted half arrays to obtain the solution to the original instance.

Algorithm for merging two sorted U and V arrays into array T

```
Procedure merge(U[1,…,m+1],V[1,…,n+1],T[1,…,m+n])
        i, j ← 1
        U[m+1], V[n+1] ← ∞
    for k ← 1 to m + n do
        if U[i] < V[j]
            then T[k] ← U[i] ; i ← i + 1
            else T[k] ← V[j] ; j ← j + 1
```

Algorithm merge sort

```
Procedure mergesort(T[1,…,n])
    if n is sufficiently small then insert(T)
    else
        array U[1,…,1+n/2],V[1,…,1+n/2]
        U[1,…,n/2] ← T[1,…,n/2]
        V[1,…,n/2] ← T[n/2+1,…,n]
```

mergesort(U[1,…,n/2])
mergesort(V[1,…,n/2])
merge(U, V, T)

Analysis

- Let T(n) be the time taken by this algorithm to sort an array of n elements.
- Separating T into U & V takes linear time; merge (U, V, T) also takes linear time.
- Now,

  $T(n)=T(n/2)+ T(n/2)+g(n)$ where $g(n) \in \Theta(n)$.

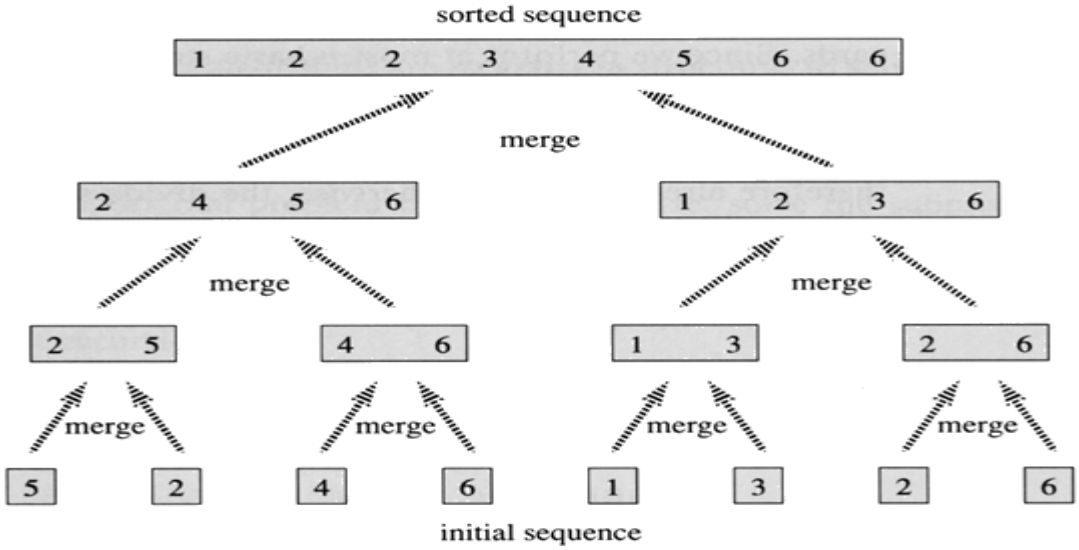
  $T(n) = 2t(n/2)+ \Theta (n)$

  Applying the general case, l=2, b=2, k=1

  Since $l = b^k$ the second case applies which yields $t(n) \in \Theta(nlogn)$.

Time complexity of merge sort is **Θ (nlogn).**

Example



(5) Quick sort method / algorithm and its complexity.

- Quick sort works by partitioning the array to be sorted.

- Each Partition is internally sorted recursively.
- As a first step, this algorithm chooses one element of an array as a pivot or a key element.
- The array is then partitioned on either side of the pivot.
- Elements are moved so that those greater than the pivot are shifted to its right whereas the others are shifted to its left.
- Two pointers low and up are initialized to the lower and upper bounds of the sub array.
- Up pointer will be decremented and low pointer will be incremented as per following condition.
    1. Increase low pointer until T[low] > pivot.
    2. Decrease up pointer until T[up] ≤ pivot.
    3. If low < up then interchange T[low] with T[up].
    4. If up ≤ low then interchange T[up] with T[i].

Algorithm

Procedure *pivot(T[i,…,j]; **var** l)*
{Permutes the elements in array T[i,…,j] and returns a value l such that, at the end, i<=l<=j, T[k]<=p for all i ≤ k < l, T[l]=p, And T[k] > p for all l < k ≤ j, where p is the initial value T[i]}
P ← T[i]
K ← i; l ← j+1
Repeat *k ← k+1* ***until*** *T[k] > p*
Repeat *l ← l-1* ***until*** *T[l] ≤ p*
While *k < l* ***do***
    Swap T[k] and T[l]
    Repeat *k ← k+1* ***until*** *T[k] > p*
    Repeat *l ← l-1* ***until*** *T[l] ≤ p*
Swap T[i] and T[l]

Procedure *quicksort(T[i,…,j])*
{Sorts subarray T[i,…,j] into non decreasing order}
if *j – i is sufficiently small* ***then*** *insert (T[i,…,j])*
else
pivot(T[i,…,j],l)
quicksort(T[i,…, l - 1])
quicksort(T[l+1,…,j]

Analysis

1. Worst Case

- Running time of quick sort depends on whether the partitioning is balanced or unbalanced.
- And this in turn depends on which element is chosen as key or pivot element.
- The worst case behavior for quick sort occurs when the partitioning routine produces one sub problem with n-1 elements and one with 0 elements.
- In this case recurrence will be,

    $T(n)=T(n-1)+T(0)+\Theta(n)$
    $T(n)=T(n-1)+\Theta(n)$
    $T(n)=\Theta(n^2)$

2. Best Case

- Occurs when partition produces sub problems each of size n/2.
- Recurrence equation:

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

  $l = 2, b = 2, k = 1$, so $l = b^k$

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

3. Average Case

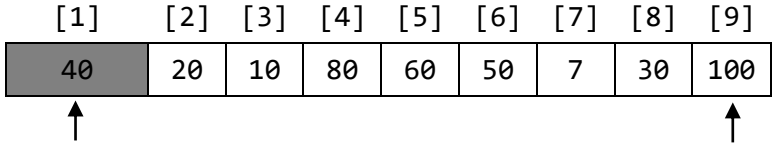
- Average case running time is much closer to the best case.
- If suppose the partitioning algorithm produces a 9-to-1 proportional split the recurrence will be

  $T(n)=T(9n/10)+ T(n/10)+ \Theta(n)$

  Solving it,

  $T(n)= \Theta(n\log n)$
- **The running time of quick sort is therefore $\Theta(n\log n)$ whenever the split has constant proportionality.**

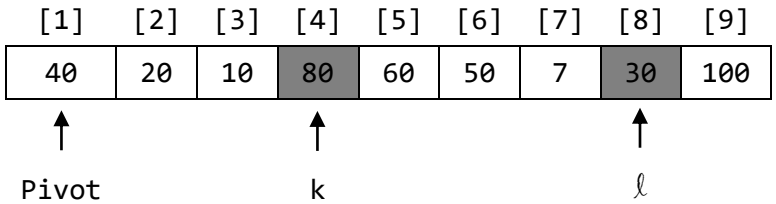
Example



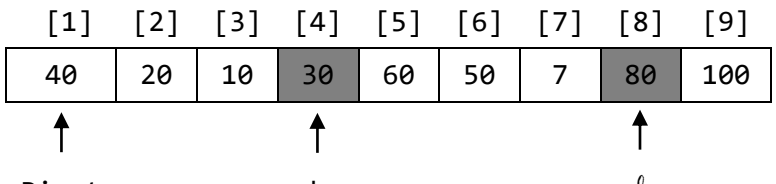
Here, Pivot = T[i], k = i, ℓ = j + 1

Repeat k = k + 1, Until T[k] > Pivot

Repeat ℓ = ℓ – 1, Until T[ℓ] <= Pivot



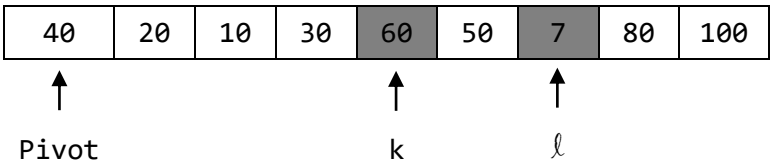
Is K < ℓ ? , Yes then Swap T[k] <-> T[ℓ]



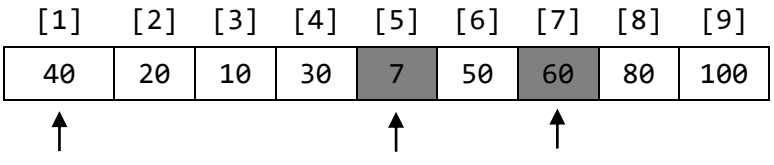
Repeat k = k + 1, Until T[k] > Pivot

Repeat ℓ = ℓ – 1, Until T[ℓ] <= Pivot

[1] [2] [3] [4] [5] [6] [7] [8] [9]

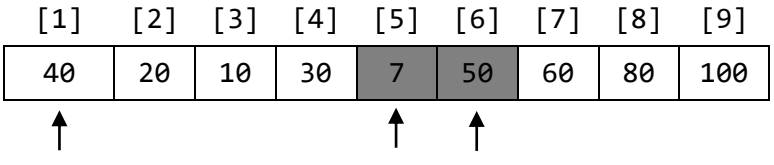


Is K < ℓ ? , Yes then Swap T[k] <-> T[ℓ]

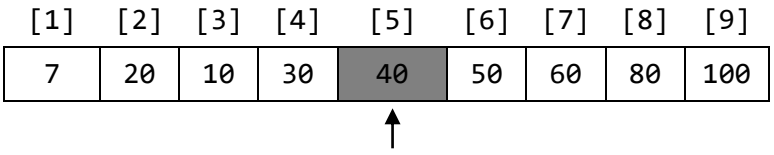


Repeat k = k + 1, Until T[k] > Pivot

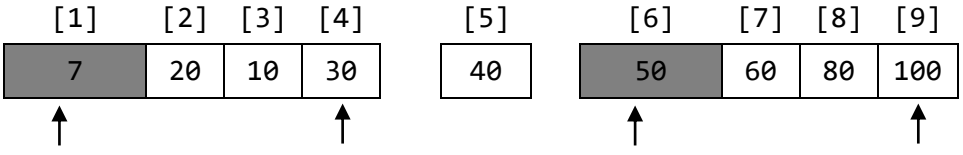
Repeat ℓ = ℓ – 1, Until T[ℓ] <= Pivot



Is K < ℓ ? , No then Swap Pivot <-> T[ℓ]



Now we have two sub list one having elements less or equal Pivot and second having elements greater than Pivot



Repeat the same procedure for each sub list until list is sorted

(6) Multiplying two n digit large integers using divide and conquer method

- Consider the problem of multiplying large integers.
- Following example shows how divide and conquer helps multiplying two large integers.

- Multiplication of 981 by 1234. First we pad the shorter operand with a non- significant zero to make it the same length as the longer one; thus 981 become 0981.
- Then we split each operand into two parts:
- 0981 gives rise to *w* = 09 and x = 81, and 1234 to y = 12 and *z* = 34.
- Notice that $981 = 10^2W + x$ and $1234 = 10^2y + z$.
- Therefore, the required product can be computed as

$$981 \times 1234 = (10^2w + x) * (10^2y + z)$$
$$= 10^4wy + 10^2(wz + xy) + xz$$
$$= 1080000 + 127800 + 2754 = 1210554.$$

- The above procedure still needs four half-size multiplications: wy, wz, xy and xz.
- There is no need to compute both *wz* and *xy;* all we really need is the *sum* of the two terms, consider the product

r = (w + x) x (y + z) = wy + (wz + xy) + xz.

- After only one multiplication, we obtain the sum of all three terms needed to calculate the desired product.
- This suggests proceeding as follows.

p = wy = 09 x 12 =108
***q = xz =* 81 x 34 = 2754**
r *= (w + x) x (y + z)* =90x46 = 4140,

- and finally

$981 \times 1234 = 10^4p + 10^2 (r - p - q) + q$
= 1080000 + 127800 + 2754 1210554.

- Thus the product of 981 and 1234 can be reduced to *three* multiplications of two-figure numbers (09 x 12, 81 x 34 and 90 x 46) together with a certain number of shifts (multiplications by powers of 10), additions and subtractions.
- It thus seems reasonable to expect that reducing four multiplications to three will enable us to cut 25% of the computing time required for large multiplications.
- We obtain an algorithm that can multiply two n-figure numbers in a time,

$T(n)= 3t (n /2) + g(n),$ when *n* is even and sufficiently large.

- Solving it gives,

$$T(n) \in \theta(n^{lg3} \mid n \text{ is a power of } 2)$$

(1) Explain Greedy Approach. ***OR*** Give the characteristics of Greedy algorithm. ***OR*** Write the general structure of greedy algorithm. ***OR*** Elements of Greedy Strategy.

- Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later.
- Greedy algorithms and the problems that can be solved by greedy algorithms are characterized by most or all of the following features.
- To solve a particular problem in an optimal way using greedy approach, there is a set or list of candidates C.
- For example: In Make Change Problem - the coins that are available, In Minimum Spanning Tree Problem - the edges of a graph that may be used to build a path, In Job Scheduling Problem - the set of jobs to be scheduled, etc..
- Once a candidate is selected in the solution, it is there forever: once a candidate is excluded from the solution, it is never reconsidered.
- To construct the solution in an optimal way, Greedy Algorithm maintains two sets. One set contains candidates that have already been considered and chosen, while the other set contains candidates that have been considered but rejected.
- The prototype for generalized greedy algorithm is given below:

Algorithm

```
Function greedy(C : set): set
{C is the set of candidates.}
S ← Ø {S is a set that will hold the solution}
while C ≠ Ø and not solution(S) do
        x ← select (C)
        C ← C \ {x}
        if feasible ( S U {x}) then
                S ← S U { x}
if solution (S) then return S
else return "no solution found"
```

- The greedy algorithm consists of four functions.
  1. Solution Function**:-** A function that checks whether chosen set of items provides a solution.
  2. Feasible Function**:-** A function that checks the feasibility of a set.
  3. Selection Function**:-** The selection function tells which of the candidates is the most promising.
  4. Objective Function**:-** An objective function, which does not appear explicitly, but gives the value of a solution.

(2) Making Change algorithm based on greedy approach.

- Suppose we live in a country where the following coins are available: dollars (100 cents), quarters (25 cents), dimes (10 cents), nickels (5 cents) and penny (1 cent).
- Our problem is to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins.
- For instance, if we must pay $2.89 (289 cents), the best solution is to give the customer total 10 coins: 2 dollars, 3 quarters, 1 dime and 4 coins of penny.
- Most of us solve this kind of problem every day without thinking twice, unconsciously using an obvious greedy algorithm: starting with nothing, at every stage we add the largest valued coin available without worrying about the consequences.
- The algorithm may be formalized as follows.

Algorithm

```
Function make-change(n): set of coins
{Makes change for amount n using the least possible number of coins.}
const C = {100, 25, 10,  5, 1} {C is the candidate set}
S ← Ø {S is a set that will hold the solution}
sum ← 0 {sum is the sum of the items in solution set S}
while sum ≠ n do
        x ← the largest item in C such that s + x ≤ n
        if there is no such item then
                return "no solution found"
        S ← S U {a coin of value x)
sum ← sum + x
return S
```

- With the given values of the coins this algorithm can produces an optimal solution to make change problem if an adequate supply of each denomination is available.
- However with a different series of values, or if the supply of some of the coins is limited, the greedy algorithm may not work.
- In some cases it may choose a set of coins that is not optimal (that is, the set contains more coins than necessary), while in others it may fail to find a solution at all even though solution exists.
- The algorithm is "greedy" because at every step it chooses the largest coin it can, without worrying whether this will prove to be a correct decision later.
- Furthermore it never changes its mind: once a coin has been included in the solution, it is there forever.

(3) Spanning tree and Minimum spanning tree.

