ALGORITHMS - I

→ Design of Efficient Algorithms

PROBLEM :- Is a question that we seek an answer for.

① E.g.: Sort a list S of n elements in non-increasing order.

② E.g.: Determine whether the number x is in the list S of n numbers.

A problem has a set of unknown variables. These variables are typically called the "Parameters" of the problem.

For ① ⟶ S & n

For ② ⟶ S, x, n.

A problem with specified ~~prob~~ parameters is called "an instance" of a problem.

A "solution" is an answer to a specific instance of a problem.

Algorithm: An algorithm is a step by step procedure to solve each and every instance of a problem.

ALGORITHM-1: Sequencial Search.

Problem: Is the key x present in the array S of n keys.

Input: (parameters) positive integer n, an array S indexed from 1 to n and a key x

Desired Output: "location", location of x in S (0 if x is not in S)

```
void seqSearch (int n, int S[], int x, index& location){
    location = 1;
    while (location <= n && S[location] != x)
        location++;
    if (location > n)
        location = 0;
}
```

ALGORITHM 2: Exchange Sort :—

Problem: Sort n keys in non-decreasing order.

Inputs: Positive integer n, array of S keys indexed from 1 to n.

Outputs: The array S containing the keys in non decreasing order.

```
void exchangeSort (int n, int S[]){
    indexe i, j;
    for (i=1; i<= n; i++)
        for (j= i+1; j<= n; j++)
            if (S[j] < S[i])
                exchange S[i] & S[j];
}
```

ALGORITHM 3: Matrix Multiplication :—

Problem: Multiplication of two $n \times n$ matrices

Input : A positive integer n, two matrices A & B each of dimension $n \times n$

Output : A matrix C which is the product of the matrices A & B

```
void matrixMultiply (int n, int A[][], int B[][], int C[][])
{
    index i, j, k;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++) { C[i][j]=0;
            for (k=1; k<=n; k++) C[i][j]= C[i][j] + A[i,j] * B[i,j];
}       }
```

ALGORITHM-4: Binary Search

Problem: Determine whether x is in the sorted array S of n keys.

Inputs: Positive integer n, sorted (non decreasing array) S of n keys

Outputs: "location", location of x (0 is x is not in S)

```
void binSearch (int n, int S[], int x, index& location){
    index low, high, mid;
    low = 1; high = n;
    location = 0;
    while (low <= high && location == 0) {
        mid = ⌊(low + high) / 2⌋;
        if (x == S[mid]) location = mid;
        else if (x < s[mid]) high = mid - 1;
        else low = mid + 1;
    }
}
```

Comparing Algorithm 1 & Algorithm 4 in terms of their efficiency

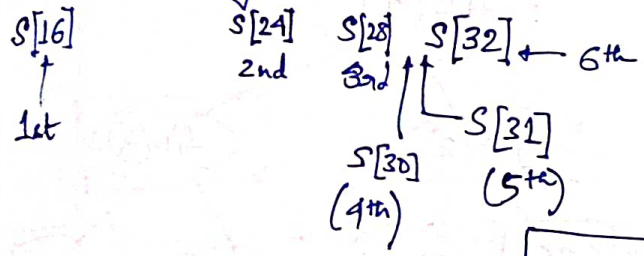
- **Algorithm 1**

  When the element is at the last position or not present.

- **Algorithm 2**

  When element is not present, x is equal to the largest element in the array.

} worst cases.



How many calls? $6 = \log_2 32 + 1$

general case: $\log_2 n$

ALGORITHM 5: Fibonacci

0, 1, 1, 2, 3, 5,

$f_0 = 0, \quad f_1 = 1$

$f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$

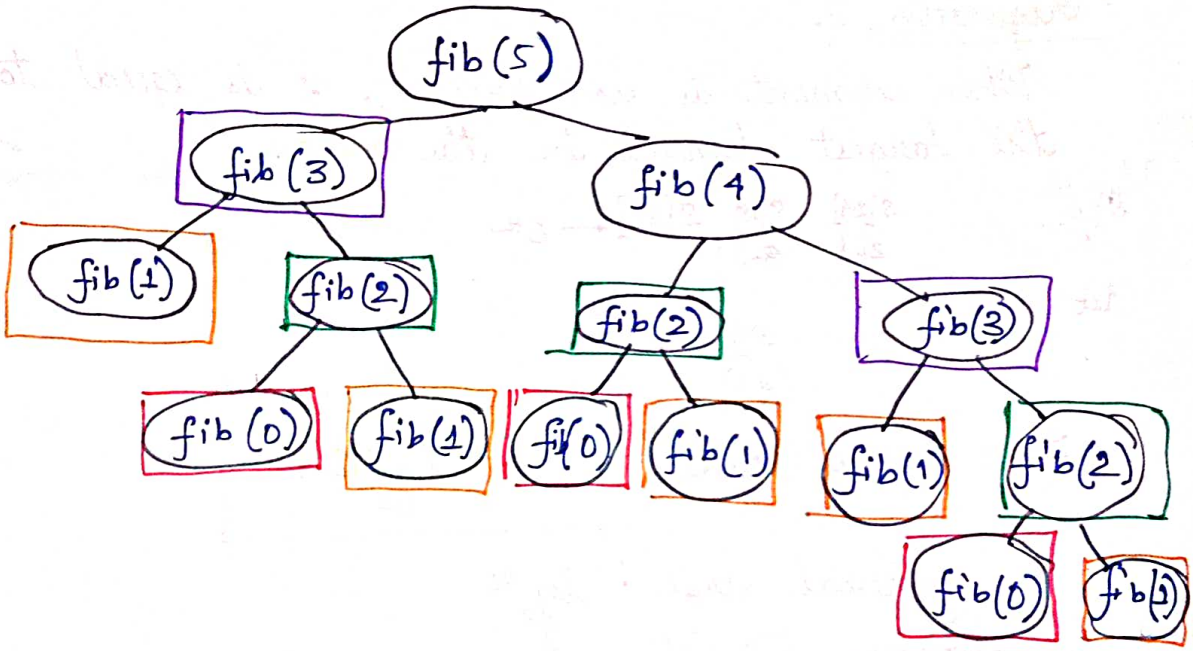
Problem: Determine the n-th term of a fibonacci sequence.

Input: a non-negative integer n

Outputs: fib, the n^{th} term of the Fibonacci sequence.

```
int fib(int n) {
    if (n <= 1) return n;
    else return fib(n-1) + fib(n-2);
}
```

Recursion Tree:



One way to identify the work done is to check the number of terms computed, when you wish to find a particular number in the fibonacci sequence.

n	# terms
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41

The number of terms "more than doubles" as you increase the value of n by two.

Let $T(n)$ be the number terms in the recursion tree for n.

$$T(n) > 2\,T(n-2)$$
$$> 2 \times 2\,T(n-4)$$
$$> \underbrace{2 \times 2 \times \cdots \times 2}_{\frac{n}{2} \text{ terms}} \times \underbrace{T(0)}_{T(0)=1}$$

$$\boxed{T(n) > 2^{\frac{n}{2}}, \quad n \geq 2}$$

Proof by induction:

Base case: $T(2) = 3 > 2^{2/2}$

$T(3) = 5 > 2^{3/2} \approx (2.83)$

Hypothesis: (Strong form of induction)

We assume $T(m) > 2^{m/2}$ $\forall\ m,\ 2 \leq m < n$

Inductive step:

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

$$T(n) > 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \quad \begin{cases} T(n-1) > 2^{(n-1)/2} \\ T(n-2) > 2^{(n-2)/2} \end{cases}$$

$$> 2 \cdot 2^{(n-2)/2} + 1 \qquad \left[\because\ 2^{(n-1)/2} > 2^{(n-2)/2}\right]$$