- Let G = ‹N, A› be a connected, undirected graph where N is the set of nodes and A is the set of edges. Each edge has a given non-negative length.
- A spanning tree of a graph G is a sub-graph which is basically a tree and it contains all the vertices of G but does not contain cycle.
- A minimum spanning tree of a weighted connected graph G is a spanning tree with minimum or smallest weight of edges.

(4) Explain Kruskal's Algorithm for finding minimum spanning tree.

- In Kruskal's algorithm, the set *A* of edges are sorted in increasing order of their length.
- The solution set *T* of edges is initially empty.
- As the algorithm progresses, edges are added to set *T*.
- We examine the edges of set *A* one by one.
  - If an edge joins two nodes in different connected components, we add it to set *T*.
  - So, the two connected components now form only one component.
  - The edge is rejected if it joins two nodes in the same connected component, and therefore cannot be added to *T* as it forms a cycle.
- The algorithm stops when n-1 edges for n nodes are added in the solution set *T*.
- At the end of the algorithm only one connected component remains, and *T* is then a minimum spanning tree for all the nodes of *G*.
- The algorithm is described as follows:

Algorithm

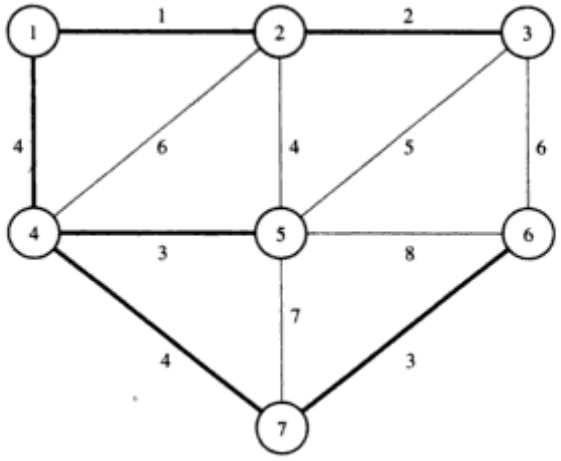
Function Kruskal*(G = (N, A): graph; length: A → R^+): set of edges*
{initialization}
(i) Sort A by increasing length
(ii) n ← the number of nodes in N
(iii) T ← Ø {Solution Set that will contain the edges of the minimum spanning tree}
(iv) Initialize n sets, each containing a different element of set N
{greedy loop}
(v) repeat
e ← {u, v} ← such that e is the shortest edge not yet considered
ucomp ← find(u)
vcomp ← find(v)
if *ucomp ≠ vcomp* ***then***
merge(ucomp, vcomp)
T ← T U {e}
until *T contains n - 1 edges*
(vi) ***return*** *T*

- The complexity for the Kruskal's algorithm is in **Θ(a log n)** where a is total number of edges and n

is the total number of nodes in the graph G.

Example:

Find the minimum spanning tree for the following graph using Kruskal's Algorithm.



Solution:

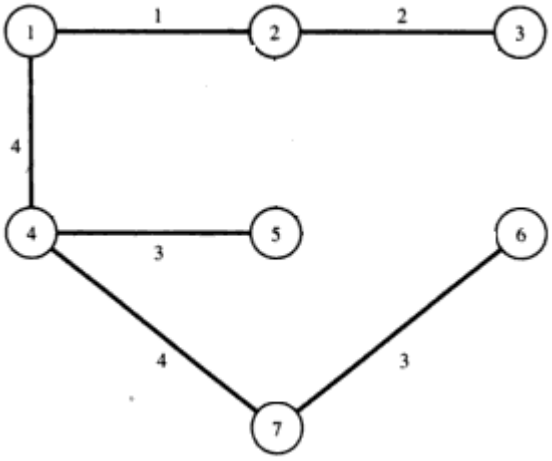
Step 1: Sort A by increasing length

- In increasing order of length the edges are: {1, 2}, {2, 3}, {4, 5}, {6, 7}, {1,4}, {2, 5}, {4,7}, {3, 5), {2, 4}, {3,6}, {5,7} and {5,6}.

Step 2:

Step	Edges considered - {u, v}	Connected Components
Initialization	-	{1}{2}{3}{4}{5}{6}{7}
1	{1,2}	{1,2}(3}{4}{5}{6}{7}
2	{2,3}	{1,2,3}{4}{5}{6}{7}
3	{4,5}	{1,2,3}{4,5}{6}{7}
4	{6,7}	{1,2,3}{4,5}{6,7}
5	{1,4}	{1,2,3,4,5}{6,7}
6	{2,5}	**Rejected**
7	{4,7}	{1,2,3,4,5,6,7}

- When the algorithm stops, solution set *T* contains the chosen edges {1, 2}, {2, 3}, {4, 5}, {6, 7}, {1, 4} and {4, 7}.
- This minimum spanning tree is shown below whose total length is **17**.



(5) Prim's Algorithm to obtain minimum spanning tree.

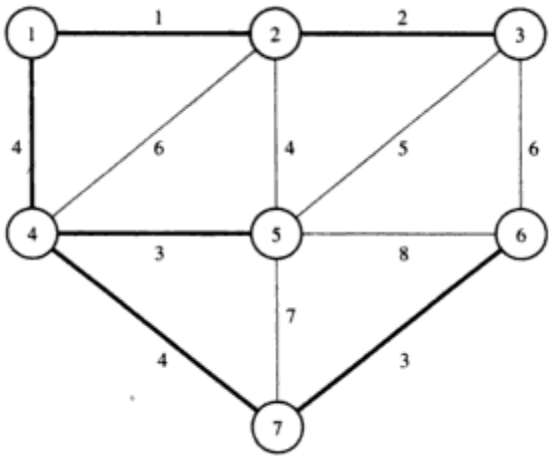
- In Prim's algorithm, the minimum spanning tree grows in a natural way, starting from an arbitrary root.
- At each stage we add a new branch to the tree already constructed; the algorithm stops when all the nodes have been reached.
- Let *B* be a set of nodes, and *A* is a set of edges.
- Initially, B contains a single arbitrary node, and solution set *T* is empty.
- At each step Prim's algorithm looks for the shortest possible edge {u, v} such that u ε B and v ε N\B.
- It then adds v to set *B* and {u, v} to solution set *T*.
- In this way the edges in *T* form a minimum spanning tree for the nodes in *B*.
- We continue thus as long as B ≠ N.
- The complexity for the Prim's algorithm is **$\Theta(n^2)$** where n is the total number of nodes in the graph G.

Algorithm

Function Prim*(G = (N, A): graph; length: A — R^+): set of edges*
{initialization}
T ← Ø
B ← {an arbitrary member of N}
while *B ≠ N* ***do***
find e = {u, v} of minimum length such that
u ε B and v ε N \ B
T ← T U {e}
B ← B U {v}
return T

Example:

Find the minimum spanning tree for the above graph using Prim's Algorithm.



Solution:

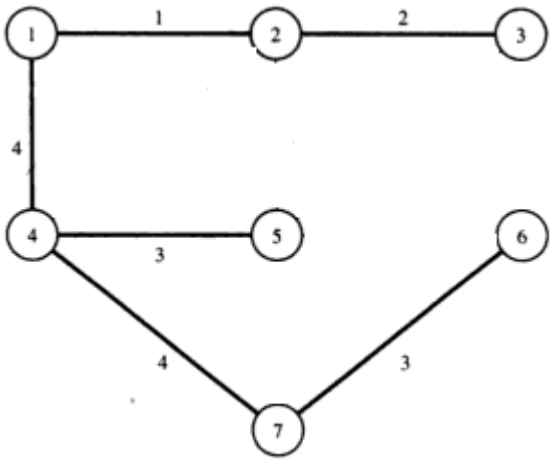
Step 1

- We arbitrarily choose node 1 as the starting node.

Step 2

Step	Edge Selected {u, v}	Set B	Edges Considered
Initialization	-	{1}	--
1	{1,2}	{1,2}	**{1,2}** {1,4}
2	{2,3}	{1,2,3}	{1,4} **{2,3}**{2,4}{2,5}
3	{1,4}	{1,2,3,4}	**{1,4}**{2,4}{2,5}{3,5} {3,6}
4	{4,5}	{1,2,3,4,5}	{2,4}{2,5}{3,5} {3,6} **{4,5}** {4,7}
5	{4,7}	{1,2,3,4,5,7}	{2,4}{2,5}{3,5} {3,6} **{4,7}** {5,6} {5,7}
6	{6,7}	{1,2,3,4,5,6,7}	{2,4}{2,5}{3,5} {3,6} {5,6} {5,7}**{6,7}**

- When the algorithm stops, T contains the chosen edges {1, 2}, {2,3}, {1,4}, {4,5}, {4,7} and {7,6}.
- This minimum spanning tree is shown below whose total length is **17**.



(6) Dijkstra's algorithm for finding shortest path.

OR

Single source shortest path algorithm

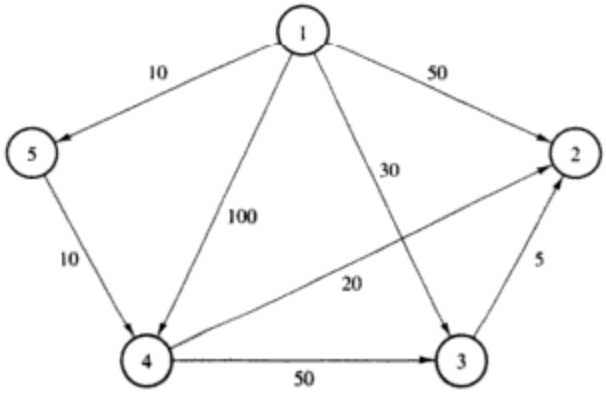
- Consider now a directed graph G = (N, A) where *N* is the set of nodes of graph G and *A* is the set of directed edges.
- Each edge has a positive length.
- One of the nodes is designated as the source node.
- The problem is to determine the length of the shortest path from the source to each of the other nodes of the graph.
- This problem can be solved by a greedy algorithm often called Dijkstra's algorithm.
- The algorithm uses two sets of nodes, *S* and *C*.
- At every moment the set *S* contains those nodes that have already been chosen; as we shall see, the minimal distance from the source is already known for every node in *S*.
- The set *C* contains all the other nodes, whose minimal distance from the source is not yet known, and which are candidates to be chosen at some later stage.
- Hence we have the invariant property *N* = *S* U *C*.
- Initially, S contains only the source itself; when the algorithm stops, S contains all the nodes of the graph and our problem is solved.
- At each step we choose the node in *C* whose distance to the source is smallest, and add it to *S*.
- We shall say that a path from the source to some other node is **special** if all the intermediate nodes along the path belong to *S*.
- At each step of the algorithm, an array D holds the length of the shortest special path to each node of the graph.
- At the moment when we add a new node v to *S*, the shortest special path to v is also the shortest of all the paths to v.
- When the algorithm stops, all the nodes of the graph are in *S*, and so all the paths from the source to some other node are special. Also the values in D give the solution to the shortest path problem.

- For simplicity, we assume that the nodes of G are numbered from 1 to n, so N = {1, 2...., n}.
- We can suppose that node 1 is the source.
- The algorithm maintains a matrix L which gives the length of each directed edge:
  - L[i, j] ≥ 0 if the edge (i, j) ϵ A, and L[i, j] = **∞** otherwise.
- The time required by the Dijkstra's algorithm is **Θ(n^2).**

Algorithm

```
Function Dijkstra(L[1 .. n, 1 .. n]): array [2.. n]
    array D[2.. n]
    {initialization}
    C ← {2,3,…, n} {S = N \ C exists only implicitly}
    for i ← 2 to n do D[i] ← L[1, i]
    {greedy loop}
    repeat n - 2 times
        v ← some element of C minimizing D[v]
        C ← C \ {v} {and implicitly S ← S U {v}}
        for each w ϵ C do
            D[w] ← min(D[w], D[v] + L[v, w])
    return D
```

Example:



- The algorithm proceeds as follows on the graph given above.

Step	v	C	D
Initialization	-	{2,3,4,5}	[50,30,100,10]
1	5	{2,3,4}	[50,30,**20**,10]
2	4	{2,3}	[**40**,30,20,10]
3	3	{2}	[**35**,30,20,10]

(7) Fractional knapsack problem with example. *OR* Knapsack problem using Greedy Approach.

- We are given n objects and a knapsack.
- Object i has a positive weight w_i and a positive value v_i for i = 1, 2... n.
- The knapsack can carry a weight not exceeding W.
- Our aim is to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint.
- In fractional knapsack problem, we assume that the objects can be broken into smaller pieces, so we may decide to carry only a fraction x_i of object i, where $0 \le x_i \le 1$.
- In this case, object i contribute $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load.
- Symbolic Representation of the problem can be given as follows:

$$\textbf{maximize } \sum_{i=1}^{n} x_i v_i \textbf{ subject to } \sum_{i=1}^{n} x_i w_i \le W$$

Where, $v_i > 0$, $w_i > 0$ and $0 \le x_i \le 1$ for $1 \le i \le n$.

Greedy Approach

- We shall use a greedy algorithm to solve the problem.
- In terms of our general template of Greedy Algorithm, the candidates are the different objects, and a solution is a vector $(x_1,...,x_n)$ telling us what fraction of each object to include in the knapsack.

Algorithm

```
Function knapsack(w[1..n],v[1..n],W): array [1..n]
    {initialization}
    for i ← 1 to n do x[i] ← 0
        weight ← 0
        {greedy loop}
        while weight < W do
            i ← the best remaining object based on vi/wi
            if weight + w[i] ≤ W then x[i] ← 1
                    weight ← weight + w[i]
            else x[i] ← (W - weight) / w[i]
                    weight ←W
    return x
```

- A feasible solution is one that respects the constraints given above, and the objective function is the total value of the objects in the knapsack.
- Three possible selection functions are there
  i. Most valuable remaining object: select the object with the highest value of v_i
  ii. Lightest weight remaining object: select the object with the lowest value of w_i
  iii. Object whose value per unit weight is as high as possible.

Example:

- We are given 5 objects and weight carrying capacity of knapsack W = 100.
- For each object, weight w_i and value v_i are given in the following table.

Object i	**1**	**2**	**3**	**4**	**5**
$\mathbf{w_i}$	10	20	30	40	50
$\mathbf{v_i}$	20	30	66	40	60
$\mathbf{v_i / w_i}$	2.0	1.5	2.2	1.0	1.2

Solution:

Selection	**x_i**					**Value**
Max v_i	0	0	1	0.5	1	146
Min w_i	1	1	1	1	0	156
Max v_i / w_i	1	1	1	0	0.8	**164**

- Here solution is given by the third selection function Max v_i / w_i, where the total value gained by selected objects is maximum.

(8) Activity Selection problem

- An activity-selection is the problem of scheduling a resource among several competing activities.
- We are given a set *S* of n activities with start time S_i and finish time f_i, of an i^{th} activity. Find the maximum size set of mutually compatible activities.
- Activities *i* and *j* are compatible if the half-open internal $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap, that is, *i* and *j* are compatible if $\boldsymbol{s_i \geq f_j}$ **and** $\boldsymbol{s_j \geq f_i}$
- Here, we sort the activities of set S as per increasing finish time so that we can directly identify mutually compatible activity by comparing finish time of first activity and start time of next activity.
- Whenever we find the compatible activity, add it to the solution set *A* otherwise consider the next activity in the sequence.

Greedy Algorithm for Activity Selection

I. *Sort the input activities by increasing finishing time.* $f_1 \leq f_2 \leq \ldots \leq f_n$

II. *Call* ***GREEDY-ACTIVITY-SELECTOR*** *(s, f)*

```
1.  n = length [s]
2.  A={i}
3.  j = 1
4.  for  i = 2  to  n
5.       do if   s_i ≥ f_j
6.            then  A= AU{i}
7.                  j = i
return  set A
```

Example:

- Let 11 activities are given *S* = {*p, q, r, s, t, u, v, w, x, y, z*}and start and finished times for proposed activities are (1, 4), (3, 5), (0, 6), 5, 7), (3, 8), 5, 9), (6, 10), (8, 11), (8, 12), (2, 13) and (12, 14) respectively.

Solution:

A = {p} Initialization at line 2
A = {p, s} line 6 - 1st iteration of FOR - loop
A = {p, s, w} line 6 -2nd iteration of FOR - loop
A = {p, s, w, z} line 6 - 3rd iteration of FOR-loop
Answer A = {p, s, w, z}

Analysis :

- Part I requires **O(n log n)** time (by using merge of heap sort).
  Part II requires **Θ (n)** time assuming that activities were already sorted in part I by their finish time.
- So, the total time required by algorithm is **Θ (nlogn).**

(9) Job scheduling with deadlines using greedy approach.

- We have set of n jobs to execute, each of which takes unit time.
- At any point of time we can execute only one job.
- Job i earns profit $\mathbf{g_i} > 0$ if and only if it is executed no later than time $\mathbf{d_i}$.
- We have to find an optimal sequence of jobs such that our total profit is maximized.
- A set of job is feasible if there exits at least one sequence that allows all the jobs in the set to be executed no later than their respective deadlines.
- The greedy algorithm for this problem consists of constructing the schedule step by step.
- At each step the job with the highest value of profit g_i is added in the solution set among those that are not yet considered, provided that the chosen set of jobs remains feasible.

Algorithm

1. *Sort all the n jobs in decreasing order of their profit.*
2. *Let total position P = min(n, max(d_i))*
3. *Each position 0, 1, 2…, P is in different set and F({i}) = i, for 0 ≤ i ≤ P.*
4. *Find the set that contains d, let this set be K. if F(K) = 0 reject the job; otherwise:*
   a. *Assign the new job to position F(K).*
   b. *Find the set that contains F(K) – 1. Call this set L.*
   c. *Merge K and L. the value for this new set is the old value of F(L).*

Example:

- Using greedy algorithm find an optimal schedule for following jobs with n=6. Profits: (P1,P2,P3,P4,P5,P6) = (20,15,10,7,5,3) and deadline (d1,d2,d3,d4,d5,d6) = (3, 1, 1, 3, 1, 3).

Solution:

Job i	1	2	3	4	5	6
Profit g_i	20	15	10	7	5	3

Deadline d_i	3	1	1	3	1	3

Steps:

- Here jobs are already sorted in decreasing order of their profit.
- Let P = min(6, 3) = 3

F =	0	1	2	3
Job selected	0	0	0	0

- d_1 = 3 : assign job 1 to position 3

F =	0	1	2	0
Job selected	0	0	0	1

- d_2 = 1 : assign job 2 to position 1

F =	0	0	2	0
Job selected	0	2	0	1

- d_3 = 1 : No free position so **reject the job**.
- d_4 = 3 : assign job 4 to position 2 as position 3 is not free but position 2 is free.

F =	0	0	0	0
Job selected	0	2	4	1

- Now no more free position is left so no more jobs can be scheduled.
- **The final optimal sequence: Execute the job in order 2, 4, 1 with total profit value 42.**

(10) Huffman Codes

- Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code.
- Huffman coding is a lossless data compression algorithm.
- The idea is to assign variable-length codes to input characters.
- Lengths of the assigned codes are based on the frequencies of corresponding characters.
- The most frequent character gets the smallest code and the least frequent character gets the largest code.
- The variable-length codes assigned to input characters are Prefix Codes.
- In Prefix codes, the codes (bit sequences) are assigned in such a way that the code assigned to one character is not a prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.
- There are mainly two major parts in Huffman Coding
  1. Build a Huffman Tree from input characters.
  2. Traverse the Huffman Tree and assign codes to characters.

Algorithm

HUFFMAN(C)

1. *n = |C|*
2. *Q = C*
3. ***for*** *i = 1* ***to*** *n-1*
4. *allocate a new node z*
5. *z.left = x = EXTRACT-MN(Q)*
6. *z.right = y = EXTRACT-MIN(Q)*
7. *z.freq = x.freq + y.freq*
8. *INSERT(Q,z)*
9. ***return*** *EXTRACT-MIN(Q) // return the root of the tree*

- We assume here that C is a set of n characters and that each character c ∈ C is an object with an attribute *c. freq* giving its frequency.
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner.
- It begins with a set of |C| leaves and performs a sequence of |C|- 1 "merging" operations to create the final tree.
- The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together.
- When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

Example:

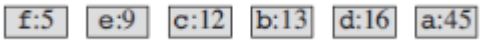
- Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.
- Suppose we have a 100,000-character data file that we wish to store compactly.
- We observe that the characters in the file occur with the frequencies given by following table.
- That is, only 6 different characters appear, and the character ***a*** occurs 45,000 times.

Characters	**a**	**b**	**c**	**d**	**e**	**f**
Frequency (in thousand)	**45**	**13**	**12**	**16**	**9**	**5**

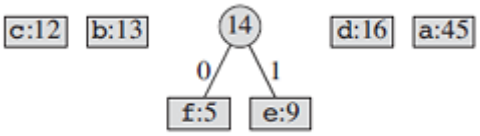
Solution:

- The steps of Huffman's algorithm for the frequencies given above as step no. 1 to 6.
- Each step shows the contents of the queue sorted into increasing order by frequency.
- At each step, the two trees with lowest frequencies are merged.
- Leaves are shown as rectangles containing a character and its frequency.
- Internal nodes are shown as circles containing the sum of the frequencies of their children.
- An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child.
- The code-word for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter.

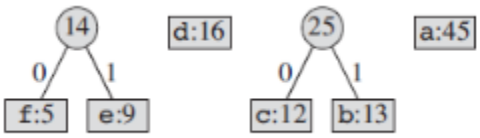
Step-1: Arrange elements in ascending order:-



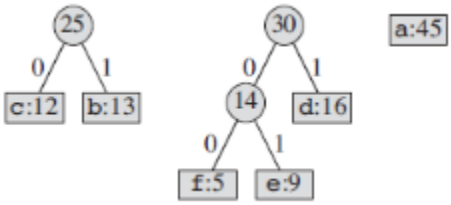
Step-2:



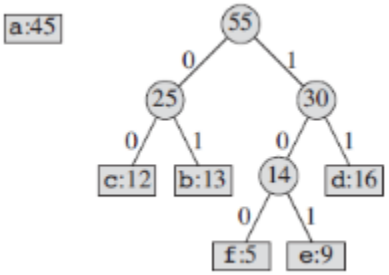
Step-3:



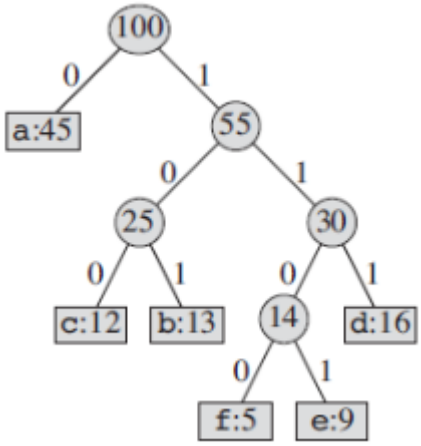
Step-4:



Step-5:



Step-6:



Frequency (in thousand)	*a*	*b*	*c*	*d*	*e*	*f*
	45	**13**	**12**	**16**	**9**	**5**
Huffman code-word	**0**	**101**	**100**	**111**	**1101**	**1100**

(1) Introduction of dynamic programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub problems.
- Divide-and-conquer algorithms partition the problem into independent sub problems, solve the sub problems recursively, and then combine their solutions to solve the original problem.
- In contrast, dynamic programming is applicable when the sub problems are not independent, that is, when sub problems share sub problems.
- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub problems.
- A dynamic-programming algorithm solves every sub problem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub problem is encountered.
- The development of a dynamic-programming algorithm can be broken into a sequence of four steps.
  1. Characterize the structure of an optimal solution.
  2. Recursively defines 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.

(2) Principle of optimality

- The dynamic programming algorithm obtains the solution using principle of optimality.
- The principle of optimality states that "in an optimal sequence of decisions or choices, each subsequence must also be optimal".
- When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach.
- The principle of optimality: "If k is a node on the shortest path from i to j, then the part of the path from i to k, and the part from k to i, must also be optimal."

(3) Binomial coefficient

Consider the problem of calculating binomial coefficient

$$\binom{n}{k} = \begin{cases} 1 & \text{if k = 0 or k = n} \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if 0 < k < n} \\ 0 & \text{Otherwise} \end{cases}$$

Suppose 0 ≤ k ≤ n. if we calculate $\binom{n}{k}$ directly by

```
function C(n, k)
    if k=0 or k=n then return 1
    else return C(n-1, k-1)+ C(n-1, k)
```

Many of the values C(i, j), i < n, j < k, are calculated over and over, for example the algorithm calculates C(5,3) as sum of C(4,2) and C(4,3) Both these intermediate results require us to calculate C(3,2). Similarly the value of C (2,2) is used several times.

(4) Making change problem using dynamic programming

Algorithm

```
function coins (N )
{Gives the minimum number og coins needed to make change for N units.
Array d[1..n] specifies the coinage: in the example there are coins for
1, 4 and 6 units.}
array d[1..n] =[1, 4, 6]
array c[1..n,0..N]
for i←1 to n do c[i,0]←0
for i←1 to n do
     for j←1 to N do
          c[i,j]←if i=1 and j < d[i] then +∞
                 else if i=1 then 1+ c[1,j-d[1]]
                 else if j < d[i] then c[i-1,j]
                 else min (c[i-1,j],1+ c[i,j-d[i]]
return c[n,N]
```

- We need to generate table c[n][N]. Where,
- n= number of denominations. Here we are having 3 denomination so n=3.
- N= number of units that you need to make change. Here we need change of 8 units so N=8
- To generate table c[i][j] use following steps

```
Step-1:  Make c[i][0]=0 for 0 < i ≤n
Step-2:  Repeat step-2 to step-4 for remaining matrix Values
          if i=1 then c[i][j] = 1+c[1][j-d1], here d1=1
Step-3:  if j<di then c[i][j] = c[i-1][j]
Step-4   otherwise c[i][j] = min(c[i-1][j],1+c[i][j-di]
```

- ***Example:*** Denominations: d1=1, d2=4, d3=6. Make a change of Rs. 8.
- c[i][j]=

i ↓	j →	0	1	2	3	4	5	6	7	8
1	d_1=1	0	1	2	3	4	5	6	7	8
2	d2=4	0	1	2	3	1	2	3	4	2
3	d3=6	0	1	2	3	1	2	1	2	**2**

We need minimum C[3][8]=2 coins for change

▪ c[1][1] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][1]=1+c[1][1-1] =1+c[1][0] =1	▪ c[1][2] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][2]=1+c[1][2-1] =1+c[1][1] =2
▪ c[1][3] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][3]=1+c[1][3-1] =1+c[1][2] =3	▪ c[1][4] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][4]=1+c[1][4-1] =1+c[1][3] =4
▪ c[1][5] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][5]=1+c[1][5-1] =1+c[1][4] =5	▪ c[1][6] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][6]=1+c[1][6-1] =1+c[1][5] =6
▪ c[1][7] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][7]=1+c[1][7-1] =1+c[1][6] =7	▪ c[1][8] here, i=1 so use, c[i][j]=1+c[1][j-d1] c[1][8]=1+c[1][8-1] =1+c[1][7] =8
▪ c[2][1] here j<d2(1<4) so use, c[i][j]=c[i-1][j] c[2][1]=c[2-1][1] =1	▪ c[2][2] here j<d2(2<4) so use, c[i][j]=c[i-1][j] c[2][2]=c[2-1][2] =2
▪ c[2][3] here j<d2(3<4) so use, c[i][j]=c[i-1][j] c[2][3]=c[2-1][3] =3	▪ c[2][4] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[2][4]=min(c[1,4], 1+c[2,4-4]) =min(4,1+0) =1
▪ c[2][5] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[2][5]=min(c[1,5], 1+c[2,5-4]) =min(5,1+1) =2	▪ c[2][6] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[2][6]=min(c[1,6], 1+c[2,6-4]) =min(6,1+2) =3
▪ c[2][7] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[2][7]=min(c[1,7], 1+c[2,7-4]) =min(7,1+3) =4	▪ c[2][8] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[2][8]=min(c[1,8], 1+c[2,8-4]) =min(8,1+1) =2
▪ c[3][1] here j<d3(1<6) so use c[i][j]=c[i-1][j] c[3][1]=c[3-1][1] =1	▪ c[3][2] here j<d3(2<6) so use c[i][j]=c[i-1][j] c[3][2]=c[3-1][2] =2
▪ c[3][3] here j<d3(3<6) so use c[i][j]=c[i-1][j] c[3][3]=c[3-1][3] =3	▪ c[3][4] here j<d3(4<6) so use c[i][j]=c[i-1][j] c[3][4]=c[3-1][4] =1
▪ c[3][5] here j<d3(5<6) so use c[i][j]=c[i-1][j] c[3][5]=c[3-1][5] =2	▪ c[3][6] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[3][6]=min(c[2,6], 1+c[3,6-6]) =min(3,1+0) =1

▪ c[3][7] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[3][7]=min(c[2,7], 1+c[3,7-6]) =min(4,1+1) =2	▪ c[3][8] use c[i][j]=min(c[i-1][j], 1+c[i][j-di] c[3][8]=min(c[2,8], 1+c[3,8-6]) =min(2,1+2) =2

(5) Assembly line scheduling using dynamic programming

- Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$
- Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
- Entry times are: e_1 and e_2; exit times are: x_1 and x_2
- After going through a station, can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after $S_{i,j}$ is $t_{i,j}$, j = 1, . . . , n – 1

Steps:

f^* : the fastest time to get through the entire factory

$f_i[j]$: the fastest time to get from the starting point through station $S_{i,j}$

$f^* = \min (f_1[n] + x_1, f_2[n] + x_2)$

Base case: j = 1, i=1,2 (getting through station 1)

$f_1[1] = e_1 + a_{1,1}$

$f_2[1] = e_2 + a_{2,1}$

General Case: j = 2, 3, …,n, and i = 1, 2

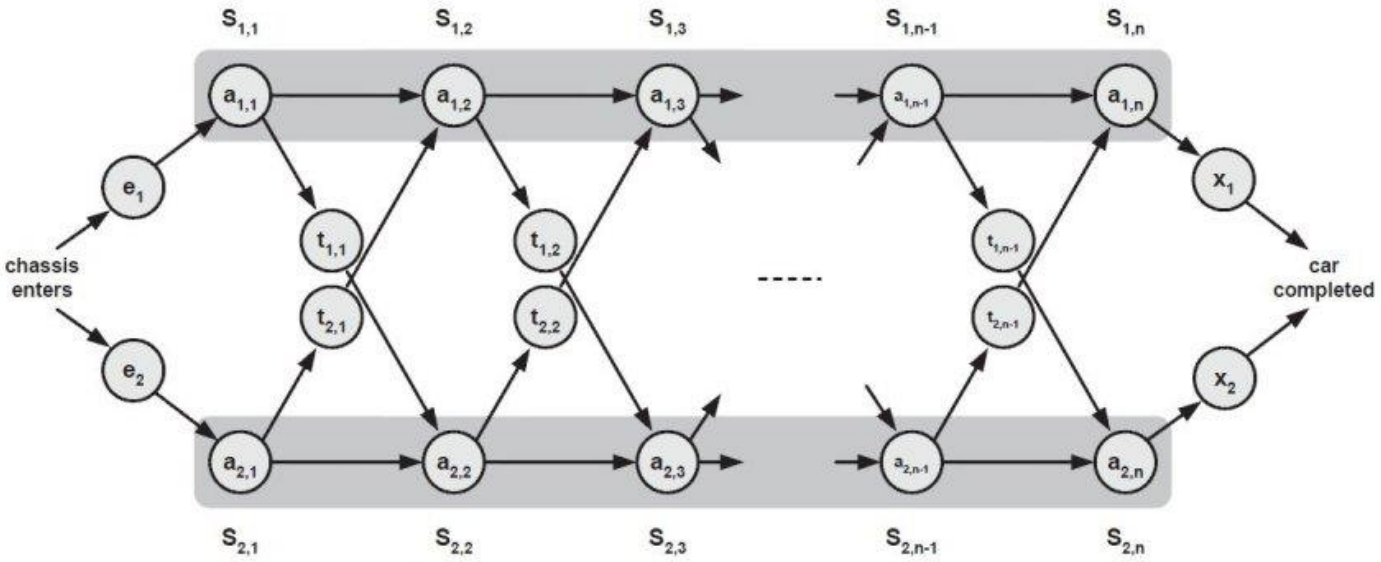
Fastest way through $S_{1,j}$ is either:

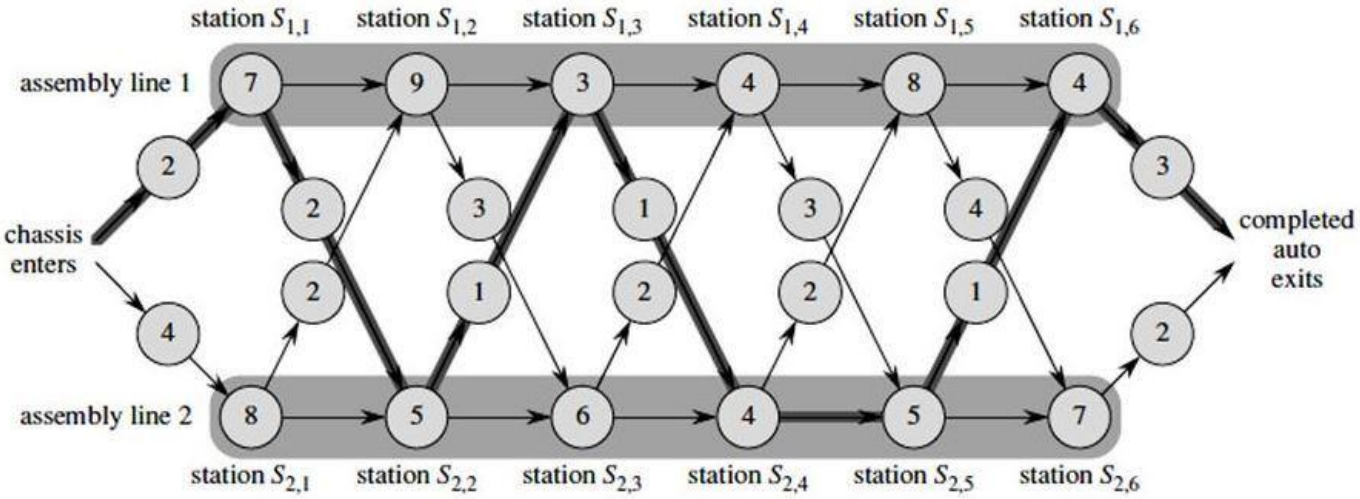
- the way through $S_{1,j-1}$ then directly through $S_{1,j}$, or $f_1[j-1] + a_{1,j}$
- the way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$ $f_2[j-1] + t_{2,j-1} + a_{1,j}$

$$f_1[j] = \begin{cases} e_1 + a_{1,1}, & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}), & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1}, & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}), & \text{if } j \geq 2 \end{cases}$$

$f^* = \min (f_1[n] + x_1, f_2[n] + x_2)$





	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

f*=min (35+3, 37+2)
= 38

- $f_1[1]$ here, j=1 so use, $f_1[1] = e_1 + a_{1,1}$
  $f_1[1] = 2 + 7 = 9$
- $f_2[1]$ here, j=1 so use, $f_2[1] = e_2 + a_{2,1}$
  $f_1[1] = 4 + 8 = 12$
- $f_1[2]$ here, j=2 so use,
  $f_1[2] = \min(f_1[j - 1] + a_{1,j}, f_2[j -1] + t_{2,j-1} + a_{1,j})$
  $= \min(9+9, 12+2+9) = 18$
- $f_2[2]$ here, j=3 so use,
  $f_2[1] = \min(f_2[j - 1] + a_{2,j}, f_1[j -1] + t_{1,j-1} + a_{2,j})$
  $= \min(12+5, 9+2+5) = 16$

Repeating same formula for $f_1[3]$… $f_1[6]$ and $f_2[3]$… $f_2[6]$

- f*=min ($f_1[n]$ + x1, $f_2[n]$ + x2)
  =min (35+3, 37+2) = 38

(6) 0/1 knapsack problem using dynamic programming

- We need to generate table V(1…n,0…W) where, n= number of objects. Here n=5
- W= capacity of knapsack. Here W=11
- To generate table V[i][j] use following steps

```
Step-1: Make V[i][0] = 0 for 0 < i ≤n
Step-2: if j < ωi then take V[i][j] = V[i-1][j]
Step-3: if j ≥ ωi then take V[i][j] = max(V[i-1][j], V[i-1][j- ωi]+Vi)
```

- Solution: V[i][j]=

		i ↓ / j →	0	1	2	3	4	5	6	7	8	9	10	11
$\omega_1=1$	$V_1=1$	1	0	1	1	1	1	1	1	1	1	1	1	1
$\omega_2=2$	$V_2=6$	2	0	1	6	7	7	7	7	7	7	7	7	7
$\omega_3=5$	$V_3=18$	3	0	1	6	7	7	18	19	24	25	25	25	25
$\omega_4=6$	$V_4=22$	4	0	1	6	7	7	18	22	24	28	29	29	40
$\omega_5=7$	$V_5=28$	5	0	1	6	7	7	18	22	28	29	34	35	**40**

Maximum profit V[5,11]=40

▪ V[0][j] =0 where j=0 to W, here W=11 so, V[0][0]=0,V[0][1]=0, V[0][2]=0, V[0][3]=0, V[0][4]=0 V[0][5]=0, V[0][6]=0, V[0][7]=0, V[0][8]=0, V[0][9]=0 V[0][10]=0, V[0][10]=0, V[0][11]=0	▪ V[i][0] =0 where i= 0 to n, here n=5 so, V[0][0]=0,V[1][0]=0, V[2][0]=0, V[3][0]=0, V[4] [0]=0 V[5] [0]=0
▪ V[1][1], here i=1,j=1, $\omega_1=1$, $v_1=1$ so j>=ω_1 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[1][1]=max(V[0][1],V[0][0]+v_1) =max(0,1+0) =1	▪ V[1][2], here i=1,j=2, $\omega_1=1$, $v_1=1$ so j>=ω_1 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[1][2]=max(V[0][2],V[0][1]+v_1) =max(0,0+1) =1
▪ V[1][3], here i=1,j=3, $\omega_1=1$, $v_1=1$ so j>=ω_1 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[1][3]=max(V[0][3],V[0][2]+v_1) =max(0,0+1) =1	▪ V[1][4], here i=1,j=4, $\omega_1=1$, $v_1=1$ so j>=ω_1 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[1][4]=max(V[0][4],V[0][3]+v_1) =max(0,0+1) =1

▪ V[1][5], here i=1,j=5, ω_1=1, v_1=1 so j>=ω_1 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[1][5]=max(V[0][5],V[0][4]+v_1) =max(0,0+1) =1	▪ V[1][6], here i=1,j=6, ω_1=1, v_1=1 so j>=ω_1 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[1][5]=max(V[0][6],V[0][5]+v_1) =max(0,0+1) =1
Same Procedure for because j>=1 V[1][7]=1, V[1][8]=1, V[1][9]=1, V[1][10]=1, V[1][11]=1,	
▪ V[2][1], here i=2,j=1, ω_2=2, v_2=6 so j<ω_2 use V[i][j]= V[i-1][j] V[2][1]=V[1][1] =1	▪ V[2][2], here i=2,j=2, ω_2=2, v_2=6 so j>=ω_2 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[2][2] =max(V[1][2],V[1][0]+v_2) =max(1,0+6) =6
▪ V[2][3], here i=2,j=3, ω_2=2, v_2=6 so j>=ω_2 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[2][3] =max(V[1][3],V[1][1]+v_2) =max(1,1+6) =7	▪ V[2][4], here i=2,j=4, ω_2=2, v_2=6 so j>=ω_2 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[2][4] =max(V[1][4],V[1][2]+v_2) =max(1,1+6) =7
▪ V[2][5], here i=2,j=5, ω_2=2, v_2=6 so j>=ω_2 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[2][5] =max(V[1][5],V[1][3]+v_2) =max(1,1+6) =7	▪ V[2][6], here i=2,j=6, ω_2=2, v_2=6 so j>=ω_2 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[2][5] =max(V[1][6],V[1][4]+v_2) =max(1,1+6) =7
Same Procedure for below like V[2][6] , because j>=2 V[2][7]=7, V[2][8]=7, V[2][9]=7, V[2][10]=7, V[2][11]=7	
▪ V[3][1], here i=3,j=1, ω_3=5, v_3=18 so j<ω_3 use V[i][j]= V[i-1][j] V[3][1]=V[2][1] =1	▪ V[3][2], here i=3,j=2, ω_3= 5, v_3=18 so j<ω_3 use V[i][j]= V[i-1][j] V[3][2]=V[2][2] =6
▪ V[3][3], here i=3,j=3, ω_3= 5, v_3=18 so j<ω_3 use V[i][j]= V[i-1][j] V[3][3]=V[2][3] =7	▪ V[3][4], here i=3,j=4, ω_3= 5, v_3=18 so j<ω_3 use V[i][j]= V[i-1][j] V[3][3]=V[2][4] =7
▪ V[3][5], here i=3,j=5, ω_3= 5, v_3=18 so j>=ω_3 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[3][5] =max(V[2][5],V[2][0]+v_3) =max(7,0+18) =18	▪ V[3][6], here i=3,j=6, ω_3= 5, v_3=18 so j>=ω_3 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[3][6] =max(V[2][6],V[2][1]+v_3) =max(7,1+18) =19
▪ V[3][7], here i=3,j=7, ω_3= 5, v_3=18 so j>=ω_3 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[3][7] =max(V[2][7],V[2][2]+v_3) =max(7,6+18)	▪ V[3][8], here i=3,j=7, ω_3= 5, v_3=18 so j>=ω_3 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[3][8] =max(V[2][8],V[2][3]+v_3) =max(7,7+18)

=24	=25
▪ V[3][9], here i=3,j=9, ω_3= 5, v_3=18 so j>=ω_3 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[3][9] =max(V[2][9],V[2][4]+v_3) =max(7,7+18) =25	▪ V[3][10], here i=3,j=10, ω_3= 5, v_3=18 so j>=ω_3 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[3][10] =max(V[2][10],V[2][5]+v_3) =max(7,7+18) =25
▪ V[3][11], here i=3,j=11, ω_3= 5, v_3=18 so j>=ω_3 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[3][11] =max(V[2][11],V[2][6]+v_3) =max(7,7+18) =25	▪ V[4][1], here i=4,j=1, ω_4=6, v_4=22 so j<ω_4 use V[i][j]= V[i-1][j] V[4][1]=V[3][1] =1
▪ V[4][2], here i=4,j=2, ω_4=6, v_4=22 so j<ω_4 use V[i][j]= V[i-1][j] V[4][2]=V[3][2] =6	▪ V[4][3], here i=4,j=3, ω_4=6, v_4=22 so j<ω_4 use V[i][j]= V[i-1][j] V[4][3]=V[3][3] =7
▪ V[4][4], here i=4,j=4, ω_4=6, v_4=22 so j<ω_4 use V[i][j]= V[i-1][j] V[4][4]=V[3][4] =7	▪ V[4][5], here i=4,j=5, ω_4=6, v_4=22 so j<ω_4 use V[i][j]= V[i-1][j] V[4][5]=V[3][5] =18
▪ V[4][6], here i=4,j=6, ω_4= 6, v_4=22 so j>=ω_4 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[4][6] =max(V[3][6],V[3][0]+v_4) =max(19,0+22) =22	▪ V[4][7], here i=4,j=7, ω_4= 6, v_4=22 so j>=ω_4 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[4][7] =max(V[3][7],V[3][1]+v_4) =max(24,1+22) =24
▪ V[4][8], here i=4,j=8, ω_4= 6, v_4=22 so j>=ω_4 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[4][8] =max(V[3][8],V[3][2]+v_4) =max(25,6+22) =28	▪ V[4][9], here i=4,j=9, ω_4= 6, v_4=22 so j>=ω_4 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[4][9] =max(V[3][9],V[3][3]+v_4) =max(25,7+22) =29
▪ V[4][10], here i=4,j=10, ω_4= 6, v_4=22 so j>=ω_4 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[4][10] =max(V[3][10],V[3][4]+v_4) =max(25,7+22) =29	▪ V[4][11], here i=4,j=11, ω_4= 6, v_4=22 so j>=ω_4 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[4][11] =max(V[3][11],V[3][5]+v_4) =max(25,18+22) =40
▪ V[5][1], here i=4,j=1, ω_5=7, v_5= 28 so j<ω_5 use V[i][j]= V[i-1][j] V[5][1]=V[4][1] =1	▪ V[5][2], here i=4,j=2, ω_5=7, v_5= 28 so j<ω_5 use V[i][j]= V[i-1][j] V[5][2]=V[4][2] =6

▪ V[5][3], here i=4,j=3, ω_5=7, v_5= 28 so j<ω_5 use V[i][j]= V[i-1][j] V[5][3]=V[4][3] =7	▪ V[5][4], here i=4,j=4, ω_5=7, v_5= 28 so j<ω_5 use V[i][j]= V[i-1][j] V[5][4]=V[4][4] =7
▪ V[5][5], here i=4,j=5, ω_5=7, v_5= 28 so j<ω_5 use V[i][j]= V[i-1][j] V[5][5]=V[4][5] =18	▪ V[5][6], here i=4,j=6, ω_5=7, v_5= 28 so j<ω_5 use V[i][j]= V[i-1][j] V[5][5]=V[4][6] =22
▪ V[5][7], here i=5,j=7, ω_5= 7, v_4=28 so j>=ω_5 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[5][7] =max(V[4][7],V[4][0]+v_4) =max(24,0+28) =28	▪ V[5][8], here i=5,j=8, ω_5= 7, v_4=28 so j>=ω_5 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[5][8] =max(V[4][8],V[4][1]+v_4) =max(28,1+28) =29
▪ V[5][9], here i=5,j=9, ω_5= 7, v_4=28 so j>=ω_5 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[5][9] =max(V[4][9],V[4][2]+v_4) =max(29,6+28) =34	▪ V[5][10], here i=5,j=10, ω_5= 7, v_4=28 so j>=ω_5 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[5][10] =max(V[4][10],V[4][3]+v_4) =max(29,7+28) =35
▪ V[5][11], here i=5,j=11, ω_5= 7, v_4=28 so j>=ω_5 use V[i][j]=max(V[i-1][j], V[i-1][j- ω_i]+v_i) V[5][11] =max(V[4][11],V[4][3]+v_4) =max(40,7+28) =40	

(7) All point shortest path Floyd algorithm

Algorithm

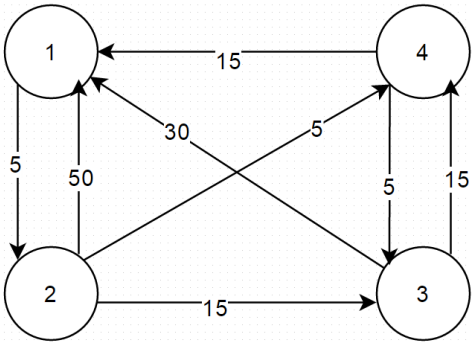
```
function Floyd(L[1..n, 1..n]) :array [1..n, 1..n]
 array D[1..n, 1..n]
 D ← L
 for k ← 1 to n do
      for i ← 1 to n do
          for j ← 1 to n do
            D[i,j] ← min(D[i,j], D[i,k]+ D[k,j])
return D
```

- We construct a matrix *D* that gives the length of shortest path between each pair of nodes.
- The algorithm initializes *D* to *L*, that is, to the direct distances between nodes. It then does n iterations, after iteration k, *D* gives length of the shortest paths that only use nodes in {1, 2… k} as intermediate nodes.
- After n iterations, *D* therefore gives the length of shortest paths using any of the nodes in *N* as an

intermediate node.