$$> 2 \cdot 2^{\frac{n}{2}-1}$$

$$\geq 2^{\frac{n}{2}}$$

$$\therefore T(n) > 2^{\frac{n}{2}}$$

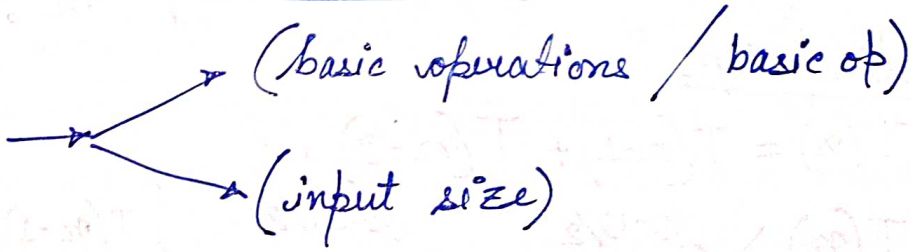
ALGORITHM 6: Efficient Version

```
int fib2(int n){
    index i;
    int f[0, ..., n];
    f[0] = 0;
    if (n > 0)
        f[1] = 0;
    for (i = 2; i <= n; ++i)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

Measure of Work done by the algorithm

- CPU cycles (Vary from machine to machine)
- No. of lines of code (instruction set to instruction set)
- Programming Language (Language to Language)

NOTATIONAL TIME:



Input Size :—

Indicator of the input size (or the number of inputs) to your algorithm.

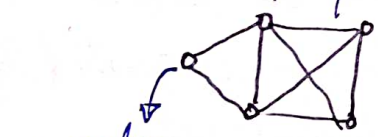
Example 1: (sequencial Search) Size of the array
$S \longrightarrow n$

Example 2: (Matrix multiplication)

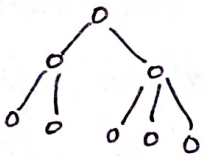
↳ Input size
n for rows
n for columns.

Is the input size always a single parameter?

- $n \times m$ matrix
- Graph, Trees.



nodes of a graph



Nodes & Edges.

Multiple input sizes.

Example 3: (Fibonacci Problem)

Fib(n) ⟶ What is the input size
⟶ 1

Basic Operations :-

Example : Algorithm 1 (Sequencial Search)

An operation that is happening a fixed number of times in the algorithm.

Comparison ← Basic Operation.

→ as bad as n times.

→ Basic ops being done input size number of times.

"Time complexity analysis" is the process of determining the number of times a basic operation is being executed as a function of the input size

This gives an idea of the efficiency of an algorithm irrespective of machine ~~cycle~~ clock cycle.

Recursive algorithm (Fibonacci): no. of terms computed $T(n) > 2^{n/2}$

Iterative version (Fibonacci): $\longrightarrow n$.

$\rightarrow$ if n is the input itself what is its size for a computer $\log_2 n$

Time Complexity Analysis

- Every case time complexity
- Worst case time complexity
- Best case time complexity
- Average case time complexity

Every case:

The basic operation is always done the same number of times for every input of size n. Denoted by $T(n)$

Example: Exchange Sort.

Basic operation: Comparison of $S[j]$ with $S[i]$

Input size: n

Complexity: How many times the algorithm passes the for-j loop.

$n \rightarrow n-1$ for-i passes

for each i. $\longrightarrow n-i$ passes of for-j loop.

$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \Rightarrow T(n)$

Every case time complexity of the matrix meltiplication?

Worst Case:

The number of basic ops that the algorithm needs to perform in the worst possible ~~scenario~~ scenario.
Denoted by $W(n)$

* If $T(n)$ exists, $W(n) = T(n)$.

Example: Worst case complexity of sequential search.

Basic op: Comparison
Input size: n
$W(n) = n$ (end of the array)

$T(n)$ for seq. search does not exist.

Best Case:

The number of basic ops. performed in the best possible scenario. Denoted by $B(n)$
* If $T(n)$ exists, $B(n) = T(n)$.
$B(n) = 1$ [first element] for seq. search.

Average Case:

Estimate the number of times the algorithm does the basic op on average

- Assume some sort of distribution on the input.

Sequential search

→ x is in the array.
- It is equally likely for x to appear anywhere inside the array.

Basic op: Comparison

Input size: n

Analysis:

1. Assume x can be anywhere in S
2. x may or may not be in S

1.

Assume that x is at the k^{th} position. The probability that x appears at this position $\frac{1}{n}$.

Expected number of basic ops for k is —

$$\frac{1}{n} + \frac{1}{n} + \cdots + \frac{1}{n} = \frac{k}{n}$$

$$A(n) = \sum_{k=1}^{n} \frac{k}{n} = \frac{1}{n} \sum_{k=1}^{n} k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

$\therefore A(n) = \frac{n+1}{2}$ when we assume x in S

x is part of S with some probability p.

x is not in S with probability: $(1-p)$

Finding x at any location $= \frac{p}{n}$.

Basic ops if x is at $k = \frac{kp}{n}$

$$A(n) = \sum_{k=1}^{p} \frac{kp}{n} + (1-p)n = n\left(1-\frac{p}{2}\right)+\frac{p}{2}$$

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

$$p = \frac{1}{2} \Rightarrow A(n) = \frac{3n}{4}+\frac{1}{4}$$

Quick-Sort:

Analysis of best case, worst case, and average case

ORDER OF COMPLEXITY

A complexity function is a special type of function that maps an integer to a non-negative real —

E.g.: $f(n) = n$
$f(n) = n^2$
$f(n) = \log_2 n$
$f(n) = 3n^2 + 4n$

$f(n) = 100n$
$f(n) = 10n + 20$
$f(n) = 100n + 25$
} Linear Functions (linear in n)

$f(n) = 10n^2$
$f(n) = 5n^2 + 2n + 10$
} Quadratic Functions (Quadratic in n)

"Pure quadratic" functions do not have a linear part

"Complete quadratic" functions have a linear part.

- A linear function is always eventually better than a quadratic function

$\frac{A}{100n}$ $\quad$ $\frac{B}{0.01n^2}$

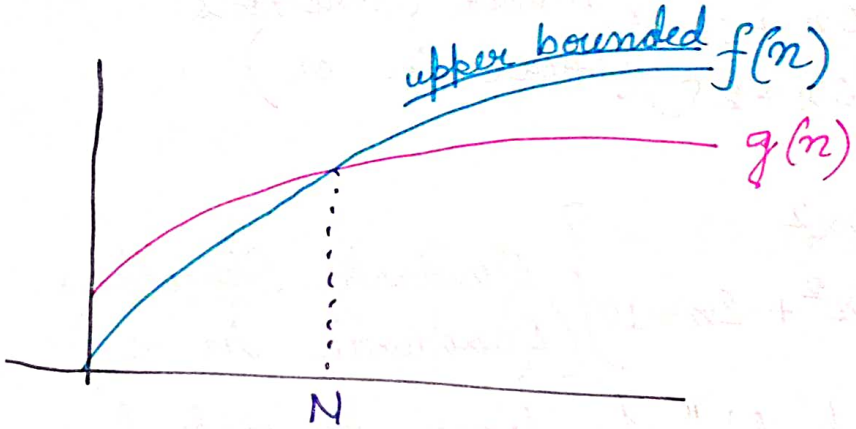
A is better than B if $n > 10^4$

For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists a real positive constant c and a non negative integer N, s.t. $n \geq N$

$$g(n) \leq cf(n)$$

$$\Rightarrow g(n) \in O(f(n))$$

Big-'O' notation or an asymptotic upper bound



Example:—

1. $5n^2 \in O(n^2)$

reason out a suitable c, N

$c = 5$, $N = 1$

$5n^2 \leq 5n^2 \quad \forall n \geq N$

2. Exchange sort —

.....

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

show that $T(n) \in O(n^2)$

$n \geq 0$ (assume $N = 0$)

$$\frac{n(n-1)}{2} \leq \frac{n(n)}{2} = \frac{1}{2} n^2$$

$c = \frac{1}{2}$, $n \geq 0$, $T(n) \in O(n^2)$

$n \in O(n^2)$

$c = 1$, $N = 1$, $n \leq n^2 \ \forall \ n \geq 1$

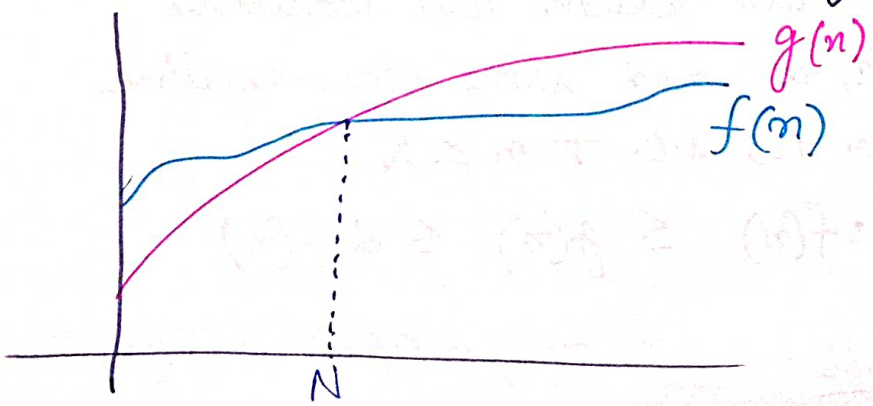
Asymptotic Lower Bound —

For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity function $g(n)$ for which there exists a positive real constant c and a non negative integer n, such that

$$g(n) \geq c f(n)$$

$$g(n) \in \Omega(f(n))$$

[Lower bounded by $f(n)$]



Example $5n^2 \in \Omega(n^2)$

$n \geq 0, \quad 5n^2 \geq 1 \times n^2$

Exchange Sort ⟶

$T(n) \in \Omega(n^2)$

For $n \geq 2$

$n - 1 \geq \frac{n}{2}$

$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$

for $c = \frac{1}{4}$ & $n \geq 2$

$T(n) \in \Omega(n^2)$

Tight Bound: Both Upper and Lower Bound.

$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

$\Theta(f(n))$ → set of complexity functions $g(n)$ for which there exists two positive real constants say c, d and some non-negative integer N, s.t. $\forall\, n \geq N$

$$cf(n) \leq g(n) \leq df(n)$$

Example — $T(n) \in O(n^2)$ & $T(n) \in \Omega(n^2)$

$\Rightarrow T(n) \in \Theta(n^2)$

For a given complexity function $f(n)$, $O(f(n))$ is the set of all complexity function $g(n)$ satisfying the following —

For every positive real constant c, there exists a non-negative integer N such that $\forall n \geq N$

$$g(n) \leq cf(n)$$

H.W.: Show that $n \in \Omega(n^2)$

$$n \in \Omega(n^2)$$
$$n \geq cn^2$$
$$\frac{1}{c} \geq n$$

for, $n \geq \frac{1}{c}$ the equality would not hold.

for $n \geq N$ it will not hold.

DIVIDE AND CONQUER

P1: Given a sorted array A of size n, search for an element with value x. (Search)

P2: Given an array of n integer elements, sort the array in the increasing order. (Sort)

P3: Given a number x, find x^n. (Powering).

You are given a problem instance of size n.

Brute Force:—

P1: Linear Search.
P2: Permute
P3: Multiply one by one.

Divide and Conquer:—

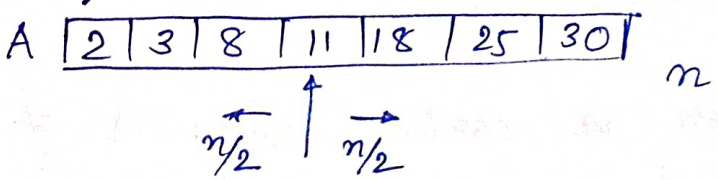
Given a problem of size n.

Divide: Break the problem into smaller instances.

Conquer: Recursively call the algorithm on each of the subproblems [Base Case: Stop when sufficiently small]

Combine: The recursive calls outputs the solutions to the subproblems. We need to combine these solutions to come up with the solution to the original instance.

P1: Binary Search.



Divide: Compare with the middle element.

Conquer: Recursively divide until there is only one element

Combine: Trivial

P3: Powering x^n

Simple algorithm: $((((x \cdot x) \cdot x) \cdot x) \cdot x) \cdots)$ $O(n)$

"If Nothing comes to mind, divide it into half"

Divide & Conquer:-

Divide:

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{if } n \text{ even} \\ x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x & \text{if } n \text{ is odd.} \end{cases}$$

Conquer: Do this recursively until $n = 1/0$

Combine: Do the multiplication.

Proposed by Pingala

$T(n) = T\left(\frac{n}{2}\right) + O(1)$ ⟶ Same as binary search. ✓

$$\boxed{T(n) = O(\log n)}$$

P2:- Sorting

① Merge Sort: $A[\quad\vdots\quad]_n$

Divide: if $n=1$, then done

sort $A\left[1 \ldots \lceil \frac{n}{2} \rceil\right]$

sort $A\left[\lceil \frac{n}{2} \rceil + 1, \ldots, n\right]$

Conquer: Recursive sorting.

Combine: $\left[\begin{array}{c}\text{sorted}\\ \frac{n}{2}\end{array}\right]$ ← merge these → $\left[\begin{array}{c}\text{sorted}\\ \frac{n}{2}\end{array}\right]$

$\left[\begin{array}{c}\text{sort}\\ n\end{array}\right]$

The merging takes $O(n)$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \longleftarrow O(n) \leq cn$$

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

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

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

$$\vdots$$

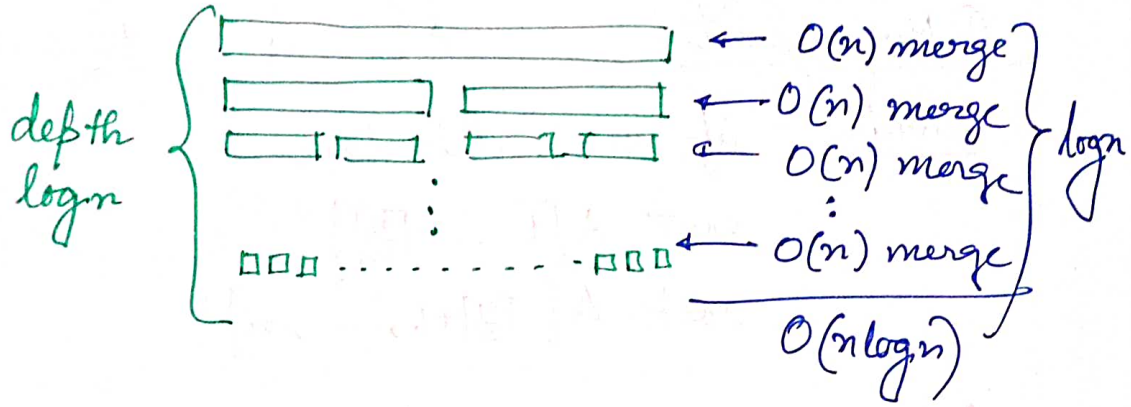
$$= 2^k T\left(\frac{n}{2^k}\right) + kcn \qquad k \approx \log_2 n$$

$$\therefore T(n) = 2^k T\left(\frac{n}{2^k}\right) + kcn$$

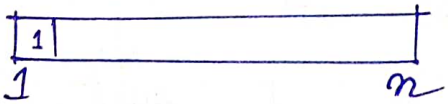
$$= n\,T(1) + cn\log n$$

$$= n \cdot O(1) + cn\log n$$

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

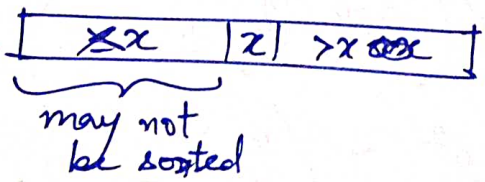


② QuickSort:



— Choose one element 'x' as pivot.

— Rearrange the array such that x comes in the sorted position and the elements $\leq x$ come to the left of x and $> x$ to the right



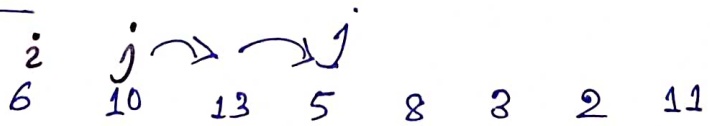
Divide: Partition the array using a pivot.

Conquer: Recursively apply Qsort ~~on~~ on both the partition.

Combine: Trivial.