- If D_k represents the matrix D after k^{th} iteration it can be implemented by
  `Dk [i,j] = min(Dk-1 [i,j], Dk-1 [i,k]+ Dk-1 [k,j])`
- We use principle of optimality to compute length from i to j passing through k.

Example



$$D_0=\begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

- Creating matrix D_0 contains distance between each node with '0' as intermediate node.

$$D_1=\begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

- Updating matrix D_1 which contains distance between each node with '1' as intermediate node.
- Update distance if minimum distance value than existing distance value found.
- Here distance (3, 2) is updated from ∞ to 35 and distance (4, 2) is updated from ∞ to 20.

$$D_2=\begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

- Updating matrix D_2 contains distance between two nodes with '2' as intermediate node.
- Update distance if minimum distance value than existing distance value found.
- Here distance (1, 3) is updated from ∞ to 20 and distance (1, 4) is updated from ∞ to 10.

$$D_3=\begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

- Updating matrix D_3 contains distance between two nodes with '3' as intermediate node.
- Update distance if minimum distance value than existing distance value found.
- Here distance (2, 1) is updated from 50 to 45

$$D_4=\begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

- Updating matrix D_4 contains distance between two nodes with '4' as intermediate node.
- Update distance if minimum distance value than existing distance value found.
- Here distance (1, 3) is updated from 20 to 15; distance (2, 1) is updated from 45 to 20 and distance (2, 3) is updated from 15 to 10

(8) Matrix chain multiplication using dynamic programming

- Matrix table M[i][j] stores cost of multiplications
- Matrix chain orders table S[i][j] stores order of multiplication
- To generate M[i][j] use following steps

```
Step-1 if i = j then M[i][j] = 0
Step-2 if i < j then M[i][j] = min( M[i][k]+M[k+1][j]+P[i-1]* P [k]* P [j]) with i ≤ k < j
```

Example:

Here dimensions are $p_0=5, p_1=4, p_2=6, p_3=2, p_4=7$

▪ For i = j M[i][j]=0
M[1][1]=0,M[2][2]=0,M[3][3]=0, M[4][4]=0

For i > j we do not calculate cost so M[i][j] will be blank for I > j

▪ **M[1][2],** here i < j for i=1,j=2
k=1 because i ≤ k ≤ j-1
Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)
For k=1
M[1][2]=M[1][1]+M[2][2]+p_0*p_1*p_2
= 0+0+5*4*6
=120
Put value of minimum value of M[1][2] in table M[i][j] and put value of k in table S[i][j].

M[i][j]=

	1	2	3	4
1	0			
2	-	0		
3	-	-	0	
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0			
2	-	0		
3	-	-	0	
4	-	-	-	0

M[i][j]=

	1	2	3	4
1	0	**120**		
2	-	0		
3	-	-	0	
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	**1**		
2	-	0		
3	-	-	0	
4	-	-	-	0

▪ **M[2][3],** here i < j for i=1,j=2
k=2 because i ≤ k ≤ j-1
Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)
For k=2
M[2][3]=M[2][2]+M[3][3]+p_1*p_2*p_3
= 0+0+4*6*2
=48
Put value of minimum value of M[2][3] in table

▪ **M[3][4],** here i < j for i=3,j=4
k=3 because i ≤ k ≤ j-1
Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)
For k=3
M[3][4]=M[3][3]+M[4][4]+p_2*p_3*p_4
= 0+0+6*2*7
=84
Put value of minimum value of M[3][4] in table

M[i][j] and put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120		
2	-	0	**48**	
3	-	-	0	
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1		
2	-	0	**2**	
3	-	-	0	
4	-	-	-	0

M[i][j]= and put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120		
2	-	0	48	
3	-	-	0	**84**
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1		
2	-	0	2	
3	-	-	0	**3**
4	-	-	-	0

- **M[1][3],** here i < j for i=1,j=3

k=1 or 2 because i ≤ k ≤ j-1

Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=1

M[1][3]=M[1][1]+M[2][3]+p_0*p_1*p_3

= 0+48+5*4*2

=88

For k=2

M[1][3]=M[1][2]+M[3][3]+p_0*p_2*p_3

=120+0+5*6*2

=180

Put value of minimum value of M[1][3] in table M[i][j]

And put value of k in table S[i][j]

- **M[2][4],** here i < j for i=2,j=4

k=2 or 3 because i ≤ k ≤ j-1

Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=2

M[2][4]=M[2][2]+M[3][4]+p_1*p_2*p_4

= 0+84+4*6*7

=252

For k=3

M[2][4]=M[2][3]+M[4][4]+p_1*p_3*p_4

= 48+0+4*2*7

=104

Put value of minimum value of M[1][3] in table M[i][j]

And put value of k in table S[i][j]

M[i][j]=

	1	2	3	4
1	0	120	**88**	
2	-	0	48	
3	-	-	0	84
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1	**1**	
2	-	0	2	
3	-	-	0	3
4	-	-	-	0

M[i][j]=

	1	2	3	4
1	0	120	88	
2	-	0	48	**104**
3	-	-	0	84
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1	1	
2	-	0	2	**3**
3	-	-	0	3
4	-	-	-	0

- **M[1][4],** here i < j for i=1,j=4

k=1 or 2 or 3 because i ≤ k ≤ j-1

Use M[i][j] =min(M[i][k]+M[k+1][j]+$P_{[i-1]}$*$p_{[k]}$*$p_{[j]}$)

For k=1

M[1][4]=M[1][1]+M[2][4]+p_0*p_1*p_4

= 0+104+5*4*7

=244

For k=2

M[1][4]=M[1][2]+M[3][4]+p_0*p_2*p_4

= 120+84+5*6*7

=414

For k=3

M[1][4]=M[1][3]+M[4][4]+p_0*p_3*p_4

= 88+0+5*2*7

=158

Put value of minimum value of M[1][3] in table M[i][j] and put value of k in table S[i][j]

M[i][j]=

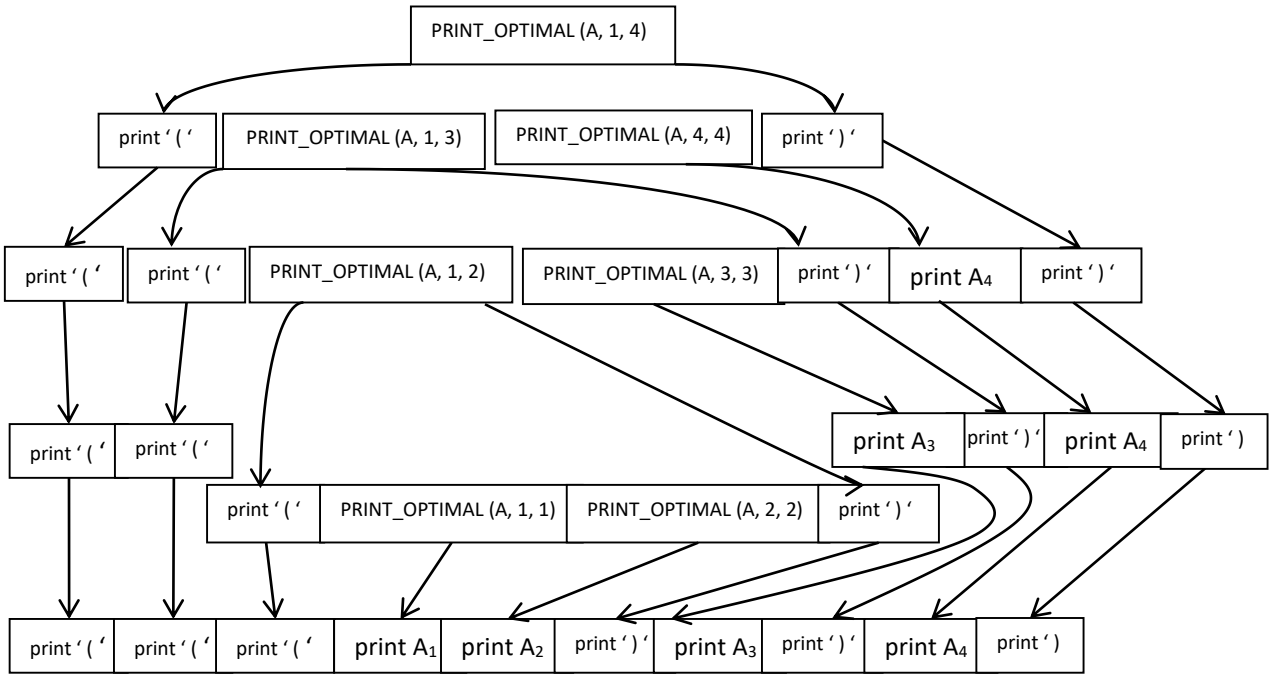
	1	2	3	4
1	0	120	88	**158**
2	-	0	48	104
3	-	-	0	84
4	-	-	-	0

S[i][j]=

	1	2	3	4
1	0	1	1	**3**
2	-	0	2	3
3	-	-	0	3
4	-	-	-	0

```
PRINT_OPTIMAL (A, i, j,S)
1   If i=j
2         print Ai
3   else
4          print '('
5          PRINT_OPTIMAL (A, i, S[i,j])
6          PRINT_OPTIMAL (A, S[i,j]+1, j)
7          print ')'
```

Here we are having 4 matrices let us denote it as A1,A2,A3,A4 so Calling PRINT_OPTIMAL (A, 1, 4)
Note :For values of k see matrix S[i][j]



(9) Longest chain subsequence using dynamic programming

Algorithm

```
LCS-LENGTH(X, Y)
 1 m ← length[X]
 2 n ← length[Y]
 3 for i ← 1 to m
 4      do c[i, 0] ← 0
 5 for j ← 0 to n
 6      do c[0, j] ← 0
 7 for i ← 1 to m
 8      do for j ← 1 to n
 9         do if xi = yj
10         then c[i, j] ← c[i - 1, j - 1] + 1
11         b[i, j] ← "↖"
12         else if c[i - 1, j] ≥ c[i, j - 1]
13         then c[i, j] ← c[i - 1, j]
14         b[i, j] ← "↑"
15         else c[i, j] ← c[i, j - 1]
16         b[i, j] ← "←"
17 return c and b
```

- We need to generate table c(1..m, 1..n) where m=length of string S1 and n= length of string S2
- b[i][j] stores directions like(←, ↑, ↖)
- To generate table c[i][j] use following steps

```
Step-1: Make c[i][0]=0 and c[0][j]=0
Step-2: if xi = yj then c[i,j] ← c[i-1,j-1]+1 and b[i,j]←"↖"
Step-3: else if c[i-1,j] ≥ c[I,j-1]   then c[i,j] ← c[i-1,j] and
        b[i,j] ← "↑"
Step-4 else c[i,j] ← c[i,j-1] and b[i,j]  ←  "←"
```

Example

Find any one Longest Common Subsequence of given two strings using Dynamic Programming.
S1=abbacdcba S2=bcdbbcaa

- Solution:
  C[i][j]=

		0	1	2	3	4	5	6	7	8
		Yj	b	c	d	b	b	c	a	a
0	Xi	0	0	0	0	0	0	0	0	0
1	a	0	0↑	0↑	0↑	0↑	0↑	0↑	↖1	↖1
2	b	0	↖1	1←	1←	↖1	↖1	1←	1↑	1↑
3	b	0	↖1	1↑	1↑	↖2	↖2	2←	2←	2←
4	a	0	1↑	1↑	1↑	2↑	2↑	2↑	↖3	↖2
5	c	0	1↑	↖2	2←	2↑	2↑	↖3	3↑	3←
6	d	0	1↑	2↑	↖3	3←	3←	3↑	3↑	3↑
7	c	0	1↑	↖2	3↑	3↑	3↑	↖4	4←	4←
8	b	0	↖1	2↑	3↑	↖4	↖4	4↑	4↑	4↑
9	a	0	1↑	2↑	3↑	4↑	4↑	4↑	↖5	↖5

C[i][j]=

		0	1	2	3	4	5	6	7	8
		Yj	b	C	d	b	b	c	a	a
0	Xi	0	0	0	0	0	0	0	0	0
1	a	0	0↑	0↑	0↑	0↑	0↑	0↑	↖1	↖1
2	b	0	↖1	1←	1←	↖1	↖1	1←	1↑	1↑
3	b	0	↖1	1↑	1↑	↖2	↖2	2←	2←	2←
4	a	0	1↑	1↑	1↑	2↑	2↑	2↑	↖3	↖3
5	c	0	1↑	↖2	2←	2↑	2↑	↖3	3↑	3↑
6	d	0	1↑	2↑	↖3	3←	3←	3↑	3↑	3↑
7	c	0	1↑	↖2	3↑	3↑	3↑	↖4	4←	4←
8	b	0	↖1	2↑	3↑	↖4	↖4	4↑	4↑	4↑
9	a	0	1↑	2↑	3↑	4↑	4↑	4↑	↖5	↖5

b c d c a

Longest common subsequence = bcdca, Length =5

(9) Difference Greedy Method and Dynamic Programming

Greedy Method

- We make greedy choice
- Solve the sub-problem after choice is made
- The choice we make may depend on previous choices
- The choice is independent of the solutions to sub-problems.
- Some problems can be solved effectively
- The optimum selection is without revising previously generated solutions

Dynamic Programming

- Choice is made at each step
- The choice depends on solutions to sub-problems
- Decisions are based on al decisions made in the previous stage
- Uses bottom up approach from smaller to larger sub-problem
- More powerful,
- Applicable to wide range of application
- Considers all possible sequences in order to obtain the optimum solution

(1) Undirected and directed graph

Graph:

- A graph G consist of a non-empty set V called the set of nodes (points, vertices) of the graph, a set E which is the set of edges and a mapping from the set of edges E to as set of pairs of elements V.
- It is also convenient to write a graph as G= (V, E).
- Notice that definition of graph implied that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X ϵ E is thus associated with a pair of nodes (U,V) where U,V ϵ V then we says that edge X connect U and V.

Directed Graph:

- A graph in which every edge is directed is called directed graph or digraph.

Undirected Graph:

- A graph in which every edge is undirected is called directed graph or digraph.

(2) Traversing Graph/Tree.

The following are the techniques for traversing the trees:

1) Preorder

   To traverse the tree in preorder, perform the following steps:

   i) Visit the root.
   ii) Traverse the left sub tree in preorder.
   iii) Traverse the right sub tree in preorder.

2) In order
   i) Traverse the left sub tree in in order.
   ii) Visit the root.
   iii) Traverse the right sub tree in in order.

3) Post order
   i) Traverse the left sub tree in Post order.
   ii) Traverse the right sub tree in Post order.
   iii) Visit the root

- The time T (n) needed to explore a binary tree containing n nodes is in **Θ (n).**

(3) Depth-First Search of graph

- Let G = (N, A) be an undirected graph all of whose nodes we wish to visit.
- Suppose it is somehow possible to mark a node to show it has already been visited.
- To carry out a depth-first traversal of the graph, choose any node v Є N as the starting point.
- Mark this node to show it has been visited.
- Next, if there is a node adjacent to v that has not yet been visited, choose this node as a new starting point and call the depth-first search procedure recursively.
- On return from the recursive call, if there is another node adjacent to v that has not been visited, choose this node as the next starting point, and call the procedure recursively once again, and so on.
- When all the nodes adjacent to v are marked, the search starting at v is finished.