```
void QuickSort (A, start, end) {
    if (start == end) return;
    i ← Partition (A, start, end);
    QuickSort (A, start, i-1);
    QuickSort (A, i+1, end);
}
```

Partition:



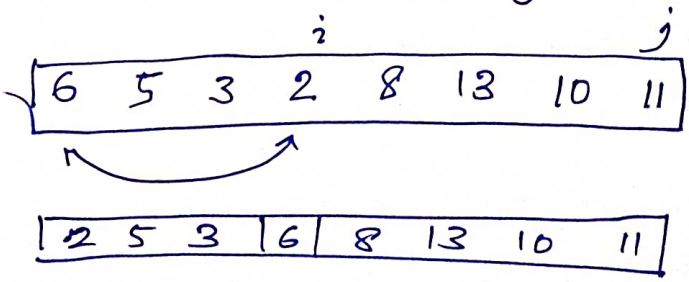
i : Boundary upto pivot

j : elements processed.

move j forward → until $A[j] \leq x$.

i ← i+1

swap A[i] and A[j]



Partition takes $O(n)$.

Quick Sort :—

$$T(n) = O(n) + T(i-1) + T(n-i)$$

Best Case

Partition in middle

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

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

Worst Case

$i = 1$

$$T(n) = O(n) + T(n-1)$$

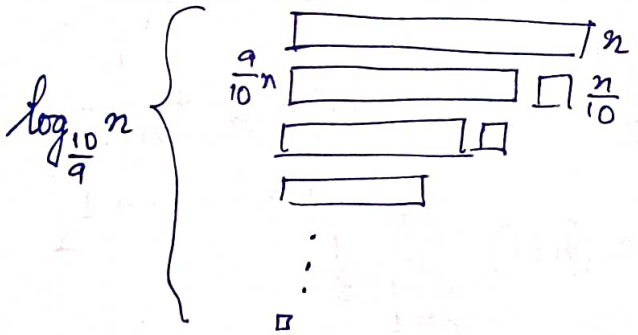
~~$O(n)$~~

$$T(n) = O(n^2)$$

Average Case.

For uneven partition—

$$T(n) = O(n) + T\left(\frac{9}{10}n\right) + T\left(\frac{n}{10}\right) \Rightarrow O(n \log n)$$



$$T(n) = O(n) + T(i-1) + T(n-i)$$

$$= cn + T(i-1) + T(n-i)$$

$1 \le i \le n$ with equal probability.

$$T(n) = cn + \frac{1}{n} \sum_{i=1}^{n} [T(n-i) + T(i-1)]$$

$$= cn + \frac{2}{n} \sum_{i=1}^{n} [T(0) + T(1) + \cdots + T(n-1)]$$

$$T(n-1) = c(n-1) + \frac{2}{n-1}\left[T(1) + \cdots + T(n-2)\right]$$

$$\Rightarrow nT(n) - (n-1)T(n-1) = cn^2 - c(n-1)^2 + 2T(n-1)$$

$$\Rightarrow nT(n) - (n+1)T(n-1) = c(2n-1)$$

$$\Rightarrow \frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2c(2n-1)}{(n+1)n} \approx \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} - \frac{T(n-2)}{n-1} \doteq \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} - \frac{T(n-3)}{n-1} = \frac{2c}{n}$$

$\leq c' \log n$

$$\Rightarrow \frac{T(n)}{(n+1)} - \frac{T(1)}{2} = 2c\left[\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + 1\right]$$

$$\Rightarrow T(n) = 2c(n+1)\sum_{i=1}^{n+1}\frac{1}{i}$$

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

Problem: Polynomial Multiplication

Suppose we have two polynomials of degree $n-1$

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1 x + a_0$$

$$B(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \dots + b_1 x + b_0$$

Let $C(x) = A(x) \cdot B(x)$

$$C(x) = C_{2n-2}x^{2n-2} + C_{2n-3}x^{2n-3} + \dots + C_1 x + C_0.$$

$$c_i = \sum_{\substack{0 \le j,k \le n-1 \\ j+k=i}} a_j b_k = \sum_{j=0}^{n-1} a_j b_{n-j} \longrightarrow O(n)$$

$\forall\, c_i,\ O(n) \Rightarrow$ Total time complexity $O(n^2)$

"Believe in Divide and Conquer"

$$A(x) = a_{n-1}x^{n-1} + \dots + a_1 x + a_0$$

$$B(x) = b_{n-1}x^{n-1} + \dots + b_1 x + b_0$$

Divide:

$t = \lceil \frac{n}{2} \rceil$

$$A(x) = \overbrace{a_{n-1}x^{n-1} + \cdots + a_t x^t}^{x^t \cdot A_{hi}(x)} + \overbrace{a_{t-1}x^{t-1} + \cdots + a_1 x + a_0}^{A_{lo}(x)}$$

$$B(x) = \underbrace{b_{n-1}x^{n-1} + \cdots + b_t x^t + b_{t-1}x^{t-1} + \cdots + b_1 x + b_0}_{B_{lo}(x)}$$

$$\Rightarrow A(x) = x^t A_{hi}(x) + A_{lo}(x)$$

$$B(x) = x^t B_{hi}(x) + B_{lo}(x)$$

$$A(x)B(x) = x^{2t}(A_{hi} B_{hi}) + x^t(A_{hi} B_{lo} + A_{lo} B_{hi}) + A_{lo} B_{lo}$$

where A_{hi}, B_{hi}, A_{lo}, B_{lo} ~~part is~~ are polynomials of degree $\frac{n}{2}$.

Conquer: Recursively do this

Combine: Add the subproblems.

$$T(n) = O(1) + 4T\left(\frac{n}{2}\right) + O(n)$$

Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a \geq 1 \quad b > 1$

$$f(n) = O\left(n^{(\log_b a - \varepsilon)}\right) \text{ for some } \varepsilon > 0$$

Then $T(n) = O(n^{\log_b a})$

"Divided but could not be conquered"

$$(A_{hi} + A_{lo})(B_{hi} + B_{lo}) - A_{hi}B_{hi} - A_{lo}B_{lo} = A_{hi}B_{lo} + A_{lo}B_{hi}$$

∴ We can just compute these multiplication.

$$O\left(n^{\log_2 3}\right) = O(n^{1.58})$$

This is known as "Karatsuba's Polynomial Multiplication"

1. Large Number multiplication:

n digit number × n digit number

2. Matrix Multiplication:

$$A_{n}\begin{bmatrix} n \\ \cdot \end{bmatrix} \times B_{n}\begin{bmatrix} n \\ \cdot \end{bmatrix}$$

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

a, b, c, d, e, f, g, h are $n \times n$ matrices.

$$C = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

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

$$\Downarrow$$

$$O(n^{\log_2^8}) = O(n^3)$$

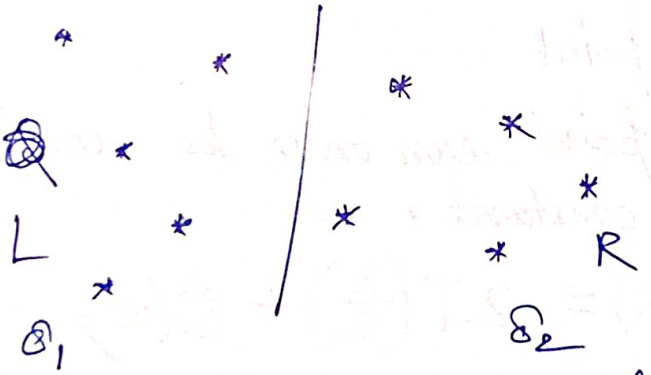
"Strassen used 7 multiplication"

Problem (Geometry):—

Given n points in a 2D plane, find the pair that is closest among all.

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Naive Method → Check for every pair. $\binom{n}{2}$ pairs
For each find distance.



L, R have $\frac{n}{2}$ points each.

Divide: Divide into two equal parts (each containg $\frac{n}{2}$ points

Conquer: Recursively divide.

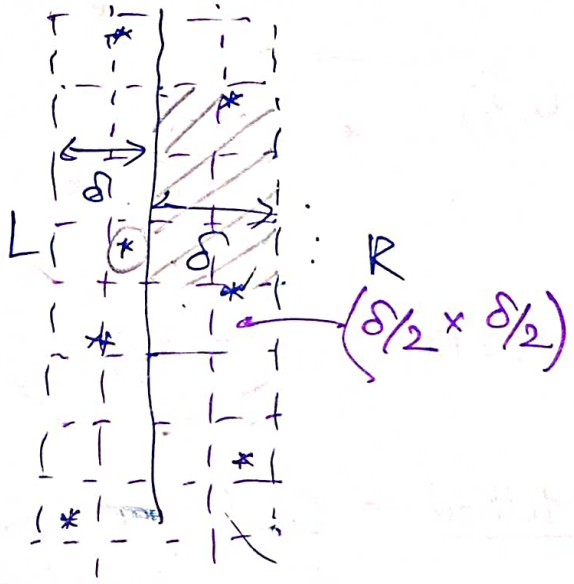
Combine: $\min(\delta_1, \delta_2)$ would not work

~~so, we divide it into~~

We gain nothing by dividing it in 2δ strip

Hint: Try using y-coordinate

P_y': Sorted in y coordinate. in the strip



* Each box can have at most one point

Each point can only be compared with constant.

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

* Try without sorting again & again

GRAPH THEORY

• Cities (nodes/vertices) and Interconnections (edges) among the cities.

Graph Theory : History :-

Euler and his Seven briges of Königsberg. He wanted to find an 'Eulerian Tour', a path which passed through each bridge exactly once.

GRAPH :-

A Graph $G = \langle V, E \rangle$ is a set of 'points" or nodes and links between them. Links are also termed as edges/arcs.

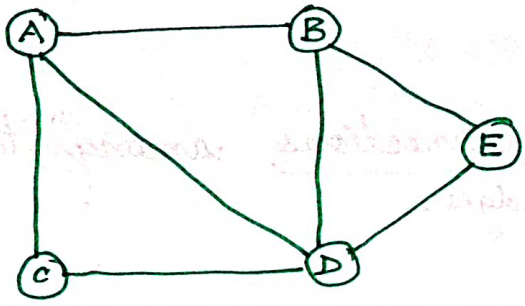
$V \longleftarrow$ Set of vertices

$E \longleftarrow$ Set of edges.

$$E \subseteq V \times V$$

E is a set of order pair of vertices.

"It is always better to represent graph pictorially"



$E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$

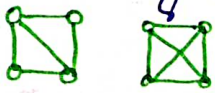
Undirected edge:

These type of edges are bi-directional in nature. i.e. $(A, B) = (B, A)$

Directed edge:

Edges that are unidirectional, in this case $(A, B) \neq (B, A)$

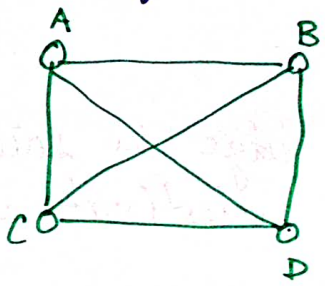
Undirected graph: A graph having only undirected edges.



Directed graph: A graph having only directed edges.

Complete Graph :-

A complete graph is where every node is connected to every other node by an edge.



An undirected complete graph of n nodes has $\binom{n}{2}$ edges.

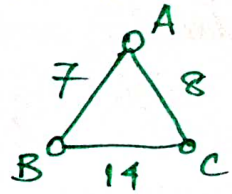
Regular Graph :-

A regular graph is where all nodes have the same number of edges going out of them. If that number is k, we call it a "k-regular graph"

A complete graph is an ~~k~~ $(n-1)$-regular graph.

Weighted Graph :-

A graph is said to be weighted if a non-negative value is assigned to every edge of the graph.



→ Typically weights represent some sort of cost.

Outgoing edge:

A directed edge is said to be outgoing from its origin vertex

Incoming edge:

A directed edge is said to be incoming on its destination vertex

Degree:

The number of edges connected to a vertex is called its degree.

A regular graph might also be defined as a graph in which all nodes have the same degree.

The minimalist construction of a graph

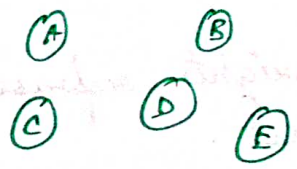
$G = \langle V, E \rangle$, $V \neq \phi$, E can be null

Minimalist construction :—

1.

A single, lonely, isolated, sad vertex. degree = 0

2. Generalized version :—



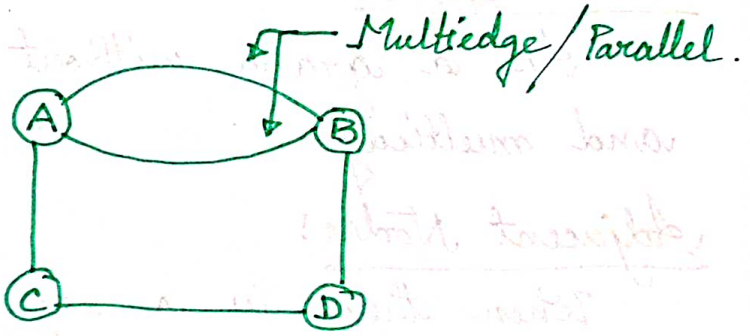
Multiple & lonely, isolated sad vertices with degree = 0

Now, degree of each node is equal to 0, then we call the graph a "Degenerate Graph"

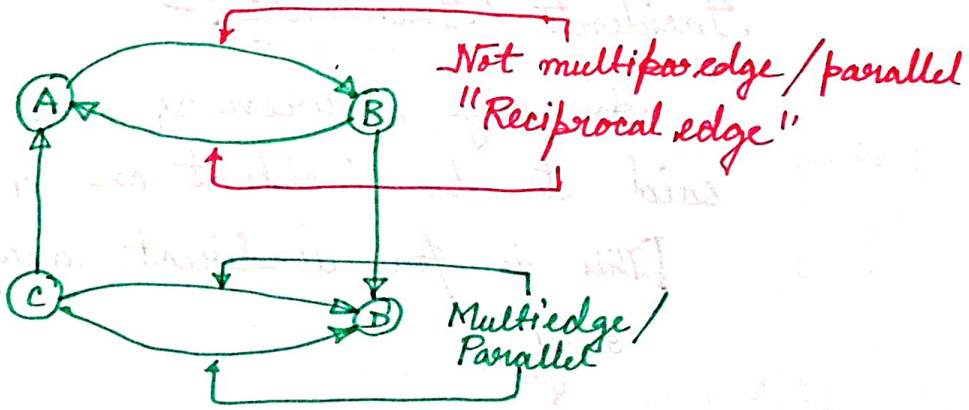
Indegree: Number of Incoming edges
Outdegree: Number of Outgoing edges.

Parallel / Multiedge:

* Two undirected edges having the exact same end vertices.



* Two directed edges having the exact same origin and destination vertices.



Self-Loop:-

An edge that starts and ends on the same vertex, or when its endpoints coincide.

(A, A)

(A, A)

Simple Graph

An a graph without self-loops and multiedges.

Adjacent Nodes:

When there is a edge between two nodes, they are called adjacent nodes.

Incident / Incidence:-

An edge between v_1 and v_2 is said to be incident on v_1 and v_2.

[This is for undirect graphs only]

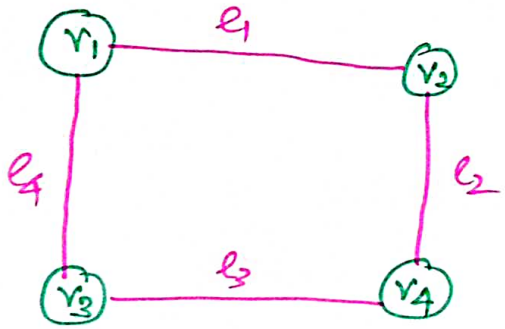
Walk :—

A walk is a finite alternating sequence of vertices and edges always beginning and ending in some vertex.

Closed Walk :—

A walk beginning and ending at the same vertex is called a "closed walk".

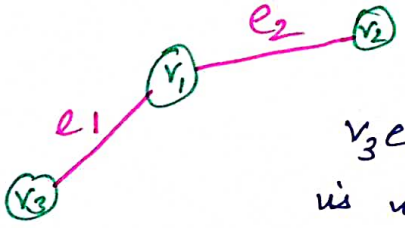
Else it is called an "open walk"



v_1 e_1 v_2 e_2 v_4 e_3 v_3 e_4 v_1 is a closed walk

Path :—

~~A walk~~ An open walk where no vertices are repeated.



$v_3 e_1 v_1 e_2 v_2$ is a path.

Non-repetition of vertices
$\Rightarrow$ non-repetition of edges.

Circuit:

A closed walk, where no edges are repeated. [Does not mean vertices cannot be repeated]

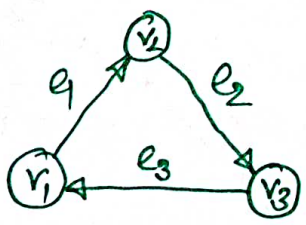
Cycle:

A path that has the same start and end vertices. (A closed path).

A graph having a cycle is called a "cyclic" graph.

A graph that has no cycles are called "acyclic" graph.

A directed graph with no cycle is known as a "directed acyclic graph" (DAG). [Very important construction].



Walk	• Vertices may repeat • Edges may repeat • Open / Close
Circuit	• Vertices may repeat • Edges ~~may~~ cannot repeat • ~~Open~~ Closed.
Path	• Vertices cannot repeat • Edges ~~may~~ cannot repeat • Open
Cycle	• Vertices ~~may~~ cannot repeat (except start and end) • Edges cannot repeat • Closed.

Sub graph:

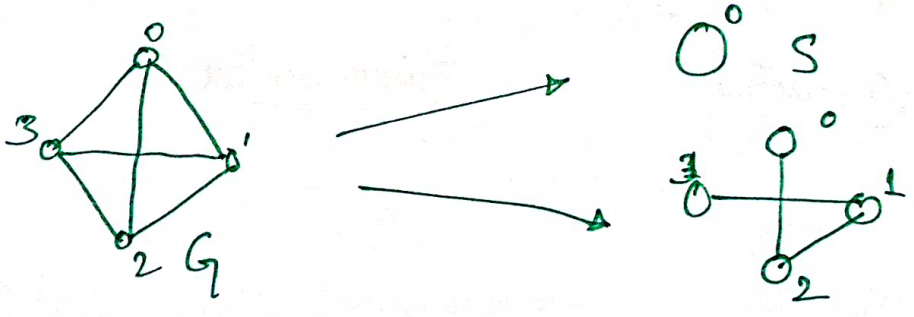
S is a subgraph of G, then

$S = \langle V', E' \rangle$ of $G = \langle V, E \rangle$

$V' \subseteq V$.

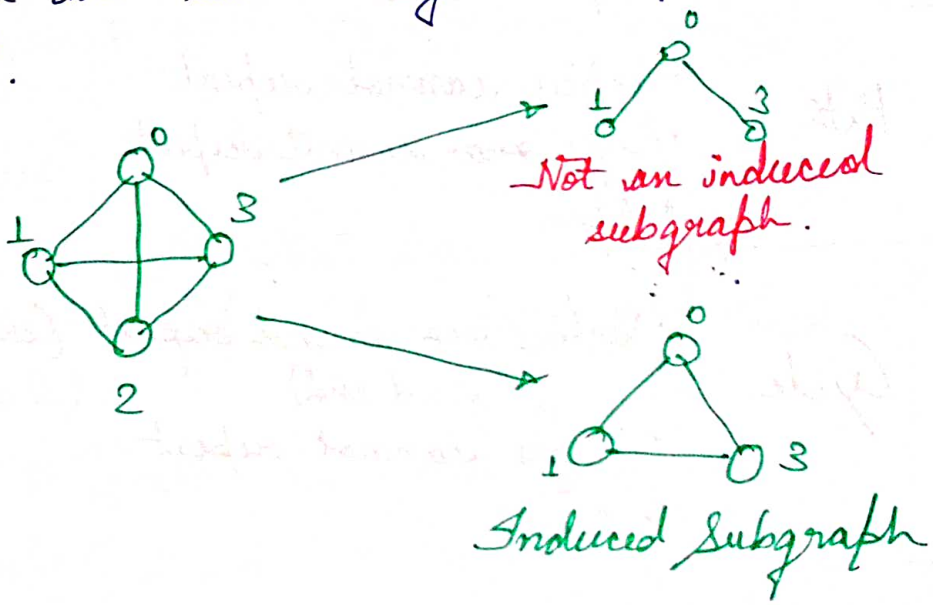
$E' \subseteq E$.

All the edges in S are in G and each edge in S has the same end point as G



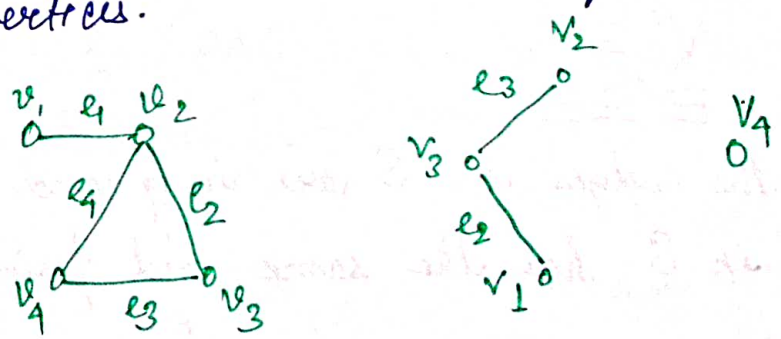
Induced Subgraph

If two vertices (u, v) are present in S and there is an edge between u and v in G, then there must be the same edge also present in S.



Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices.



Connected

Disconnected

Computer Representation of Graphs

1. Adjacency Matrix :-

- A matrix representation of the graph.

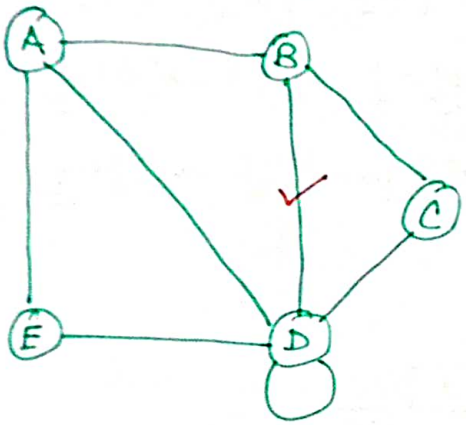
Let $G = \langle V, E \rangle$, $|V| = n$, $n \geq 1$.

G is represented as a 2D array A

$A[i, j] = 1$ iff $(v_i, v_j) \in E$

$= 0$ otherwise

[Must be symmetric]



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

For a directed graph, direction is maintained.

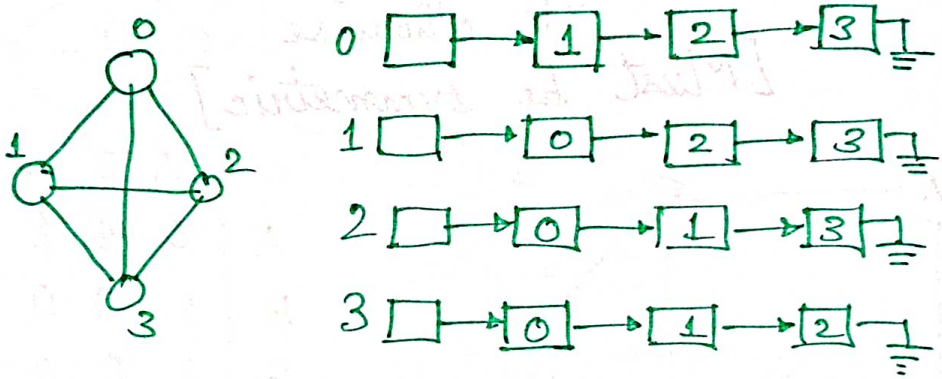
$$A[i, j] = \begin{cases} 1 & \text{iff } \langle v_i, v_j \rangle \in E \\ 0 & \text{otherwise.} \end{cases}$$

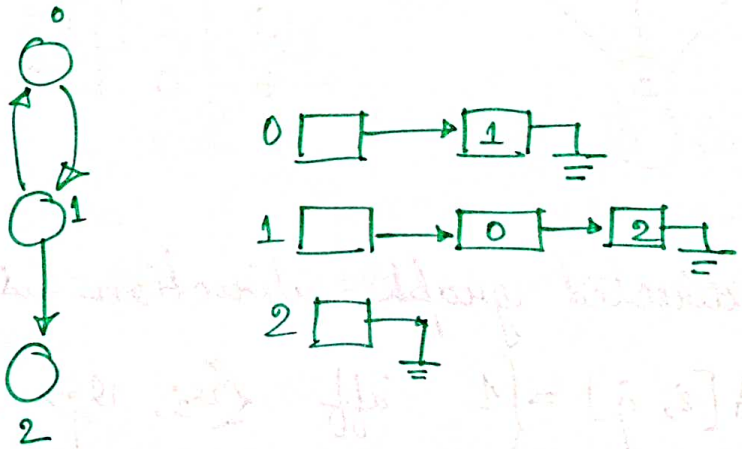
[May not be symmetric]

2. Adjacency List:

In this representation, every vertex of the graph contains a list of its adjacent vertices.

If there are n nodes, there will be n lists. In this set of lists, the list i represents nodes adjacent to i.



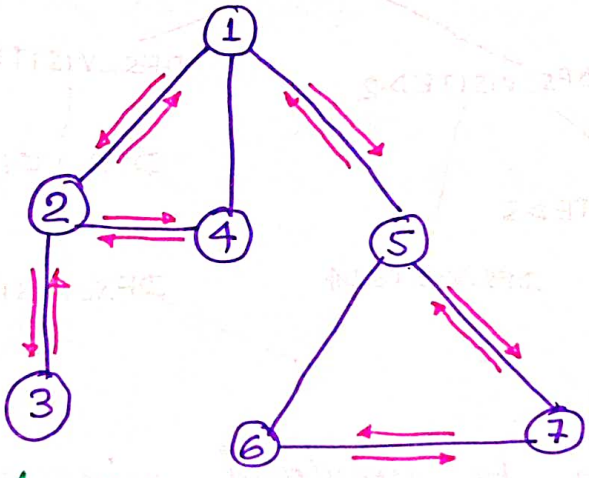


Advantage of Adjacency List over matrix: Sparse graph (less no. of edges)

However for a dense graph, both adjacency matrix and adjacency list provide similar benifits.

Traversal on graphs:—

- Depth first search (DFS)
  Depth by depth
- Breadth first search (BFS)
  Breadth by breadth.



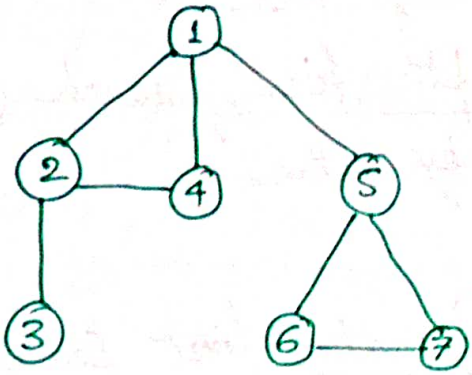
One possible DFS traversal is shown

Cost of traversal $O(n+m)$; $m = |E|$, $n = |V|$

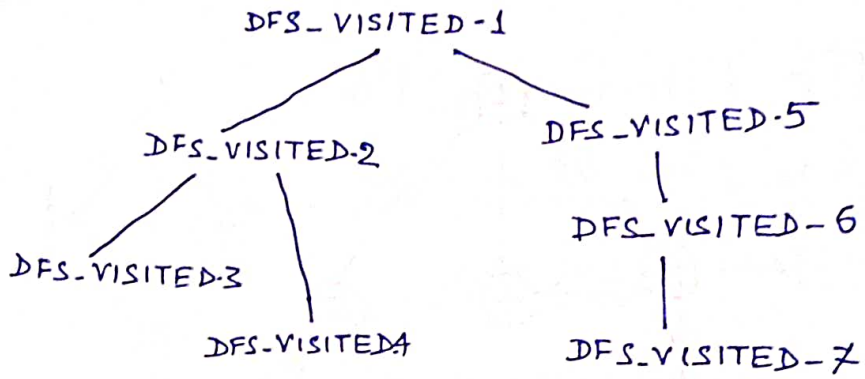
Algorithm :—

```
DFSvisit(u) {
    u.visit = True;
    for every outgoing edge (u,v), do
        if v.visited = False then
            DFS visit(v)
    end
end
}
DFS {
    for [~~vertex u~~ every vertex u).
        if (u.visited = False)
            DFSvisit(u);
}
```

Simulating stack using recursion.



One possible visit sequence:



DFS can be simulated using a stack. (or a recursion to implement stack).

DFS Tree:

While executing DFS, if DFS visit(u) calls DFS visit(v) directly, then we connect (u, v) with a "tree edge" and name v a child of u and u as the parent of v.

DFS → Two important orders →

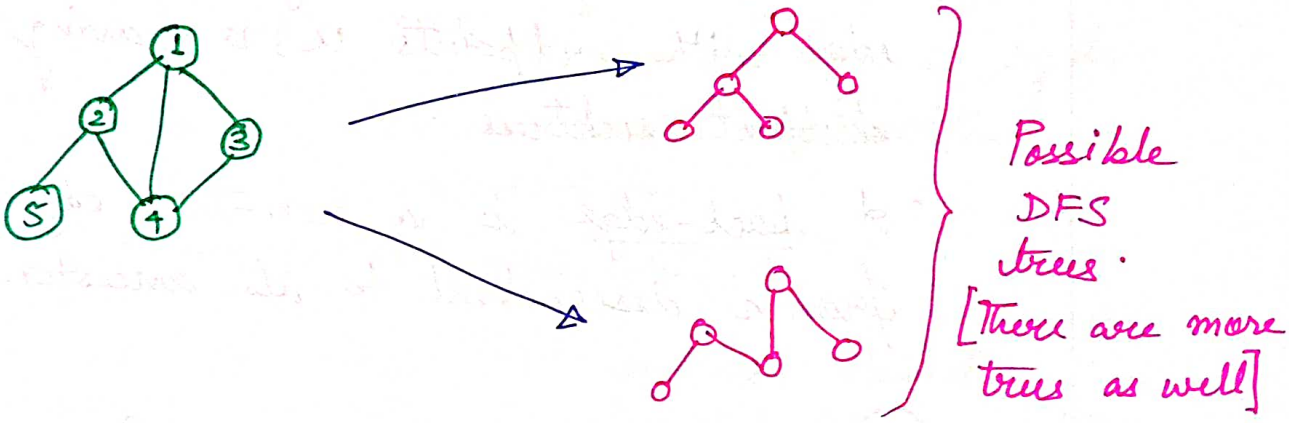
When a node is pushed → pre-order of DFS

When a node is popped → post-order of DFS.

Pre order: 1 2 3 4 5 6 7

Post order: 3 4 2 7 6 5 1.

#DFS tree is not unique.



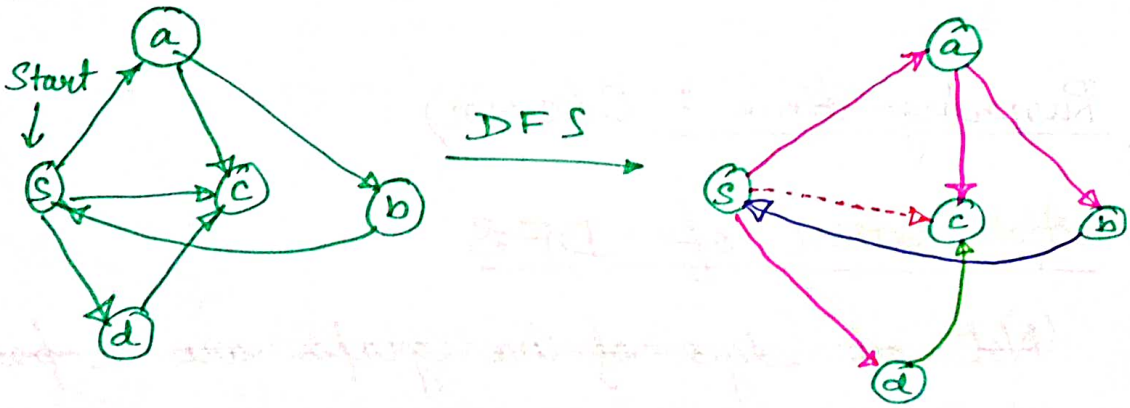
Running time : $O(n+m)$

~~Application of DFS~~

~~Not all edges of the graph are a part of DFS~~

Edge Classification:-

- Not all edges are a part of the DFS tree.
  - A tree edge is an edge present in the DFS tree.
  - A forward edge is a non-tree edge from an ancestor to a vertex ~~which is non~~
    - A cross-edge is ~~an edge~~ a non-tree edge with endpoints u, v having distinct subtrees.
      - A back-edge is a non-tree edge from a descendant to its ancestor.



→ Tree edges.
→ Back edges.
⇢ Forward edge
→ Cross edge

Algorithm to classify edges post DFS

Edge type	Pre / Post - order
Tree / forward	$pre(u) < pre(v)$; $post(v) < post(u)$
Back edge	$pre(v) < pre(u)$, $post(u) < post(v)$
Cross edge	$pre(v) < post(v) < pre(u) < post(u)$

Cycle Finding :-

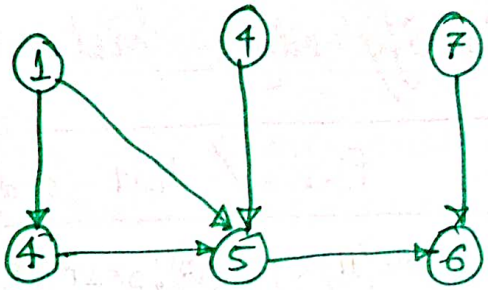
H.W.: Prove that a graph has a cycle iff it has a back edge with respect to a DFS Tree.

[Algorithm for cycle detection]

white ← nodes that are not visited.
gray ← nodes "being" visited. We are still exploring its descendents.
black ← nodes already visited.

```
colour [v ∈ V] ← white
for all nodes v of the graph, do
    if colour [v] = white do
        dfs (v)

dfs (node v)
    colour [v] ← gray
    for all nodes w adjacent to v, do
        if colour [w] = "gray"
            spit out "Cycle found"
        else if colour [w] = "white", then
            dfs (w)
    colour [v] ← black.
```

Work for a directed graph.



DAG

"Any DAG can be topologically sorted"

"Topological sort" is an ordering of the vertices where the edges in the DAG go from predecessor to successor

Algorithm: Sort according to ~~top~~ post order and reverse it.

Greedy algorithm: Select vertex with indegree 0 ~~and delete~~ and delete it.

Breadth First Search :-

$G = \langle V, E \rangle$

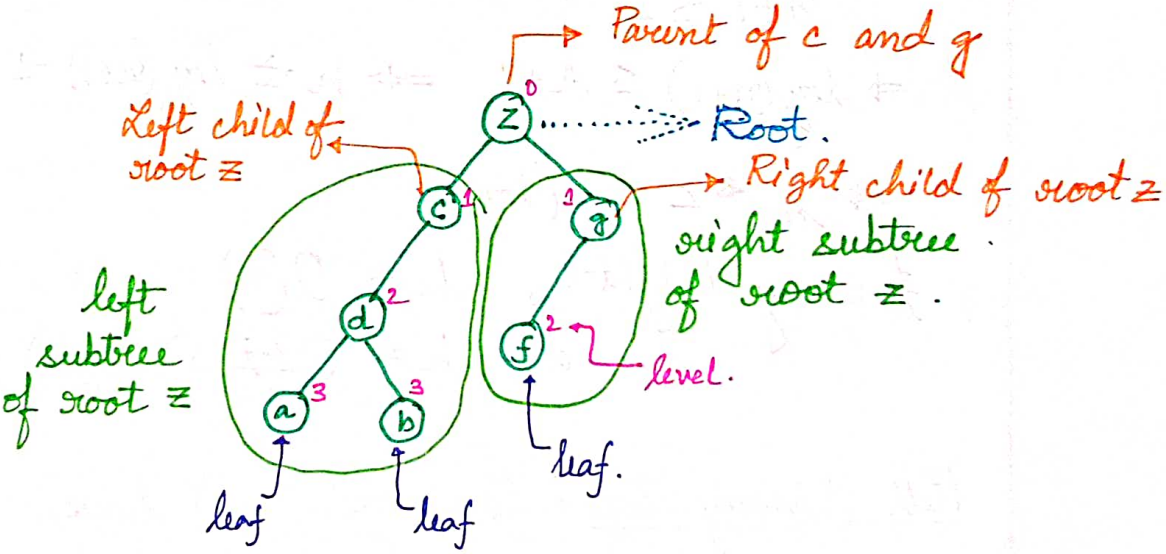
```
BFS(G){
    for u ∈ V do
        BFSvisit(u, G)
    end for.
}