- If there remain any nodes of G that have not been visited, choose any one of them as a new starting point, and call the procedure yet again.
- Continue thus until all the nodes of G are marked.
- Here is the recursive algorithm.

procedure dfsearch(G)

for each v Є N **do** mark[v] ← not-visited

for each v Є N **do**

if mark[v] ≠ visited **then** *dfs*(v)

procedure dfs(v)

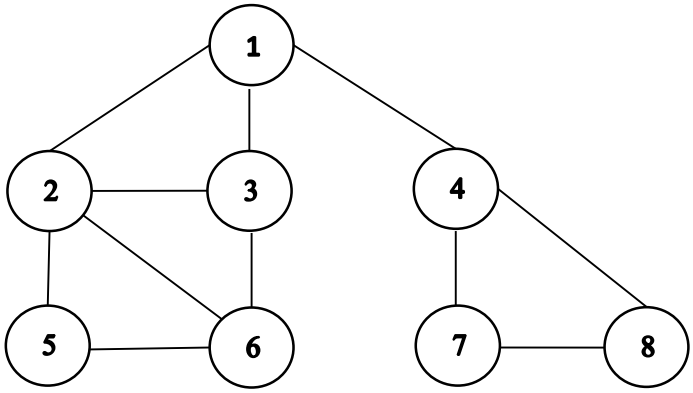
{Node v has not previously been visited}

mark[v] ← visited

for each node w adjacent to v **do**

if mark[w] ≠ visited **then** *dfs*(w)

Example:



Depth-first search progresses through the graph

1.	*dfs* (1)	initial call
2.	*dfs* (2)	recursive call
3.	*dfs* (3)	recursive call
4.	*dfs* (6)	recursive call
5.	*dfs* (5)	recursive call ; progress is blocked
6.	*dfs* (4)	a neighbor of node 1 has not been visited
7.	*dfs* (7)	recursive call
8.	*dfs* (8)	recursive call
9.	there are no more nodes to visit.	recursive call; progress is blocked

- The execution time is in **Θ (max (a, n))**
- Depth first traversal of a connected graph associates a spanning tree of a graph.
- If the graph being explored is not connected, a DFS associates to it not a single tree but a forest of trees, one for each connected component of the graph.

(4) Breadth-First Search of graph

- Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s.
- It computes the distance (smallest number of edges) from s to each reachable vertex.
- It also produces a "breadth-first tree" with root s that contains all reachable vertices.
- For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.
- Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k + 1.
- The algorithm for breadth-first search is as follows:

procedure search(G)

for each v ε N **do** mark[v] ← not visited
for each v ε N **do**
    if mark[v] ≠ visited **then** { dfs2 or bfs } (v)

procedure bfs(v)

Q ← empty-queue
mark[v] ← visited
enqueue v into Q
while Q is not empty **do**
    u ← first(Q)
    dequeue u from Q
    for each node w adjacent to u **do**
        if mark[w] ≠ visited **then** mark[w] ← visited
            enqueue w into Q

- Breadth first search in not naturally recursive. To understand the differences and similarities, let us first consider the non-recursive formulation of DFS algorithm.
- Let stack be a data type allowing two operations PUSH and POP. It handles elements in LIFO odder.
- The function top denotes the first element at the top of the stack.

procedure dfs2(v)

P ← empty stack
mark[v] ← visited
PUSH v onto P
while P is not empty **do**
    while there exists a node w adjacent to top(P) such that

mark[w] ≠ visited **do**

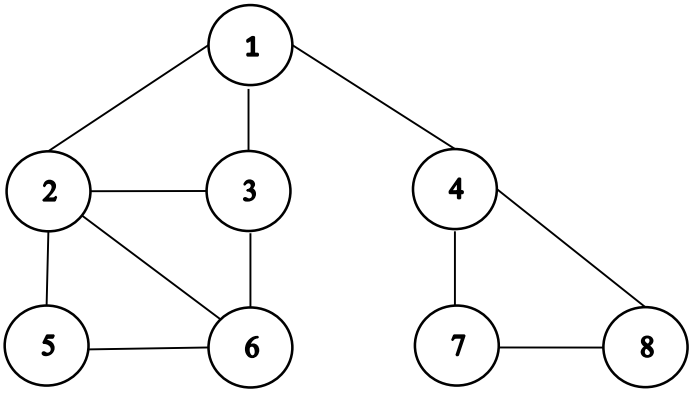
mark[w] = visited

PUSH w onto P {w becomes new top(P)}

POP P

Example:

Consider the following graph.



If node 1 is starting point

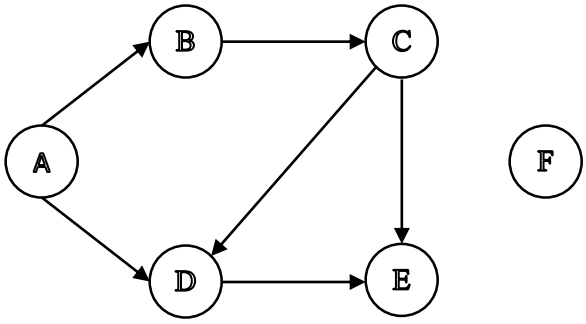
	Node Visited	Q
1.	1	2,3,4
2.	2	3,4,5,6
3.	3	4,5,6
4.	4	5,6,7,8
5.	5	6,7,8
6.	6	7,8
7.	7	8
8.	8	-

(5) Topological sorting

A topological sort of the nodes of a directed acyclic graph is the operation of arranging the nodes in order in such a way that if there exists an edge (i, j), then i precedes j in the list.

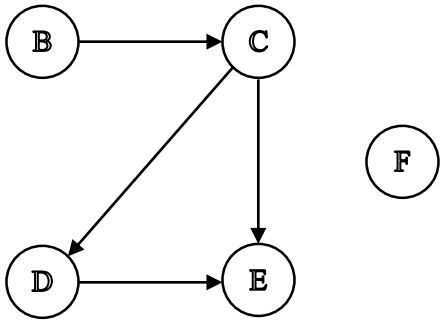
Given graph (V, E) find a linear ordering of vertices such that for all edges (v,w) v represents w in ordering.

Example:



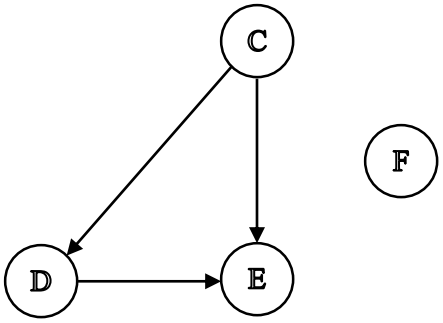
Step-1: Identifying vertices that have no incoming edge. Here, node A and F having no incoming edges.

Step-2: Select vertex delete this vertex with all its outgoing edges and put it in output list. Here, vertex A is deleted.



O/P: A

Now selecting vertex B and deleting it with its edges



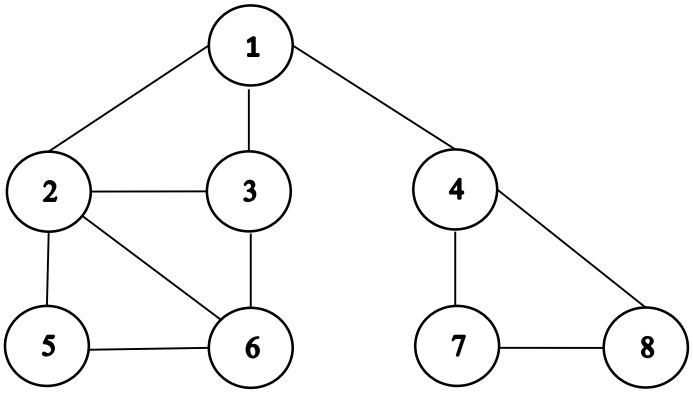
O/P: A,B

Repeating the steps gives ordered output as A,B,C,D,E,F

(6) Articulation Point

- A node v of a connected graph is an articulation point if the subgraph obtained by deleting v and the entire edges incident on v is no longer connected.
- A graph G is **biconnected or unarticulated** if it is connected and has no articulation points.
- A graph is **bicoherent or 2 – edge connected** if each articulation point is joined by at least two edges to each component of the remaining subgraph.

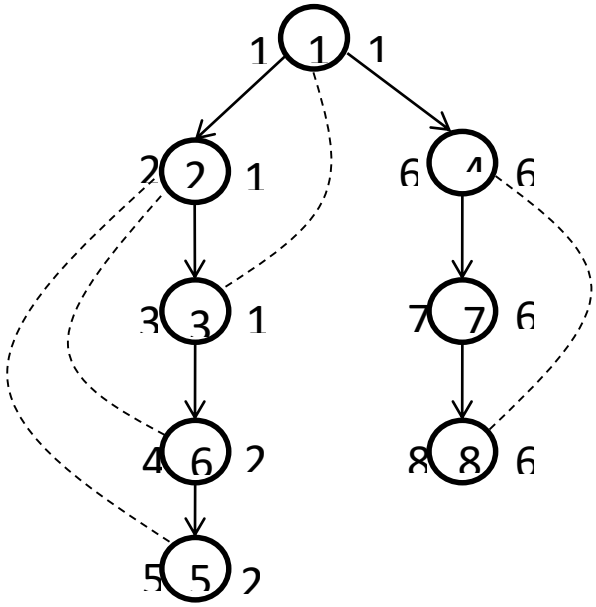
For example, node 1 is an articulation point of the graph in the above figure; If we delete it, there remain two connected components {2, 3, 5, 6} and {4, 7, 8}.



The following is the complete algorithm for finding the articulation points of an undirected graph G:

1) Carry out a depth-first search in G, starting from any node. Let T be the tree generated by this search, and for each node v of G, let prenum[v] be the number assigned by the search.
2) Traverse T in postorder. For each node v visited, calculate highest[v] as the minimum of
   (a) prenum [v];
   (b) prenum [w] for each node w such that there is an edge {v, w} in G with no corresponding edge in T;
   (c) highest [x] for every child x of v.
3) Determine the articulation paints of G as follows.
   (a) The root of T is an articulation point if and only if it has more than one child.
   (b) Any other node v is an articulation point if and only if it has a child x such that highest[x] ≥ prenum[v].

Following figure shows the depth first tree of the given graph with prenum on left and highest on the right of every node.



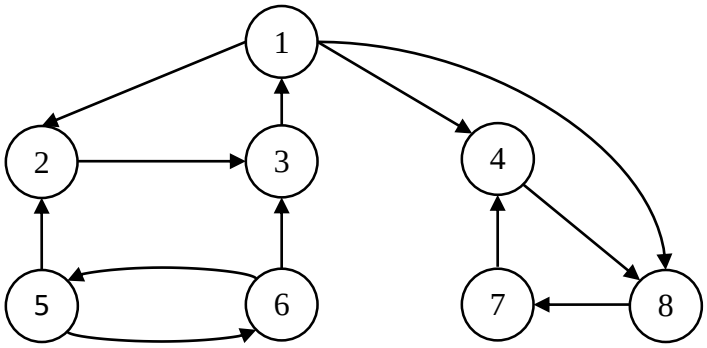
Articulation points:

Step 3 (a) Root of T has two child nodes so root (node 1) is an articulation point.

Step 3 (b) For node 4, highest [5] ≥ prenum [4] so node 4 is an articulation point.

(7) Connected Components

- A directed graph is strongly connected if there exists a path from u to v and also a path from v to u for every distinct pair of nodes u and v.
- If a directed graph is not strongly connected, we are interested in the largest sets of nodes such that the corresponding sub graphs are strongly connected.
- Each of these sub graphs is called a strongly connected component of the original graph. In the graph of Figure, for instance, nodes {1, 2, 3} and the corresponding edges form a strongly connected component. Another component corresponds to the nodes {4, 7, 8}.
- Despite the fact that there exist edges (1, 4) and (1, 8), it is not possible to merge these two strongly connected components into a single component because there exists no path from node 4 to node 1.



(8) Back Tracking

- In its basic form, backtracking resembles a depth first search in a directed graph.
- The graph is usually a tree or at least it does not contain cycles.
- The graph exists only implicitly.
- The aim of the search is to find solutions to some problem.
- We do this by building partial solutions as the search proceeds.
- Such partial solutions limit the regions in which a complete solution may be found.
- Generally, when the search begins, nothing is known about the solutions to the problem.
- Each move along an edge of the implicit graph corresponds to adding a new element to a partial solution. That is to narrow down the remaining possibilities for a complete solution.
- The search is successful if, proceeding in this way, a solution can be completely defined.
- In this case either algorithm may stop or continue looking for alternate solutions.
- On the other hand, the search is unsuccessful if at some stage the partial solution constructed so far cannot be completed.
- In this case the search backs up like a depth first search.
  When it gets back to a node with one or more unexplored neighbors, the search for a solution resumes.

(9) The Eight queens problem

- The classic problem of placing eight queens on a chessboard in such a way that none of them threatens any of the others.
- Recall that a queen threatens the squares in the same row, in the same column, or on the same diagonals.
- The most obvious way to solve this problem consists of trying systematically all the ways of placing eight queens on a chessboard, checking each time to see whether a solution has been obtained.
- This approach is of no practical use, even with a computer, since the number of positions we would have to check is $\binom{64}{8}$= 4426165368.
- The first improvement we might try consists of never putting more than one queen on any given row.
- This reduces the computer representation of the chessboard to a vector of eight elements, each giving the position of the queen in the corresponding row.

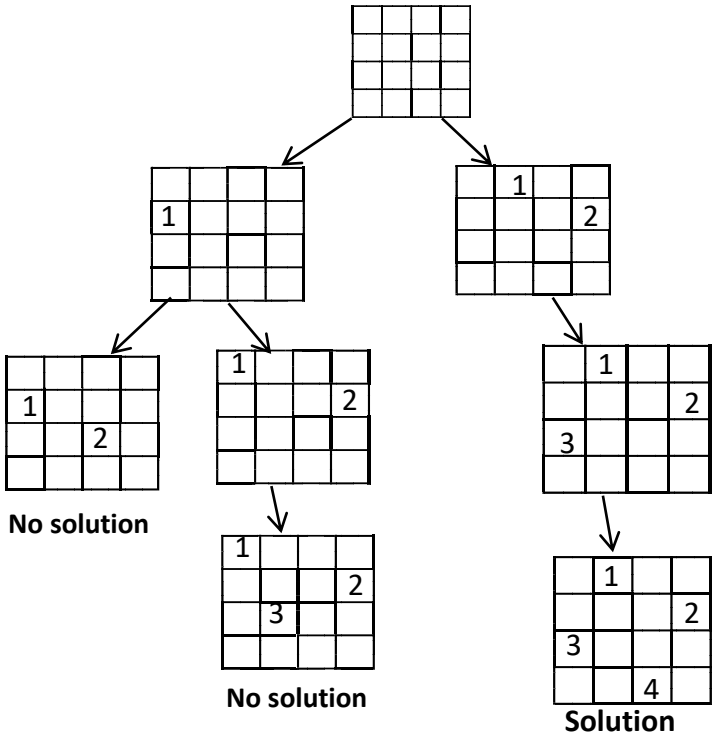
Solution using Backtracking

- Backtracking allows us to do better than this. As a first step, we reformulate the eight queen's problem as a tree searching problem. We say that a vector V[1…k] of integers between 1 and 8 is k-promising, for 0 ≤ k ≤ 8, if none of the k queens placed in positions (1, V[1]), (2, V[2]), … , (k, V[k]) threatens any of the others.
- Mathematically, a vector V is k-promising if, for every pair of integers i and j between 1 and k with i ≠ j, we have V[i] - V[j] does not belongs to {i-j, 0, j – i}. For k ≤ 1, any vector V is k-promising.
- Solutions to the eight queens' problem correspond to vectors that are 8-promisng.
- Let N be the set of k-promising vectors, 0 ≤ k ≤ 8.
- Let G = (N, A) be the directed graph such that (U, v) ∈ A if and only if there exists an integer k, 0 ≤ k ≤ 8 such that,
- U is k-promising,
- V is (k + 1)-promising, and U[i] = V[i] for every i ∈ [1..k].