void BFSvisit(u, G) {
        Mark u visited.
        Add u to queue Q.
        While Q is not empty
            v ← head of Q
            for w | (v, w) ∈ E do.
                if w is not visited then
                    mark w as visited.
                    add w to Q
                end if
            end for
            remove v from Q.
        end while.
}
```

BINARY TREES

We capture some sort of hierarchy within a set.

Definition: Binary Tree is a set of elements.

- with one special element called the "root"
- all other elements can be partitioned into two (possibly empty) sets, left and right, each of which is also a binary tree.



* **Leaf node:** A node with no children

* **Level:**

  Level of root = 0
  Level of any node = distance of node from ~~any base~~ root.
  = no. of edges in the path from root to node

* **Height of the tree:** maximum level of any node.

Maximum number of nodes at level h is 2^h.

Maximum number of nodes ~~at a level of height~~ in a tree of height h, is $2^{h+1} - 1$

Minimum height of a binary tree with n nodes —

$$n \leq 2^{h+1} - 1$$

$$n+1 \leq 2^{h+1}$$

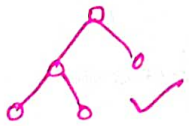
$$\Rightarrow \log_2(n+1) \leq h+1 \quad \Rightarrow h \geq \log_2(n+1) - 1$$

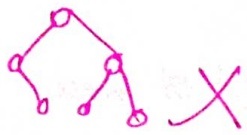
$$\Rightarrow h \geq \Omega(\log n)$$

for any height h, $h = O(n)$

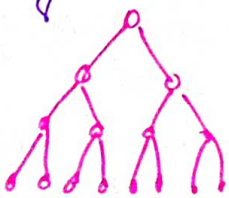
and $h = \Omega(\log n)$

Full binary tree: Each node will have either 0 or 2 children.





Complete binary tree: tree with $2^{h+1} - 1$ nodes



Almost Complete binary tree :- Nodes are filled from top to bottom & from left to right with no gaps.

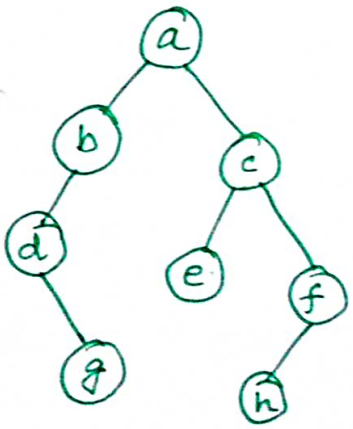
Each ~~tree~~ level except the last one must have 2^h nodes. The last level must be filled left to right.

OPERATIONS ON A BINARY TREE :

Main application of bi-tree comes with special type of bi-tree.

* Traversal
  - "visit" each node once.
  - order of visiting?

  ① Preorder : root, left, right.
  ② Inorder : left, root, right.
  ③ Postorder : left, right, root.



Preorder: a, b, d, g, c, e, f, h
Inorder: d, g, b, a, e, c, h, f
Postorder: g, d, b, e, h, f, c, a

For a node (x):

= NIL if doesn't exist	key[x] : data stored left[x] : left child right[x] : right child parent[x] : parent. (optional.)

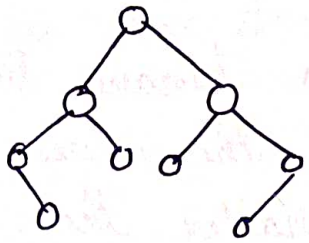
```
typedef struct node {
        int key;
        struct node *left, *right, *parant;   (optional)
} TNODE;
```

```
Preorder (x)
        if x = NIL return
        visit(x)        // generate template
        Preorder (left[x])
        Preorder (right[x])
```

```
Inorder (x)
        if x = NIL return
        ~~visit(x)~~ Inorder (left[x])
        Visit(x)
        Inorder (right[x])
```

```
Postorder (x)
        if x = NIL return
        Postorder (left[x])
        Postorder (right[x])
        visit(x)
```

P1) Count the number of nodes in a binary tree



```
findsize(x)
    if x = NIL return 0
    c1 = findsize(left[x])
    c2 = findsize(right[x])
    return c1 + c2 + 1.
```

} Either traversal would work.

P2) Count the number of leaf nodes in a tree.

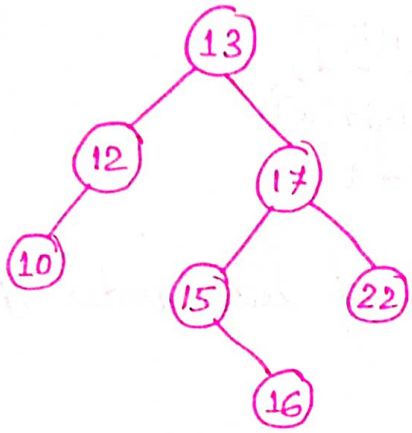
```
numleaves(x)
    if x = NIL return 0
    c1 = numleaves(x)
    c2 = numleaves(x)
    if (c1 + c2 == 0) return c1 + c2 + 1
    return c1 + c2.
```

} Similar to post-order

Many applications come with a special Binary Trees.

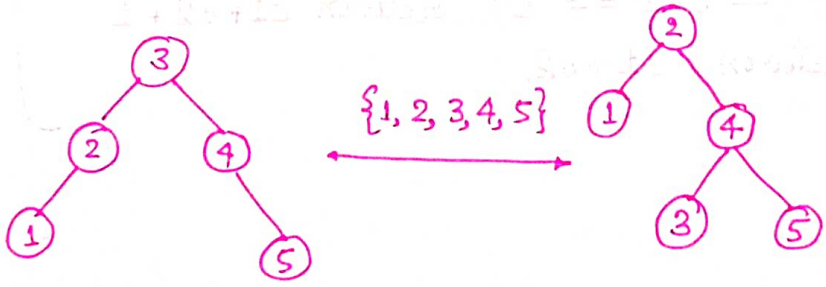
BINARY SEARCH TREE (BST)

BST is a binary tree such that for any node, the values in its left subtree are smaller than the node and right have a smaller value than the node.



l-subtree
all values are smaller

r-subtree
all values are larger.



"Search resembles binary search"

```
search (x, k)
    if (x = NIL) return not found
    if (key[x] = k) return found.
    if (key[x] > k) return search(left[x], k)
    return search(right[x], k)
```

Complexity (Worst) : $O(h) = O(n)$

* But we can use some tricks to make it $O(\log n)$

Main Operation:

- Search
- Insert
- Delete

} $O(h)$, which can be made $O(\log n)$

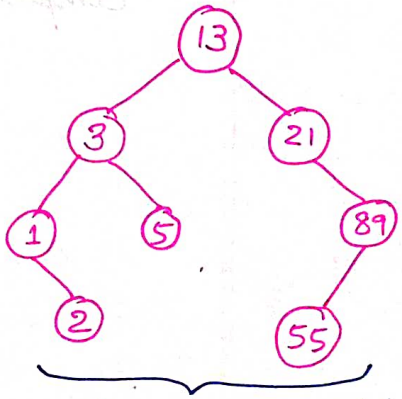
Other operation:

- Find Min
- Find Max
- Find Successor (Find the element next to x)
- Find Predecessor

```
FindMin(x)
    while (left[x] != NIL)
        x = left[x]
    return key[x]
```

```
FindMax(x)
    while (right[x] != NIL)
        x = right[x]
    return key[x]
```

```
FindSuccessor(x)
    if (right[x] != NIL)
        return FindMin(x)
    y = parent[x]
    while((y != NIL) and (x = right[y]))
        x = y
        y = parent[y]
    return y
```



If x has a right subtree, its min of right subtree, else its just the node p whose max is 2 in left subtree

We can write FindPredecessor in a similar fashion.

For insert,

Insert (x, k).

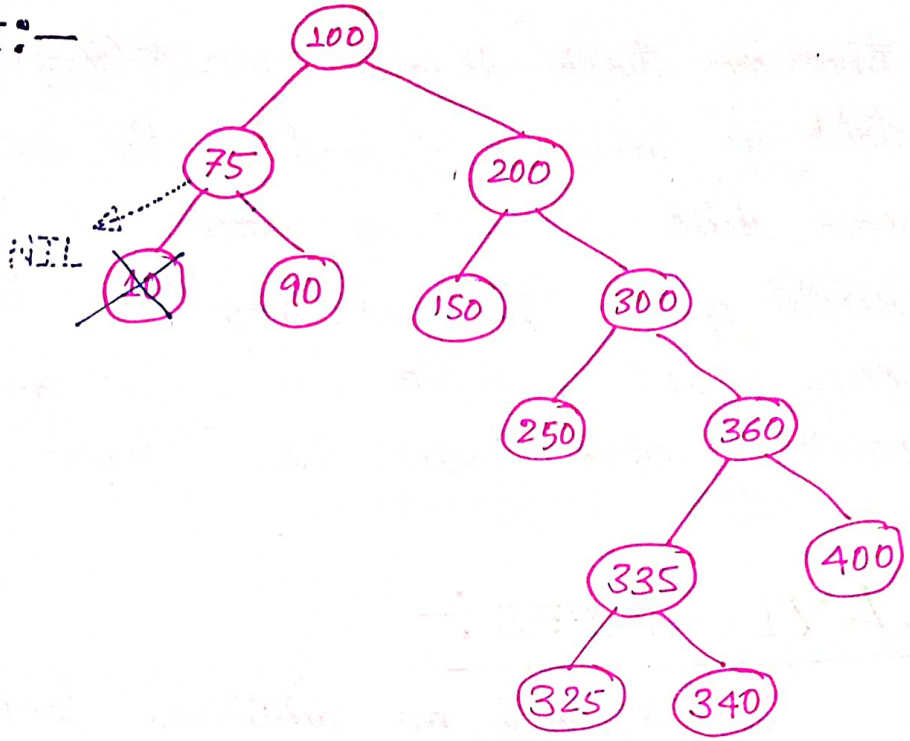
search for (x, k)

at any point we encounter NIL, we insert a new node with key k.

```
Insert (x, k)
    if key [x] > k
        if left [x] = NIL
            add k as a left child of x
            (create a new node y)
        else
            Insert (left [x], k)
    else
        if right [x] = NIL
            add k as a right child of x
        else
            Insert (right [x], k)
```

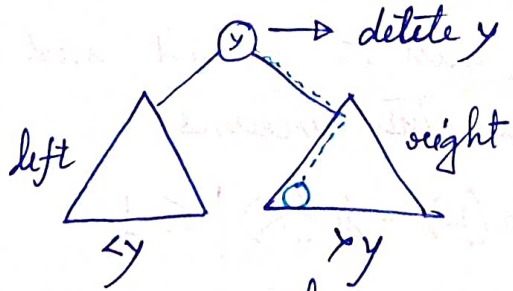
Complexity $O(h)$

Delete :—



for delete (y) —

* If y is a leaf node, simply remove it.
* If y had a single child, elevate the single child. (Modify p parent[y] and modify parent of deleted node).
* If y had two children



Find right successor of y = z.

~~delete z,~~

change key[y] = key[z]
and then delete z.

z cannot have more than ~~two~~ one child.

search, insert, delete in O(n)

"There are tricks to make $h = O(\log n)$. The trick is simple; we modify the insert and delete in such a way that the height of the tree at any time, by doing something extra. But "doing something extra" must be $O(\log n)$"

AVL TREES :-

BST's with an additional property, for any node, the heights of the left and right subtree differs by at most 1.

"This implies $h = O(\log n)$!! Yay !!"

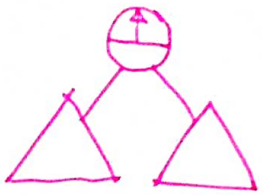
Formally, let x be a node in AVL Tree. Let L and R be left and right subtrees of x. This means

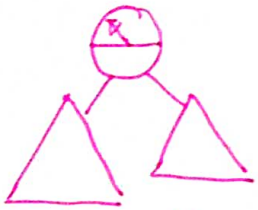
$$|h(L) - h(R)| \leq 1$$

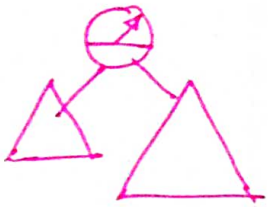
* In addition to left, right, parent and key, we maintain a balance factor,

$$b = h(R) - h(L)$$

b should have a value of -1, 0 or 1.





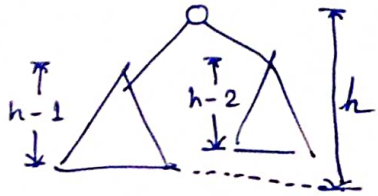


For a given AVL tree of ~~tsgf~~ height h,
Let minimum no. of nodes be M_h.

L	$h-1$	$h-2$	$h-1$
R	$h-1$	$h-1$	$h-2$
	Case 1	Case 2	Case 3.

for h to be minimum, L and R must also have minimum number of nodes.

However case 1 can be ignored.



$$\therefore M_h = M_{h-1} + M_{h-2} + 1$$

$$\therefore (M_h + 1) = (M_{h-1} + 1) + (M_{h-2} + 1)$$

$$\begin{cases} M_0 = 1 \\ M_1 = 2 \end{cases}$$

Let $N_h = M_h + 1$

$$\Rightarrow N_h = N_{h-1} + N_{h-2}$$

$$\begin{cases} N_0 = 2 \\ N_1 = 3 \end{cases}$$

$$\therefore N_h = F_{h+3}$$

$$\therefore N_h = F_{h+3} \approx \frac{1}{\sqrt{5}} \phi^{h+3} \Rightarrow M_h \approx \frac{1}{\sqrt{5}} \phi^{h+3} - 1$$

$$\Rightarrow n \geq \frac{1}{\sqrt{5}} \phi^{h+3} - 1 \quad , \phi = \frac{\sqrt{5}+1}{2}.$$

$$\Rightarrow h = O(\log n)$$

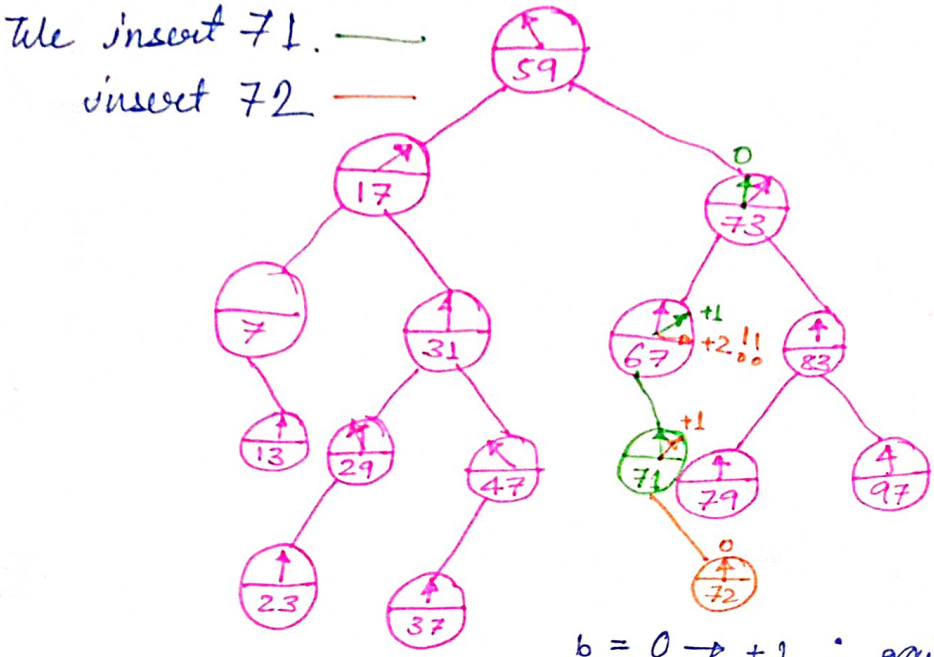
BST Deletion :