The algorithm for the 8 – queens problem is given as follows:

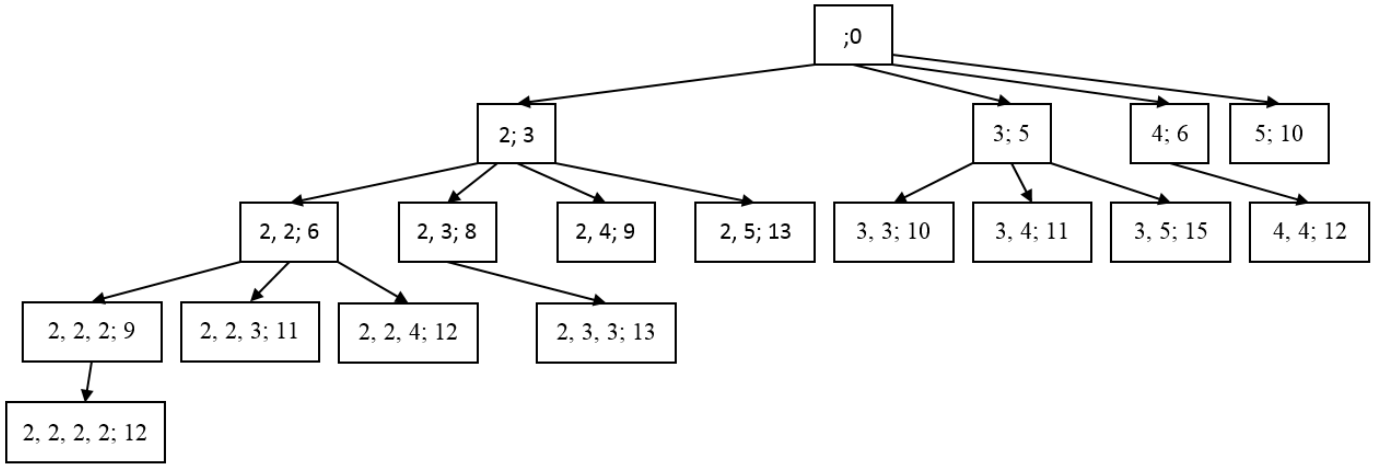
```
procedure queens (k, col, diag45, diag135)
        {sol[1..k] is k  promising,
        col = {sol[i] | 1 ≤ i ≤k},
        diag45 = {sol[i] – i + 1 | 1 ≤ i ≤ k}, and
        diag135 = {sol[i] + i – 1 | 1 ≤ i ≤ k}}
        if k = 8 then {an 8-promising vector is a solution}
                write sol
        else {explore (k+1)  promising extensions of sol }
                for j ← 1 to 8 do
                        if j does not belongs to col and j – k does not belongs to diag45  and j + k does not
                        belongs to diag135
                        then sol[k+1] ← j
                                {sol[1..k+1] is (k+1)-promising}
                                queens(k + 1, col U {j},
                                                diag45 U {j - k}, diag135 U {j + k})
```

Solution of 4 queens problem



(10) Knapsack problem using back tracking

- We are given a certain number of objects and a knapsack.
- We shall suppose that we have n types of object, and that an adequate number of objects of each type are available.
- This does not alter the problem in any important way. For i = 1, 2,..., n, an object of type i has a positive weight w_i and a positive value v_i.
- The knapsack can carry a weight not exceeding W.
- Our aim is to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint.
- We may take an object or to leave it behind, but we may not take a fraction of an object.
- Suppose for concreteness that we wish to solve an instance of the problem involving four types of objects, whose weights are respectively 2, 3, 4 and 5 units, and whose values are 3, 5, 6 and 10. The knapsack can carry a maximum of 8 units of weight.
- This can be done using backtracking by exploring the implicit tree shown below.



0/1 knapsack space tree

- Here a node such as (2, 3; 8) corresponds to a partial solution of our problem.
- The figures to the left of the semicolon are the weights of the objects we have decided to include, and the figure to the right is the current value of the load.
- Moving down from a node to one of its children corresponds to deciding which kind of object to put into the knapsack next. Without loss of generality we may agree to load objects into the knapsack in order of increasing weight.
- Initially the partial solution is empty.
- The backtracking algorithm explores the tree as in a depth-first search, constructing nodes and partial solutions as it goes.
- In the example, the first node visited is (2;3), the next is (2,2;6), the third is (2,2,2;9) and the fourth (2,2,2.2; 12).
- As each new node is visited, the partial solution is extended.
- After visiting these four nodes, the depth-first search is blocked: node (2, 2, 2, 2; 12) has no unvisited successors (indeed no successors at all), since adding more items to this partial solution would violate the capacity constraint.
- Since this partial solution may turn out to be the optimal solution to our instance, we memorize it.
- The depth-first search now backs up to look for other solutions.
- At each step back up the tree, the corresponding item is removed from the partial solution.
- In the example, the search first backs up to (2, 2, 2; 9), which also has no unvisited successors; one step further up the tree, however, at node (2, 2; 6), two successors remain to be visited.
- After exploring nodes (2, 2, 3; 11) and (2, 2, 4; 12), neither of which improves on the solution previously memorized, the search backs up one stage further, and so on.
- Exploring the tree in this way, (2, 3, 3; 13) is found to be a better solution than the one we have, and later (3, 5; 15) is found to be better still.
- Since no other improvement is made before the search ends, this is the optimal solution to the instance.
- Algorithm can be given as follows:

function backpack(i, r)

{Calculates the value of the best load that can be constructed using items of types i to n and whose total weight does not exceed r }

b ← 0

{Try each allowed kind of item in turn}

for k ← i to n do

if w[k] ≤ r then

b ← max(b, v[k] + backpack(k, r - w[k]))

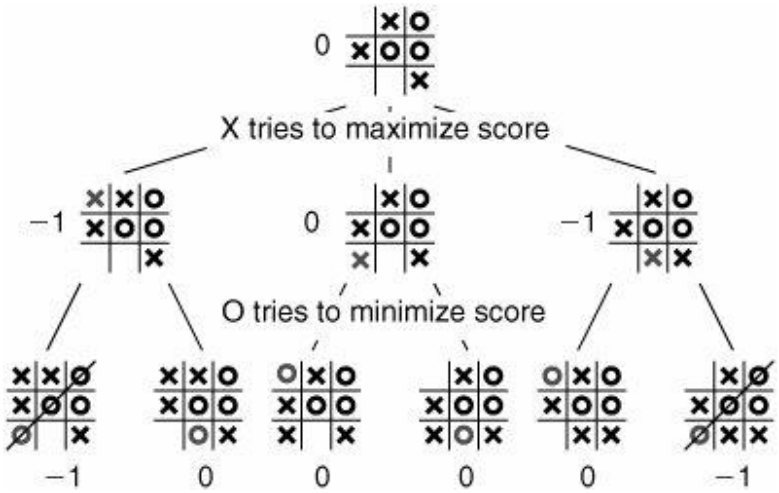
return b

Now to find the value of the best load, call backpack (1, W).

(11) Minmax Principle

- Sometimes it is impossible to complete a search due to large number of nodes for example games like chess.
- The only solution is to be content with partial solution.
- Minmax is a heuristic approach and used to find move possibly better than all other moves.
- Whichever search technique we use, the awkward fact remains that for a game such as chess a complete search of the associated graph is out of the question.
- In this situation we have to be content with a partial search around the current position.
- This is the principle underlying an important heuristic called Minimax.
- Minimax (sometimes Minmax) is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario.
- Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision making in the presence of uncertainty.
- A Minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game.
- A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves.
- Although this heuristic does not allow us to be certain of winning whenever this is possible, it finds a move that may reasonably be expected to be among the best moves available, while exploring only part of the graph starting from some given position.
- Exploration of the graph is normally stopped before the terminal positions are reached, using one of several possible criteria, and the positions where exploration stopped are evaluated heuristically.
- In a sense, this is merely a systematic version of the method used by some human players that consists of looking ahead a small number of moves.

Example: Tic tac toe



(12) Branch and Bound Travelling Salesman Problem

Branch and Bound

- Set up a bounding function, which is used to compute a bound (for the value of the objective function) at a node on a state-space tree and determine if it is promising.
- Promising (if the bound is better than the value of the best solution so far): expand beyond the node.
- Non-promising (if the bound is no better than the value of the best solution so far): not expand beyond the node (pruning the state-space tree).

Traveling Salesman Problem

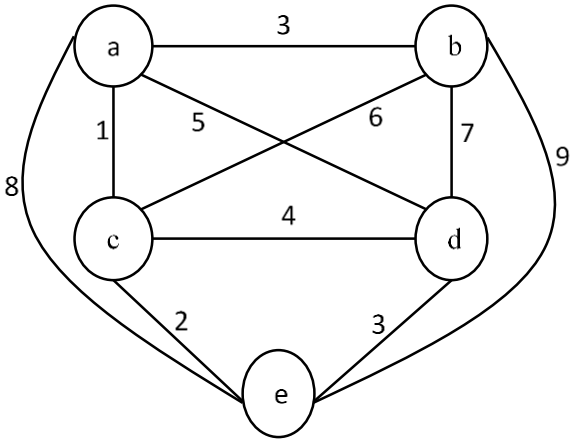
Construct the state-space tree:

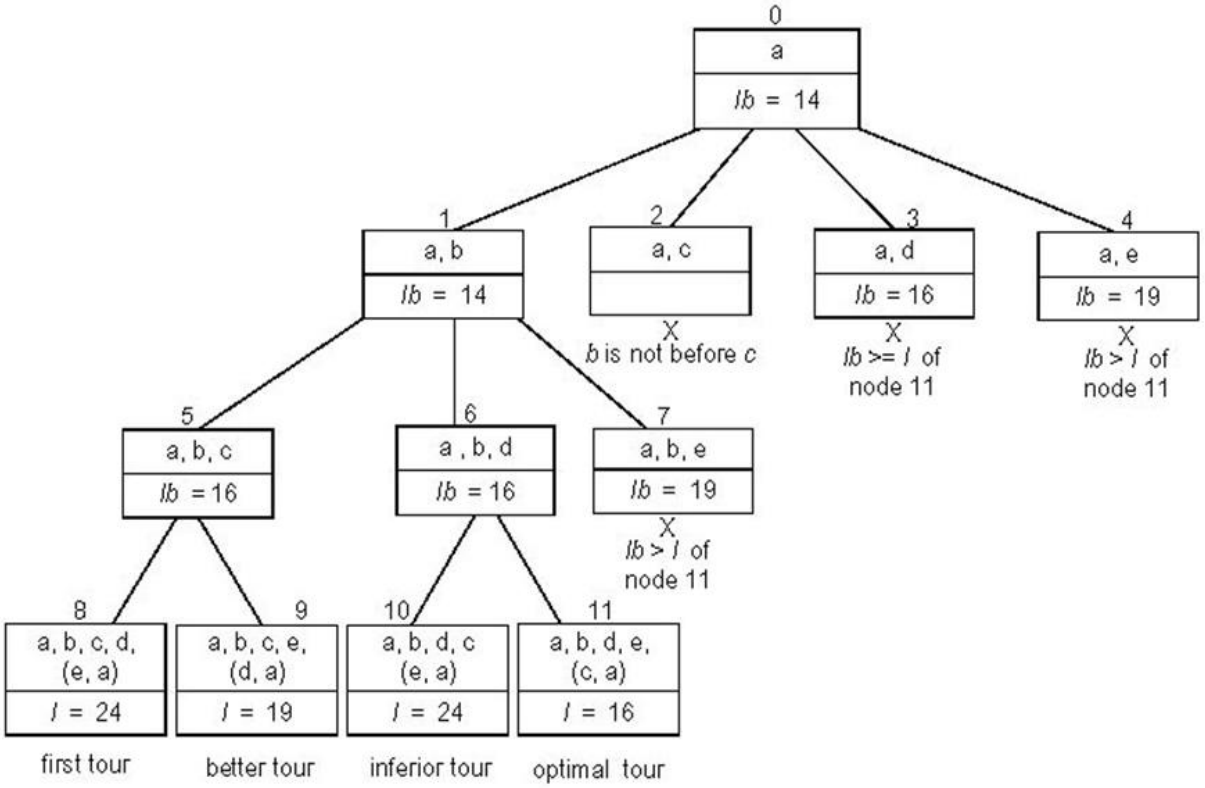
- **A node = a vertex:** a vertex in the graph. A node that is not a leaf represents all the tours that start with the path stored at that node; each leaf represents a tour (or non-promising node).
- **Branch-and-bound:** we need to determine a lower bound for each node.
  - For example, to determine a lower bound for node [1, 2] means to determine a lower bound on the length of any tour that starts with edge 1—2.
- Expand each promising node, and stop when all the promising nodes have been expanded. During this procedure, prune all the non-promising nodes.
  - **Promising node:** the node's lower bound is less than current minimum tour length.
  - **Non-promising node:** the node's lower bound is NO less than current minimum tour length.
- Because a tour must leave every vertex exactly once, a lower bound on the length of a tour is b (lower bound) minimum cost of leaving every vertex.
- The lower bound on the cost of leaving vertex v_1 is given by the minimum of all the nonzero entries in row 1 of the adjacency matrix.
- The lower bound on the cost of leaving vertex v_n is given by the minimum of all the nonzero entries in row n of the adjacency matrix.
- Because every vertex must be entered and exited exactly once, a lower bound on the length of a tour is the sum of the minimum cost of entering and leaving every vertex.
- For a given edge (u, v), think of half of its weight as the exiting cost of u, and half of its weight as the

entering cost of v.

- The total length of a tour = the total cost of visiting (entering and exiting) every vertex exactly once.
- The lower bound of the length of a tour = the lower bound of the total cost of visiting (entering and exiting) every vertex exactly once.
- Calculation:
  - For each vertex, pick top two shortest adjacent edges (their sum divided by 2 is the lower bound of the total cost of entering and exiting the vertex); add up these summations for all the vertices.
- Assume that the tour starts with vertex a and that b is visited before c.

Example:





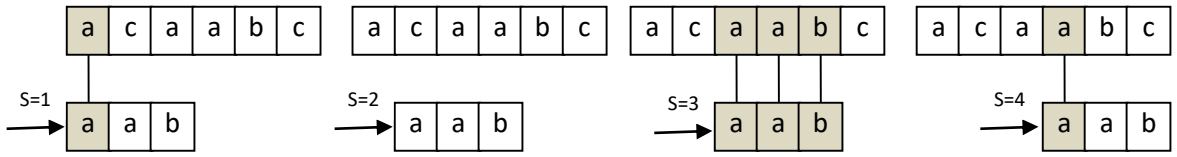
(13) The naive string matching algorithm

- The string-matching problem is defined as follows.
- We assume that the text is an array T [1…n] of length n and that the pattern is an array P [1…m] of length m ≤ n.
- We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have Σ = {0, 1} or Σ = {a, b ..., z}.
- The character arrays P and T are often called strings of characters.
- We say that pattern P occurs with shift s in text T (or, equivalently, that pattern P occurs beginning at position s + 1 in text T) if 0 ≤ s ≤ n - m and T [s + 1…s + m] = P [1…m] (that is, if T [s + j] = P[j], for 1 ≤ j ≤ m).
- If P occurs with shift s in T, then we call s a valid shift; otherwise, we call s an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.
- The naive algorithm finds all valid shifts using a loop that checks the condition P [1…m] = T [s + 1…s + m] for each of the n - m + 1 possible values of s.

```
NAIVE-STRING-MATCHER(T, P)
1       n ← length[T]
2       m ← length[P]
3       for s ← 0 to n – m do
4               if P[1,…, m] == T[s + 1,…, s + m]
5                       then print "Pattern occurs with shift" s
```

- The naive string-matching procedure can be interpreted graphically as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text, as illustrated in Figure.
- The for loop beginning on line 3 considers each possible shift explicitly.
- The test on line 4 determines whether the current shift is valid or not; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found.
- Line 5 prints out each valid shift s.

Example:



- In the above example, valid shift is s = 3 for which we found the occurrence of pattern P in text T.
- Procedure NAIVE-STRING-MATCHER takes time O ((n - m + 1) m), and this bound is tight in the worst case. The running time of NAIVE-STRING- MATCHER is equal to its matching time, since there is no preprocessing.