```
BST delete (BST T, int key){
    BST p, u, v;
    u = T;  p = NULL;
    while (u != NULL){
        if (key == u -> val){
            if (u -> left == NULL || u -> right == NULL){
                v = (u -> left == NULL)? u -> right : u -> left;
                if (p == NULL)  T = v;
                else if (key < p -> val)  p -> left = v;
                else  p -> right = v;
            } else {
                p = u; v = u -> left
                while (v -> right != NULL){
                    p = v; v = v -> right;
                } u -> val = v -> val;
                if (p == u)  p -> left = v -> left
                else  p -> right = v -> left
            } return T;
        } p = u;
        if (key < u -> val)  u = u -> left;
        else  u = u -> right;
    } return T;
}
```

Insertion in AVL Trees :-

→ Proceed like a normal BST

→ It may throw the tree out of balance. some additional steps are required to maintain the AVL property.



b = 0 → +1 : gained height.

1. While going up, we encounter a balance factor of -2

* (-1 to -2)

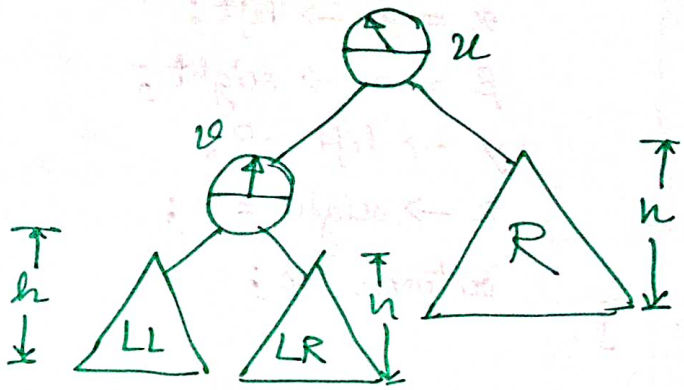
⇒ Its left subtree must have gained height —— balance factor is +1 or -1.

-2 , +1 → opposite sign } Done here, No need to go up.

-2, -1 → same sign. }

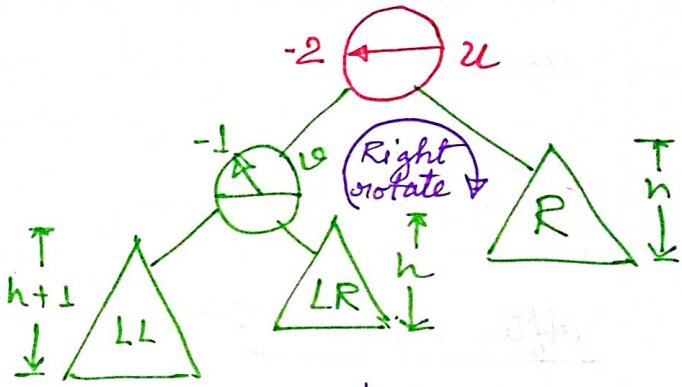
Same sign :—

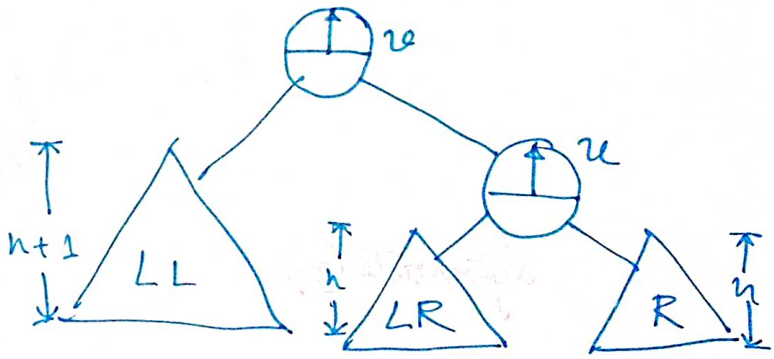
Before insertion :

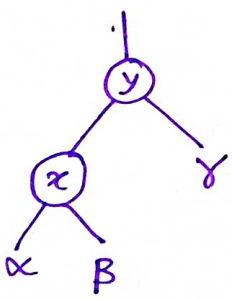


After insertion :

Right Rotation



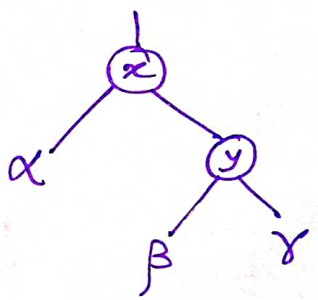




$\alpha < x < \beta < y < \gamma$

Right-rotate(y)

Binary search tree property ~~doesn't~~ doesn't change.



$\alpha < x < \beta < y < \gamma$

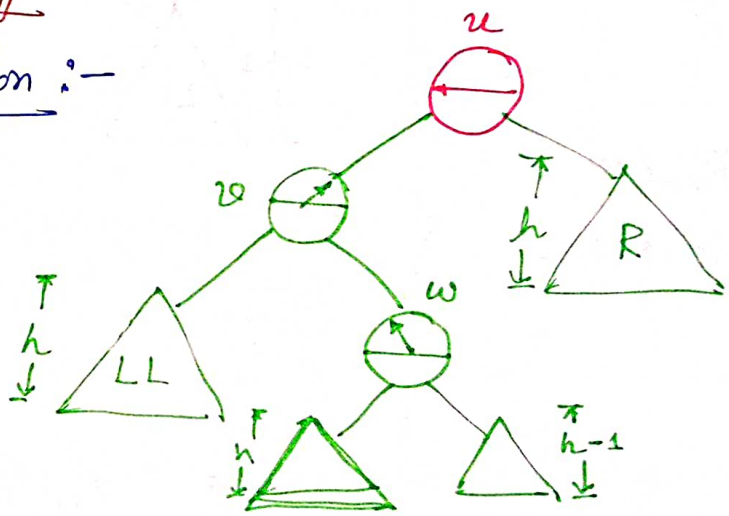
```
Rightrotate(y) {
    x = y -> left;
    β = x -> right;
    y -> left = β;
    x -> right = y;
    return x;
}
```

O(1)

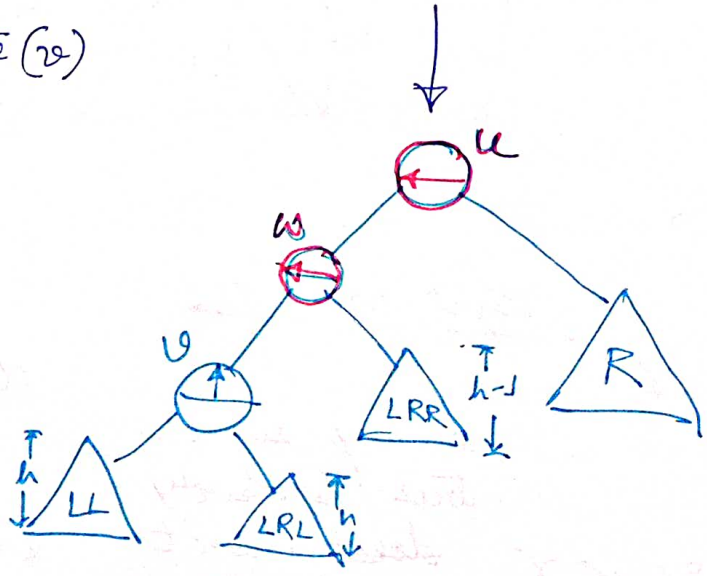
"We can have a similar situation of left rotation"

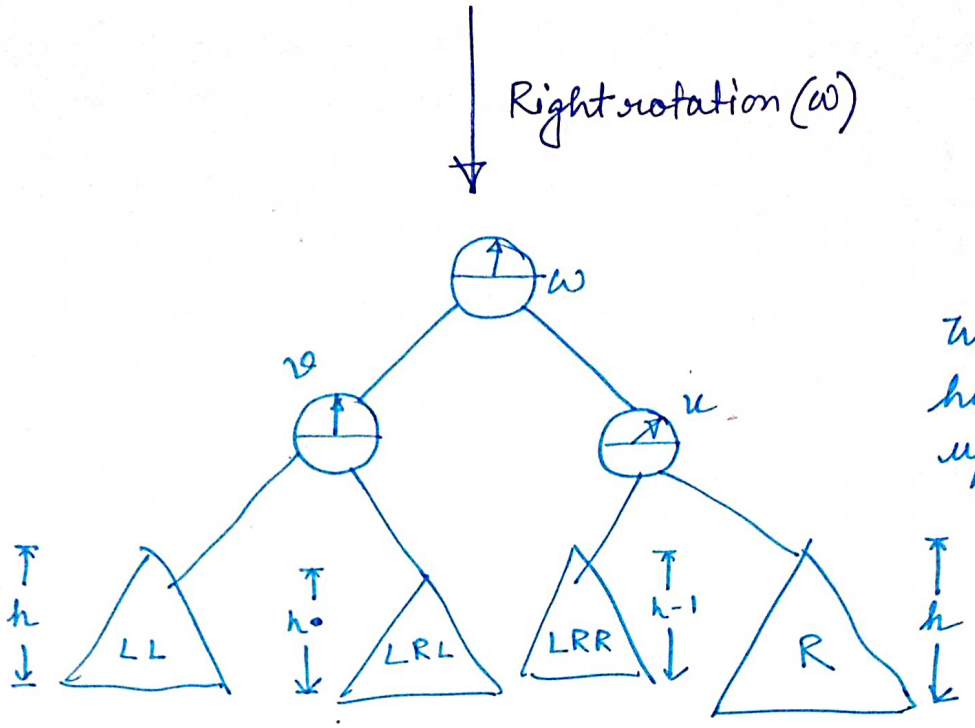
Opposite sign:-

After insertion :-



Liftrotate(v)





We do not have to go up.

$\therefore$ **Complexity of Insert = $O(\log n)$**

"Deletion may take $O(\log n)$ rotations, but that too leads complexity of $O(\log n)$"

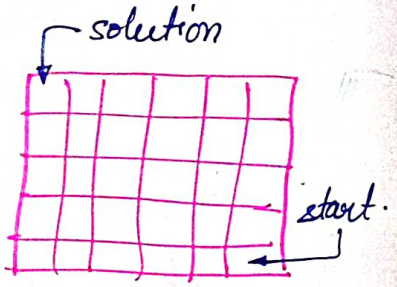
DYNAMIC PROGRAMMING

Divided the problem into subparts

↳ Smaller instances are related.
(DNQ had smaller instances unrelated)
(optimal substructure property)

"Dynamic Programming" comes from the Control Engineering (EE) subjects. "Programming" means maintaining a table (to store the solution to the subproblems.

D&C ⟶ top-down approach
DP ⟶ bottom-up approach.



Steps of DP:-

- Establish a recursive property that gives the solution to an instance of the problem, in terms of optimal substructure
- Solve an instance of the problem in bottom-up approach fashion, by solving smaller & instances first.

$f(n) = f(n-1) + f(n-2)$

Hunt for optimal substructure.

How does the smaller instances combine to give the solution to the larger instance

SHORTEST PATH FINDING

* Path — open path with no vertices repeated.
* weighted graph path length — sum of edge wt's in the path.
* Shortest path length — path between two nodes with least path length (not unique)

Floyd's Algorithm :-

— Allows us to compute the length of the shortest path between any two vertices.
— One of the shortest paths

Optimization Problem :—

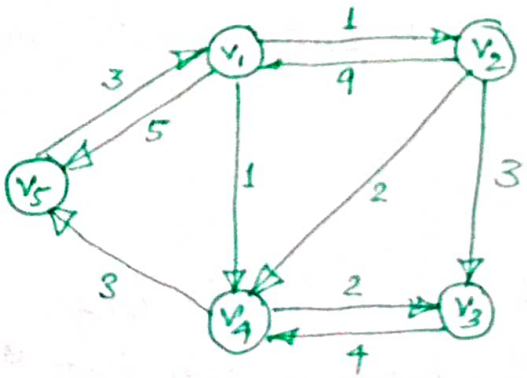
- Will have many candidate solution
- Each solution of this instance having some sort of value associated with it.
- The final solution to the instance is the optimal value.

Finding Shortest Path :—

Candidate solutions : all paths between a pair of vertices

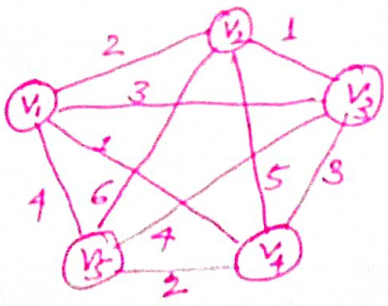
value : path length

optimal value : path with minimum length.



Brute force : Find all paths present between any pair of vertices and from them choose the shortest one

A complete graph (with weights on edges)



Paths of length $n = (n-2)!$ between v_1 & v_2

$\therefore O((n-2)!)$ ← Worse than exponential!!

FLOYD'S SHORTEST PATH ALGORITHM :—

$$W[i][j] = \begin{cases} \text{weight of edge } (v_i, v_j) & \text{if exists} \\ \infty & \text{if } (v_i, v_j) \text{ does not exist \& } i \neq j \\ 0 & \text{if } i = j. \end{cases}$$

W	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

We define an additional matrix D —

$D^{(k)}[i][j]$ = length of the shortest path from v_i to v_j using the vertices $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices.

$D^{(0)}[2][5]$ = length of $[v_2, v_5] = \infty$

$D^{(1)}[2][5]$ = ~~length of $[v_2, v_5]$;~~
min {length $[v_2, v_5]$, length $[v_2, v_1, v_5]$} = 14

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14$

$D^{(3)}[2][5] = D^{(2)}[2][5] = 14$

$D^{(4)}[2][5]$ = min (length $[v_2, v_1, v_5]$, length $[v_2, v_4, v_5]$,
length $[v_2, v_1, v_4, v_5]$, length $[v_2, v_3, v_4, v_5]$)
$= 5$

$D^{(5)}[2][5] = D^{(4)}[2][5] = 5$.

$\therefore$ Shortest path between v_2 & v_5 is 5

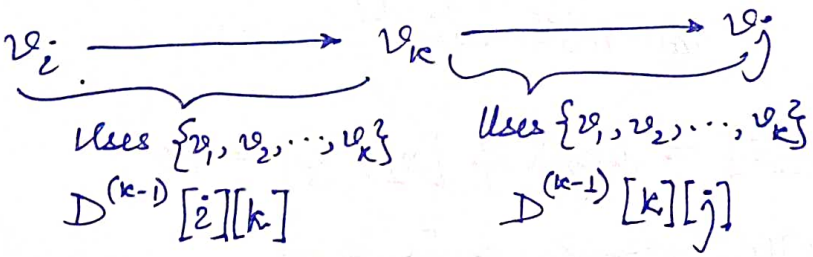
$D^{(0)}[i][j] = W_{ij}$

Establish a recursive property which will help me to compute $D^{(k)}$ from $D^{(k-1)}$

Case I: At least one of the shortest path from v_i to v_j using only vertices in $\{v_1, \ldots, v_k\}$ without using v_k

$\therefore D^{(k)}[i][j] = D^{(k-1)}[i][j]$

Case II: All shortest paths from v_i to v_j using only vertices $\{v_1, \ldots, v_k\}$ as intermediate vertices and has v_k.



$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$$

Considering both Case I and Case II —

$$\boxed{D^{(k)}[i][j] = \min\left(D^{(k-1)}[i][j],\ D^{(k-1)}[i][k] + D^{(k-1)}[k][j]\right)}$$