(14) The Rabin-Karp algorithm

- This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number.
- Let us assume that Σ = {0, 1, 2… 9}, so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix-d notation, where d = |Σ|).
- We can then view a string of k consecutive characters as representing a length-k decimal number. The character string 31415 thus corresponds to the decimal number 31,415.
- Given a pattern P [1…m], let p denotes its corresponding decimal value.
- In a similar manner, given a text T [1…n], let t_s denote the decimal value of the length-m substring T[s + 1…s + m], for s = 0, 1. . . n - m.
- Certainly, t_s = p if and only if T [s + 1…s + m] = P [1…m]; thus, s is a valid shift if and only if t_s = p.
- We can compute p in time θ(m) using Horner's rule:

  P = P[m] + 10 (P[m - 1] + 10(P[m - 2] + · · · + 10(P[2] + 10P[1])…))
- The value t_0 can be similarly computed from T [1…m] in time θ(m).
- To compute the remaining values t_1, t_2, . . . , t_{n-m} in time θ(n - m), it suffices to observe that t_{s+1} can be computed from t_s in constant time, since

  $t_{s+1} = 10(t_s – 10^{m-1}T[s + 1]) + T[s + m + 1]$
- Subtracting 10^{m-1} T[s + 1] removes the high-order digit from t_s, multiplying the result by 10 shifts the number left one position, and adding T [s + m + 1] brings in the appropriate lower order digit.
- For example, if m = 5 and t_s = 31415 then we wish to remove the high order digit T [s + 1] = 3 and bring in the new lower order digit (suppose it is T [s + 5 + 1] = 2) to obtain t_{s+1} = 10 (31415 – 10000 *3) + 2 = 14152
- The only difficulty with this procedure is that p and t_s may be too large to work with conveniently.
- There is a simple cure for this problem, compute p and the t_s's modulo a suitable modulus q.

  $t_{s+1} = (d(t_s – T[s + 1]h) + T[s + m + 1]) \bmod q$

  Where $h = d^{m-1} \pmod q$ is the value of the digit "1" in the high-order position of an m-digit text window.
- The solution of working modulo q is not perfect, however $t_s \equiv p \pmod q$ does not imply that t_s = p but if $t_s \not\equiv p \pmod q$ definitely implies $t_s \neq p$, so that shift s is invalid.
- Any shift s for which $t_s \equiv p \pmod q$ must be tested further to see whether s is really valid or it is just a spurious hit.
- This additional test explicitly checks the condition T [s + 1…s + m] = P [1…m].

Algorithm RABIN-KARP-MATCHER(T, P, d, q)

n ← length[T];
m ← length[P];
$h \leftarrow d^{m-1} \bmod q$;
p ← 0;
t_0 ← 0;

```
for i ← 1 to m do
        p ← (dp + P[i]) mod q;
        t0 ← (dt0 + P[i]) mod q
for s ← 0 to n – m do
        if p == ts then
                if P[1..m] == T[s+1..s+m] then
                        print "pattern occurs with shift s"
        if s < n-m then
        ts+1 ← (d(ts – T[s+1]h) + T[s+m+1]) mod q
```

Analysis

- RABIN-KARP-MATCHER takes **Θ(m)** preprocessing time and it matching time is **Θ(m(n – m + 1))** in the worst case.

Example:

Given T = 31415926535 and P = 26

We choose q = 11

P mod q = 26 mod 11 = 4

3	1	4	1	5	9	2	6	5	3	5	31 mod 11 = 9 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	14 mod 11 = 3 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	41 mod 11 = 8 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	15 mod 11 = 4 equal to 4 -> spurious hit
3	1	4	1	5	9	2	6	5	3	5	59 mod 11 = 4 equal to 4 -> spurious hit
3	1	4	1	5	9	2	6	5	3	5	92 mod 11 = 4 equal to 4 -> spurious hit
3	1	4	1	5	9	2	6	5	3	5	26 mod 11 = 4 equal to 4 -> an exact match!!
3	1	4	1	5	9	2	6	5	3	5	65 mod 11 = 10 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	53 mod 11 = 9 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	35 mod 11 = 2 not equal to 4

(15) String Matching with finite automata

- Many string-matching algorithms build a finite automaton that scans the text string *T* for all occurrences of the pattern *P*.
- We begin with the definition of a finite automaton. We then examine a special string-matching automaton and show how it can be used to find occurrences of a pattern in a text.
- Finally, we shall show how to construct the string-matching automaton for a given input pattern.

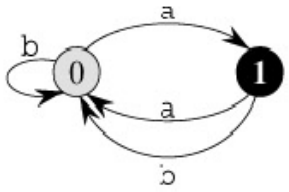
Finite automata

A ***finite automaton*** *M* is a 5-tuple (*Q*, *q*0, *A*, Σ, *δ*), where

- *Q* is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- *A* C *Q* is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
  δ is a function from *Q* × Σ into *Q*, called the ***transition function*** of *M*.
- The finite automaton begins in state q0 and reads the characters of its input string one at a time.
- If the automaton is in state q and reads input character a, it moves ("makes a transition") from state q to state δ(q, a).
- Whenever its current state q is a member of A, the machine M is said to have accepted the string read so far. An input that is not accepted is said to be rejected.
- Following Figure illustrates these definitions with a simple two-state automaton.

state	input a	input b
0	1	0
1	0	0

(a)



(b)

String-matching automata

- There is a string-matching automaton for every pattern *P*; this automaton must be constructed from the pattern in a preprocessing step before it can be used to search the text string.
- In our example pattern *P* = abababa.
- In order to properly search for the string, the program must define a **suffix function (σ)** which checks to see how much of what it is reading matches the search string at any given moment.
  $\sigma(x) = \max \{k : P_k \sqsupset x\}$
  P= ababa
  P_1=a
  P_2=ab
  P_3=aba
  P_4=abab
  σ (abababa)=aba {here aba is suffix of pattern P}
- We define the string-matching automaton that corresponds to a given pattern *P*[1,…,*m*] as follows.

- The state set Q is {0, 1 . . . m}. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function δ is defined by the following equation, for any state q and character a:

$$\delta(q, \alpha) = \sigma(P_q \alpha)$$

ALGORITHM FINITE-AUTOMATON-MATCHER(*T*, *δ*, *m*)

- $n \leftarrow length[T]$
- $q \leftarrow 0$
- **for** $i \leftarrow 1$ **to** n **do**
- $q \leftarrow \delta(q, T[i])$
- **if** $q == m$

  then print "Pattern occurs with shift" $i - m$

ALGORITHM COMPUTE-TRANSITION-FUNCTION(*P*, Σ)

- $m \leftarrow length[P]$
- **for** $q \leftarrow 0$ to m **do**
- **for** each character $\alpha \in \Sigma$ **do**
- $k \leftarrow \min(m + 1, q + 2)$
- **repeat** $k \leftarrow k - 1$
- **until** $P_k \sqsupset P_q \alpha$
- $\delta(q, \alpha) \leftarrow k$

  return δ

- This procedure computes $\delta(q, \alpha)$ in a straightforward manner according to its definition.
- The nested loops beginning on lines 2 and 3 consider all states q and characters α and lines 4-7 set $\delta(q, a)$ to be the largest k such that $P_k \sqsupset P_q \alpha$. The code starts with the largest conceivable value of k, which is min (m, q + 1), and decreases k until $P_k \sqsupset P_q \alpha$.
- Time complexity for string matching algorithm

Algorithm	Preprocessing time	Matching time
Naive	0	O((n – m + 1)m)
Rabin-Karp	Θ (m)	O((n – m + 1)m)
Finite automaton	O (m|∑|)	Θ (n)

Example:

	1	2	3	4	5	6	7	8	9	10	11
T=	a	b	a	b	a	b	a	c	a	b	a

	1	2	3	4	5	6	7
P=	a	b	a	b	a	b	a

q=0 α=a k=2 $P_2 \supset P_0\ \alpha$

ab⊐ϵa {here 'ab' is not suffix of 'ϵa' so k=k-1}

k=1 $P_1 \supset P_0\ \alpha$

a⊐ϵa { here 'a' suffix of 'ϵa' so make entry in transition table}

α=b k=2 $P_2 \supset P_0\ \alpha$

ab⊐ϵb {here 'ab' is not suffix of 'ϵb' so k=k-1}

k=1 $P_1 \supset P_0\ \alpha$

a⊐ϵb { here 'a' is not suffix of 'ϵb' so k=k-1}

k=0 $P_0 \supset P_0\ \alpha$

ϵ⊐ϵb { here 'ϵ' suffix of 'ϵa' so make entry in transition table}

α=c k=2 $P_2 \supset P_0\ \alpha$

ab⊐ϵc {here 'ab' is not suffix of 'ϵc' so k=k-1}

k=1 $P_1 \supset P_0\ \alpha$

a⊐ϵc { here 'a' is not suffix of 'ϵc' so k=k-1}

k=0 $P_0 \supset P_0\ \alpha$

ϵ⊐ϵc { here 'ϵ' suffix of 'ϵc' so make entry in transition table}

- Repeat this procedure for q=0 to 7 we can get transition function table then pattern can be matched using this table

	input			
State	a	b	c	*P*
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(16) The Knuth-Morris-Pratt algorithm

- Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.
- A matching time of O(n) is achieved by avoiding comparisons with elements of 'T' that have previously been involved in comparison with some element of the pattern 'P' to be matched. i.e., backtracking on the text 'S' never occurs.

The prefix function, π

- The prefix function, π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern 'P'. In other words, this enables avoiding backtracking on the text 'T'.

The KMP Matcher

- With text 'T', pattern 'P' and prefix function 'π' as inputs, finds the occurrence of 'P' in 'T' and returns the number of shifts of 'P' after which occurrence is found.

KMP-MATCHER(T, P)

```
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                    //Number of characters matched.
 5  for i ← 1 to n           //Scan the text from left to right.
 6    do while q > 0 and P[q + 1] ≠ T[i]
 7         do q ← π[q]    //Next character does not match.
 8       if P[q + 1] = T[i]
 9         then q ← q + 1     //Next character matches.
10        if q = m            //Is all of P matched?
11          then print "Pattern occurs with shift" i - m
12               q ← π[q]    //Look for the next match.
```

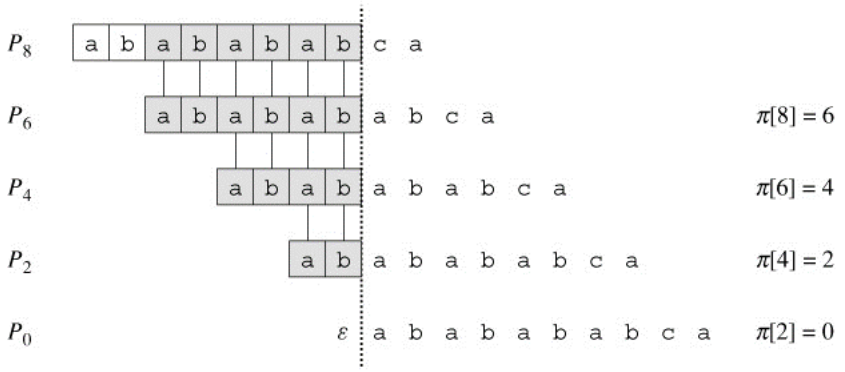
COMPUTE-PREFIX-FUNCTION(P)

```
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5    do while k > 0 and P[k + 1] ≠ P[q]
 6         do k ← π[k]
 7       if P[k + 1] = P[q]
 8         then k ← k + 1
 9       π[q] ← k
10  return π
```

Example:

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

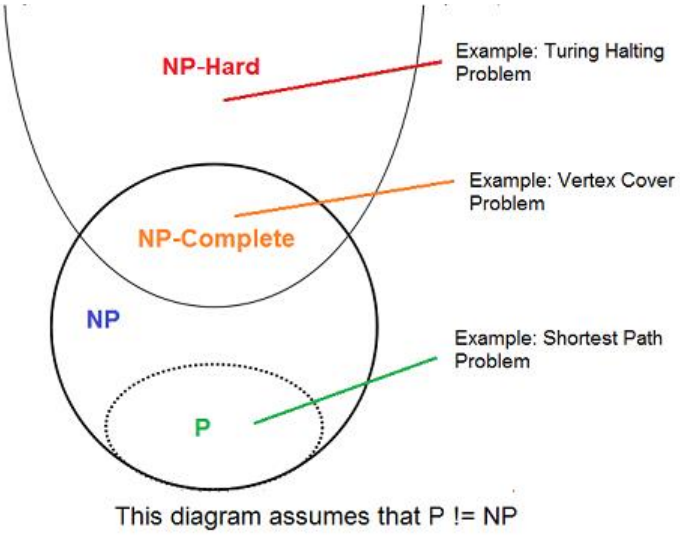
(a)



(b)

- For the pattern P = abababababca and q = 8. Figure (a) the π function for the given pattern.
- Since π [8] = 6, π [6] = 4, π [4] = 2, and π [2] = 0, by iterating π we obtain π*[8] = {6, 4, 2, 0}. Figure (b)
- We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_8; this happens for k = 6, 4, 2, and 0.
- In the figure, the first row gives P and the dotted vertical line is drawn just after P_8.
- Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_8.
- Successfully matched characters are shown shaded Vertical lines connect aligned matching characters.
- Thus, {k: k < q and $P_k \sqsupset P_q$} = {6, 4, 2, 0}. π*[q] = {k : k < q and $P_k \sqsupset P_q$ } for all q.

(17) P and NP Problems



P Problems

- The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.
  Fundamental complexity classes.
- The class of problems that can be solved in polynomial time is called P class.
- These are basically tractable. Few examples of P class problems are,
  1. A set of decision problems with yes/no answer.
  2. Calculating the greatest common divisor.
  3. Sorting of n numbers in ascending or descending order.
  4. Searching of an element from the list, etc.

NP Problems

- NP = Non-Deterministic polynomial time
- NP means verifiable in polynomial time
- The class NP consists of those problems that are "verifiable" in polynomial time.
- What we mean here is that if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.
- NP is one of the most fundamental classes of problems in computational complexity theory.
- The abbreviation NP refers to non-deterministic polynomial time.
- These problems can be solved in non-deterministic polynomial time.
- For example
  1. Knapsack problem $O(2^{n/2})$
  2. Travelling salesperson problem $(O(n^2 2^n))$.
  3. Graph coloring problem.
  4. Hamiltonian circuit problems, etc...

(18) NP hard and NP Complete problems

NP Complete

- The class of problems "NP-complete stands for the sub-lass od decision problems in NP that are hardest.
- The class NP-complete is abbreviated as NPC and comes from:
  -Non-deterministic polynomial
  -Complete-"Solve one, solve them all"

A decision problem L is said to be NP-Complete if:

(i) L is in NP that means any given solution to this problem can be verified quickly in polynomial time.
(ii) Every problem is NP reducible to L in polynomial time.

- It means that if a solution to this L can be verified in polynomial time then it can be shown to be in NP.
- A problem that satisfies second condition is said to be NP-hard that will be examined in recent.
- Informally it is believed that if a NPC problem has a solution in polynomial time then all other NPC problems can be solved in polynomial time.
- The list given below is the examples of some well-known problems that are NP-complete when expressed as decision problems.
  i. Boolean circuit satisfiability problem(C-SAT).
  ii. N-puzzle problem.
  iii. Knapsack problem.
  iv. Hamiltonian path problem.
  v. Travelling salesman problem.
  vi. Sub graph isomorphism problem.
  vii. Subnet sum problem.
  viii. Clique Decision Problem (CDP).
  ix. Vertex cover problem.
  x. Graph coloring problem.
- The following techniques can be applied to solve NPC problems and they often give rise to substantially faster algorithms:
  i. Approximation Approach.
  ii. Randomization.
  iii. Restriction.
  iv. Parameterization.
  v. Heuristics.

NP Hard

The class NP-hard written as NPH or NP-H stands for non-deterministic polynomial time hard. The class can be defined as:

i. NPH is a class of problems that are "At least as hard as the hardest problems in NP"
ii. A problem H is NP-hard if there is a NPC problem L that is polynomial time reducible to H (i.e. L

H).

iii. A problem H is said to be NPH if satisfiability reduces to H.

iv. If a NPC problem L can be solved in polynomial time by an oracle machine with an oracle for H.

- NP-hard problems are generally of the following types-decisions problems, search problems, or optimization problems.
- To prove a problem H is NP-hard, reduce a known NP-hard problem to H.
- If a NPH problem can be solved in polynomial time, then all NPC problems can also be solved in polynomial time. As a result, all NPC problems are NPH, but all NPH problems are not NPC.