```
void floyd(int n, int W[][], int D[][]){
    index i, j, k;
    D = W;
    for(k=1; k<=n; k++)
        for(i=1; i<=n; i++)
            for(j=1; j<=n; j++)
                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
}
```

We still need to output one of the paths which is optimal.

"All pair shortest path"

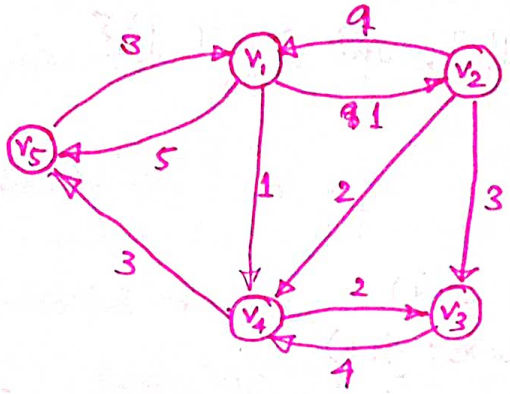
Complexity : $O(n^3)$

Floyd's Algorithm Part 2: Finding the path.

We build the path matrix P[][]

$$P[i][j] = \begin{cases} \text{highest index of an intermediate vertex on the shortest path if one such intermediate vertex exists} \\ 0 \quad \text{otherwise.} \end{cases}$$

```
void floyd2(int n, int W[][], int D[][], int P[][]){
    index i, j, k;  for(i=1; i<=n; i++) for(j=1; j<=n; j++) P[i][j]=0;
    D = W
    for(k=1; k<=n; k++)
        for(i=1; i<=n; i++)
            for(j=1; j<=n; j++)
                if(D[i][j] > D[i][k] + D[k][j]){
                    P[i][j] = k;
                    D[i][j] = D[i][k] + D[k][j];
                }
}
```



W

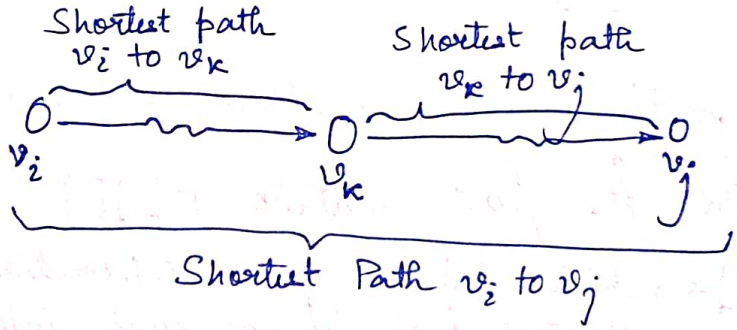
	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

D

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

P

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



How should the path matrix be traversed?

— In order traversal.

```
path (index vi; index vj) {
    if [P[vi][vj] != 0) {
        path (vi, P[vi][vj]);
        print vm || P[vi][vj];
        path (P[vi][vj], vj);
    }
    if (P[vi][vj] == 0) { base case }
}
```

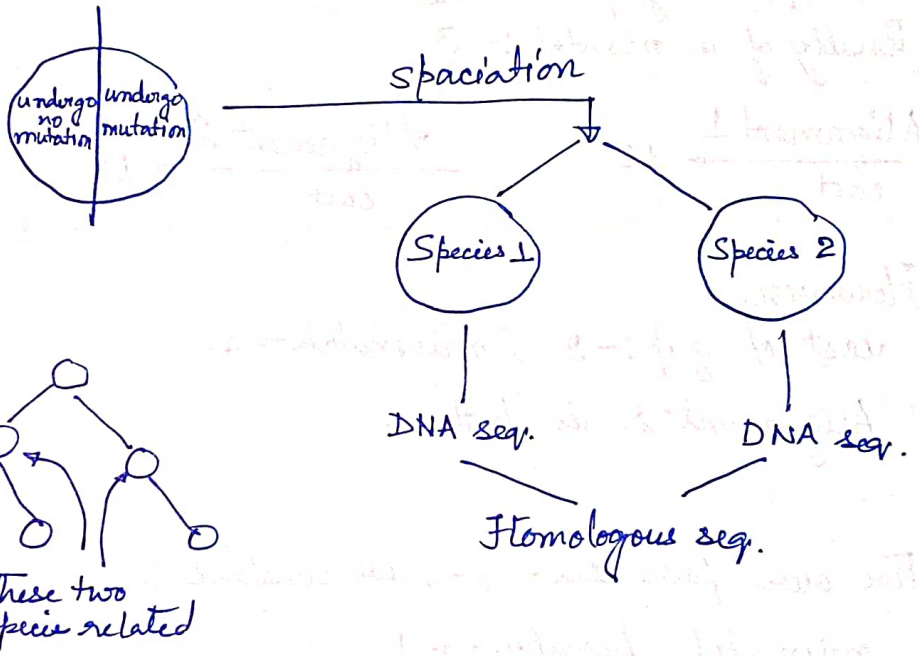
Sequence Alignment :—

Identifying homologous sequences required for the reconstruction of a phylogenetic tree.

DNA → Composed of base pairs; A, G, C, T

DNA sequence → section or a site of the entire macromolecule

DNA mutation →
- Insertion
- Deletion
- Substitution.



Phyllogenetic Tree.

```
A C G T A C A G
| | | | | | | |
T G C A T G T C
```
→ DNA seq.

H_1 : A A C A G T T A C C

H_2 : T A A G G T C A

Align H_1 with H_2 → Use insertion / deletion / substitution
(a gap insertion —)

```
_ A A C A G T T A C C
T A A _ G G T _ _ C A
```
Alignment 1

```
A A C A G T T A C C
T A _ A G G T _ C A
```
Alignment 2

Which alignment is better?

Trying to "well-form" an "ill-formed" problem.

Cost of a gap:- 1
Penalty of a mismatch:- 3

Alignment 1 → cost → 10

Alignment 2 → cost → 11

However,
cost of gap:- 2, mismatch → 1,
Alignment-2 is better.

For our purposes —, we consider
mismatch penalty → 1
gap penalty → 2

```
A  A  C  A  G  T  T  A  C  C
X[0] X[1] X[2] X[3] X[4] X[5] X[6] X[7] X[8] X[9]

T  A  A  G  G  T  C  A
Y[0] Y[1] Y[2] Y[3] Y[4] Y[5] Y[6] Y[7]
```

We define

opt(i, j) as the cost of optimal alignment of the subsequences $x[i, \ldots, 9]$ and $y[j, \ldots, 7]$. Our answer would be opt(0, 0), the optimal alignment of the entire $x[0, \ldots, 9]$, and $y[0, \ldots, 7]$

— x[0] is aligned with y[0] if $x[0] = y[0]$, penalty = 0
else if $x[0] \neq y[0]$, penalty = 1

— x[0] is aligned with gap, gap penalty of 2 at the first site
— y[0] is aligned with gap, gap penalty of 2 at the first site

Let A_{opt} be the optimal alignment of $x[0, \ldots, 9]$ & $y[0, \ldots, 7]$

If x[0] is aligned with y[0] —
→ B is of $X[1 \ldots 9]$ with $Y[1 \ldots 7]$ which is embedded.
We argue that B is the optimal alignment.
(If there is a better alignment C, we could construction a better alignment of $X[0 \ldots 9]$ & $Y[1 \ldots 7]$ than A_{opt})

We define $p = \begin{cases} 0 & \text{if } x[0] = y[0] \\ 1 & \text{if } x[0] \neq y[0] \end{cases}$

$$opt(0, 0) = \min(opt(1, 1) + p,\ opt(1, 0) + 2,\ opt(0, 1) + 2)$$

$$p = \begin{cases} 0 & \text{if } x[i] = y[j] \\ 1 & \text{if } x[i] = y[j] \end{cases}$$

$$opt(i, j) = \min(opt(i+1, j+1) + p,\ opt(i+1, j) + 2,\ opt(i, j+1) + 2)$$

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]
A	A	C	A	G	T	T	A	C	C
T	A	—	A	G	G	T	—	C	A
y[0]	y[1]		y[2]	y[3]	y[4]	y[5]		y[6]	y[7]

Base Condition:

m is length of seq. x

n is length of seq. y.

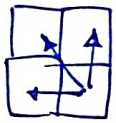
end of x $\cdot (i = m)$ but $j < n$.

$$\left.\begin{cases} opt(m, j) = 2(n-j) \\ opt(i, n) = 2(m-i) \end{cases}\right\} \text{Base condition}$$

Building the table

i \ j	T 0	A 1	A 2	G 3	G 4	T 5	C 6	A 7	— 8
A 0	7	8	10	12	13	15	18	18	20
A 1	6	6	8	10	11	13	14	16	18
C 2	6	5	6	8	9	11	12	14	16
A 3	7	5	4	6	7	9	11	12	14
G 4	9	7	5	4	5	7	9	10	12
T 5	8	8	6	4	4	5	7	8	10
T 6	9	8	7	5	3	3	5	6	8
A 7	11	9	7	6	4	2	3	4	6
C 8	13	11	9	7	5	3	1	3	4
C 9	14	12	10	8	6	4	2	1	2
— 10	16	14	12	10	8	6	4	2	0

Filling up the table in diagonal to diagonal ~~to~~



Extracting the alignment part

1. Start from [0][0]
2. (a) slot [0][1]
   is [0][0] built from [0][1]
   i.e. opt[0, 1) + 2 == 7? 8 ≠ 7, no
   (b) slot [1][0]
   opt(1,0) + 2 == 7? 6+2 ≠ 7, no
   (c) slot [1][1] ✓
   opt(1, 1) + p == 7? 6+1 == 7, yes!

⋮

Extracting the actual alignment

1. Start from bottom right & follow highlighted path
2. If we move slot [i][j] in the diagonal direction
   ⇒ align x[i] with y[j]
3. If we move directly up to come at slot i, j
   ⇒ x[i] is placed in the x sequence & gap is placed at the y sequence.
4. If we move directly left to come at slot i, j
   ⇒ y[j] is placed in the y sequence & gap is placed at the x sequence.

Optimal Alignment:—

A A C A G T T A C C
T A _ A G G T _ C A

Practice Assignment:—

Find the Levenshtein distance between two strings

CAT
BAT

min operations (insertion (1) / deletion (1) / substitution (1)) to get to target word from source words.

Divide and Conquer — Dynamic Programming

They use solutions to the subproblems

Top-Down

Bottom up (Subproblems repeat)

GREEDY ALGORITHMS

Instead of taking min/max/___, can we identify a greedy property that will always be true for optimal solution.

The coin change problem:

Suppose we have to make a change of n Rupees using coins of denomination: 1, 2, 5 (Sufficient number), using minimum number of coins.

DP:-

Change [i] := min number of coins required to make a change for i Rupees.

Final answer: Change [n]

Change [i] = 1 + min (Change [i-1], Change [i-2], Change [i-5]);

Base case:

Change [1] = 1
Change [2] = 1
Change [3] = 2
Change [4] = 2
Change [5] = 1

Complexity: O(n)

Greedy Property: If a change with a coin of larger denomination, we will choose that.

```
If (n ≥ 5) select 5
else if(n≥2) select 2
else select 1.
```

∴ To make a change of n Rupees,
We use $\left\lfloor \frac{n}{5} \right\rfloor$ 5 Rs. coin

Complexity : O(1)

Proving that this will give you an optimal solution.

Greedy solution : G
Assume there is a optimal solution : $O \neq G$
Starting from O, we construct another optimal solution O', such that $O' \equiv G$

$G = \langle t_5, t_2, t_1 \rangle$
$O = \langle o_5, o_2, o_1 \rangle$

If $O \equiv G$, we are done

Let $O \neq G$, these numbers differ atleast in one position.

Assume $o_5 \neq t_5$

1. $o_5 > t_5$ ⟶ Not possible due to greedy property
2. $o_5 < t_5$ ⟶ Optimal solution is using 1, 2 Rs coins to make a change for at least 5 Rs —
   Then it uses
   - 2, 2, 2 ⟶ 5, 1
   - 2, 2, 1 ⟶ 5
   - 2, 1, 1, 1 ⟶ 5
   - 1, 1, 1, 1, 1 ⟶ 5

We show that in all these cases, we can actually do something better, therefore it contradicts the optimality of O

∴ $o_5 = t_5$.

Assume $O_2 \neq t_2$

We are left with at most 4 Rupees

1 → ✗

2 → $t_2=1, t_1=0$ $\quad O_2=0,\ O_1=2$ ✗

3 → $t_2=1, t_1=1$ $\quad O_2=0,\ O_1=3$ ✗

4 → $t_2=2, t_3=0$ $\quad O_2=0,\ O_1=4$ ✗

$O_2=1,\ O_1=2$ ✗

$O_2 = t_2$

$\Rightarrow O_1 = t_1$

$\Rightarrow O \equiv G$

What if we use notes of arbitrary denomination

Suppose 1, 4, 5, then it wont work.

$8 = \underbrace{5 + 1+1+1}_{\text{greedy}} = \underbrace{4+4}_{\text{optimal}}$.

"If the greedy approach is correct, we can prove optimality. If greedy approach is not correct, we should provide a counter example"

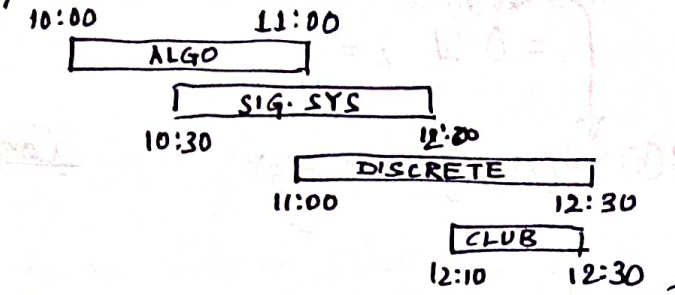
We can always apply DP

change[i] = 1 + min (change[i-5], change[i-4], change[i-1].

Activity Selection Problem

n requests for a classroom

request i →



Optimization Problem — Allow as long many full activities as possible.

n Requests, $S(i) < f(i)$

Two requests i and j are said to be compatible if both are allowed at the same time ($S(j) \geq f(i)$ or $f(j) \leq S(i)$).

Optimization problem — Select a compatible subset of requests is of max size

* Brute force: Try all subsets (exponential time)
* DP: Assume that they are sorted with $f(i)$

$$a_1, a_2, a_3, \ldots, a_n$$
$$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_n$$

Defining subproblem

T_{ij} = set of all requests that start after a_i finishes and finish before a_j starts.

S_{ij} = Optimal solution for T_{ij}

$C[i, j] = |S_{ij}|$

Final solution:

2 dummy requests a_0, $f_0 \leq \min\{S(i)\}$; a_{n+1}, $S_{n+1} \geq f_n$ } Final solution $= C[0, n+1]$

$$C[i,j] = \max_{a_k \in T_{ij}} \{1 + C[i,k] + C[k,j]\}$$

$= 0$ if $i = j$

$s(k) \geq f(i)$, $f(k) \leq s(j)$

Complexity: $O(n^3)$

* Greedy:

$a_0 \quad a_1 \quad \cdots \quad a_n \quad a_{n+1}$

$f_1 \leq \cdots \leq f_n$

Can I identify a greedy property/choices

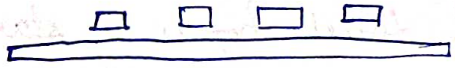
Greedy choice 1: choose activity a_1. / 1': largest start time (activity with the earliest finish time)

Greedy choice 2: choose activity with the earliest start time.

Greedy choice 3: Choose activity with shortest activity [$f(i) - s(i)$ is smallest]

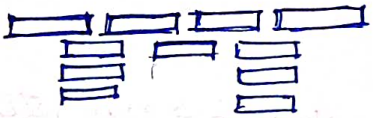
Greedy choice 4: Choose activity with minimum number of incompatibilities.

GC 2: Cex



GC 3: Cex

GC 3 Cex



GC 1: Earliest Finish Time.

T_{ij}, let a_m be the element in T_{ij} with minimum finish time.

a_m is in some optimal solution.

Proof: Let S_{ij} be an optimal solution, with a_{opt} being the earliest finish time.

Case I: $a_{opt} = a_m$ ⟶ done

Case II: $a_{opt} \neq a_m$

$f_{opt} < f_m$

We construct S'_{ij} as ~~$S'_{ij} = (\{S_{ij}\} - \{a_{opt}\})$~~

$$S'_{ij} = (S_{ij} - \{a_{opt}\}) \cup \{a_m\}$$

Is S'_{ij} also compatible? yes. (all activities were after finish time)

S'_{ij} is optimal & $a_m \in S'_{ij}$. [Proved] ■

Complexity: $O(n)$; $O(n \log n)$ if we need to sort.

Greedy solution:

```
activity_selection ( s[ ], f[ ])   // sorted as per f_i
    A = {1};
    last_selected = 1;
    for m = 2 to n
        if s_m > f_last_selected
            A = A ∪ {m};
            last_selected = m;
```

Problem: (Activity Scheduling)

- → Single request
- → set of n request
- → Each request comes with a time requirement (t_i), time taken to completion, continuously; and a deadline d_i
- → The task is to schedule all request, such that the maximum lateness is minimized.

Request i starts at $s(i)$ and ends at $s(i)+t_i = f(i)$

Case I: $d_i > f_i$ (on time), lateness $= 0 = l_i$

Case II: $d_i < f_i$ (late)

lateness $= f(i) - d_i = l_i$

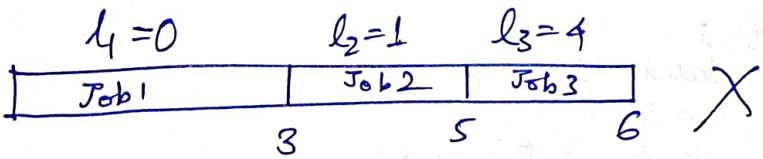
We want to minimize lateness,

minimize $\max_i l_i$

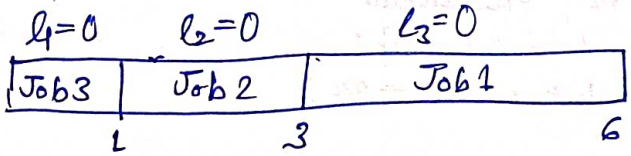
Job 1 — $t_i = 3$ — $d_1 = 6$

Job 2 — $t_i = 2$ — $d_2 = 4$

Job 3 — $t_i = 1$ — $d_3 = 2$



X



Greedy choices:

Approach 1:→ Choose job with the earliest deadline

Approach 2:→ Choose job having minimum $d_i - t_i$

Approach 3:→ Choose the job with smallest duration, that is minimum t_i.

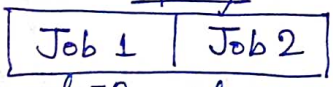
"For each approach try hard # to find counter example"

Approach 3:– Cex

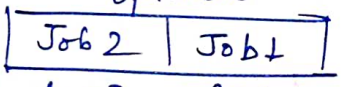
Job 1: $t_1 = 1$ $d_1 = 100$

Job 2: $t_2 = 10$ $d_2 = 10$

Greedy



Optimal

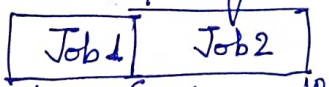


Approach 2:– Cex

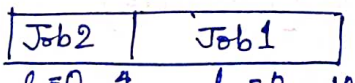
Job 1: $t_1 = 6$ $d_1 = 10$, $d_1 - t_1 = 4$

Job 2: $t_2 = 4$ $d_2 = 9$, $d_2 - t_2 = 5$

Greedy



Optimal



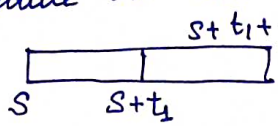
Approach 1: Choosing job with earliest deadline.

Proof:

The algorithm is just sorting the jobs according to deadline.

$$d_1 \leq d_2 \leq \cdots \leq d_n$$

Schedule in this order



$$\text{finish time} = s + \sum_{i=1}^{n} t_i$$

Greedy choice (G): Schedule jobs in increasing order of deadline —

$$d_i < d_j \iff i \text{ is scheduled before job } j.$$

Assume there is an optimal solution O.

if $O \neq G$

We change O such that optimality is maintained.

In O, $\exists\ i, j$; i is scheduled before j but $d_i > d_j$ (inversion)

Case I: O has no inversion $\Rightarrow O = G$

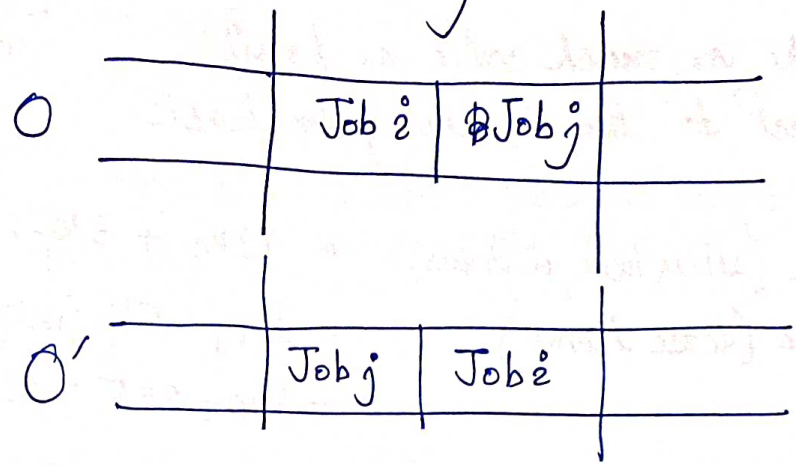
Case II: has at least one inversion

A scheduled before B

There will be two consecutive jobs i & j such that the deadline decreases for the first time

$d_i > d_j$; and job i is scheduled just before job j in O.

Now we swap i and j; hence removing an inversion



Suppose each request r starts at $s(r)$ and finishes at $f(r) = s(r) + t_r$.

$$L = \max_r l_r$$

$O \Rightarrow L = \max \{l_1, l_2, \ldots, l_i, l_j, \ldots, l_n\}$

$O' \Rightarrow \bar{L} = \max \{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_i, \bar{l}_j, \ldots, l_n\}$

Now, $\bar{l}_k = l_k \quad \forall\, k \in \{1, 2, \ldots, n\} \setminus \{i, j\}$

If $L = \max(l_i, l_j)$. then $\bar{L}$ can be different.

$\boxed{l_j = f(j) - d_j}$, $l_i = f(i) - d_i$ $\quad [\because d_i > d_j \;\&\; f(i) < f(j)]$

$\bar{l}_j = \bar{f}(j) - d_j \le f(j) - d_j \le l_j$ $\quad [\because \bar{f}(j) \le f(j)]$

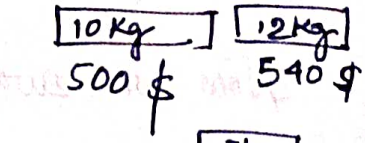
$\bar{l}_i = \bar{f}(i) - d_i = f(j) - d_i \le f(j) - d_j \le l_j$ $\quad [\because d_j < d_i]$

Inversion argument.

Complexity: (Sorting) $\longrightarrow$ $O(n \log n)$

Knapsack:

Knapsack Capacity (W = 15 Kg)



Take as much value as possible subject to the capacity of knapsack

Case 1 (either take or leave) → 12kg + 3kg : 630$

Case 2 (loose items) → 10Kg + 5kg (12Kg)

= 500 + 225 = 725$

First we compute value per Kg of the item, we take as much as we can for first item then for the second item and so on.

Complexity: $O(n \log n)$ [sorting]

(Very similar to coin change)

Proof:-

Given $\{1, \ldots, n\}$ items sorted as per value/weight ratio f.

$$f_1 \geq f_2 \geq f_3 \geq \ldots \geq f_n$$

Let $G = \langle x_1, \ldots, x_n \rangle$

$x_i \in [0, 1]$, indicate fraction of item x_i

$\therefore$ Knapsack capacity $= W = \sum_{i=1}^{n} x_i w_i$

Let $O = \langle y_1, \ldots, y_n \rangle \neq G$. $\therefore \exists\, i$ such that $y_i \neq x_i$

We take the first occurrence of such i, $[\because x_j = y_j,\ j < i]$

Now, due to greedy choice, $x_i > y_i$ [as $x_i < y_i$ cannot happen]

Now, we change y_i to x_i in O to O'

Now, we will have to reduce some weights to make a decrement of $(x_i - y_i)w_i$ from $\{i+1, \ldots, n\}$.

Now, we argue that

value of O' $\geq$ value of O

Now, difference of value O to O'

$$= (x_i - y_i) w_i p_i - \sum_{k=i+1}^{n} t_k v_k \quad , \quad \left[\text{such that } \sum_{k=i+1}^{n} t_k w_k = (x_i - y_i) w_i\right]$$

$$= (x_i - y_i) w_i p_i - \sum_{k=i+1}^{n} t_k w_k p_k .$$

$$\geq (x_i - y_i) w_i p_i - p_{i+1} \sum_{k=i+1}^{n} t_k w_k \qquad [\because p_{i+1} \geq p_k,\ k \geq i+1]$$

$$= (x_i - y_i) w_i p_i - p_{i+1} (x_i - y_i) w_i$$

$$= (x_i - y_i) w_i (p_i - p_{i+1}) \geq 0 \qquad [\because x_i > y_i\ ;\ w_i > 0 \text{ and } p_i \geq p_{i+1}]$$

[Proved] ∎

DISJOINT SET DATA STRUCTURE

Abstract Data Type (ADT)

For designing an algorithm, we need certain operations.

Operational view

Stack → push, pop, peek, IsEmpty, ...
Queue → enqueue, dequeue, peek, IsEmpty, ...

Implementation view

Stack → linked list, array.
Queue → linked list, queue.

ADT: We agree on a set of operation.

Disjoint Set / Union Find as ADT

There are n elements $x_1, \ldots, x_n$ divided into disjoint set. Initially, each element is a group in itself. The operations are —

① MakeSet(x):- Make a singleton element with x — id "x"

② FindSet(x):- Given x, find id. of the set it belongs to

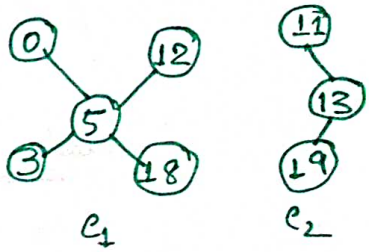
③ Union(x, y):- Create a single set out of two sets containing x & y. New set can have id any of the set.

Example:-

Think of a social network

$V: \{0, 1, 2, \ldots, n-1\}$ (set of people)

and a set of edges that define E



C_1 C_2

Task:-

Find out connected components, For each person, print the components it belongs to.

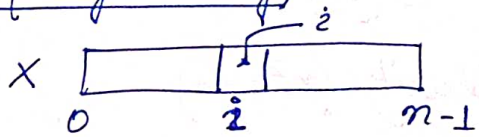
Additional Constraint:-

For each $x \in V$, do : makeSet(x)
For each $(x, y) \in E$ do:
    if find(x) ≠ find(y)
        Union(x, y)
For each ~~(as?)~~ $x \in V$
    output — Person 'x' belongs to component 'find(x)'

Complexity: O(n) makeSets + O(m) finds + O(n) unions.

Implementation of Disjoint Sets:

→ Array (single array)

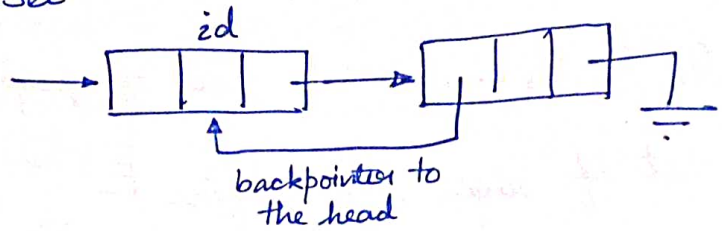


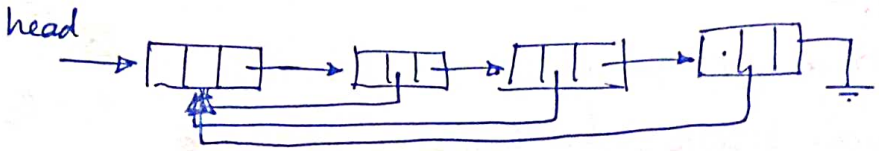
X[i]: id of the set containing i.

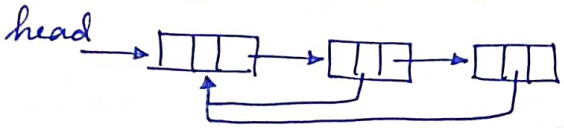
O(n) → makeSet
O(1) → find
O(n) → Union [A = find(x) ≠ find(y) = B. Change id of all elements of B to A]

Implementation using linked list

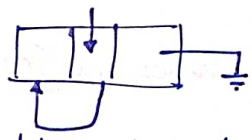
~~Make Set~~



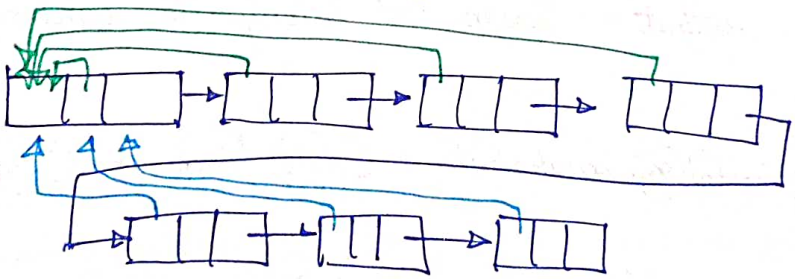




$O(n)$ linked lists.

Make Set : 

For merging, we put one linked list at the tail of another ; change back pointers



Union $(x, y) = O(n)$

overall complexity $(n^2 + m)$

Now, we append the ~~last~~ smaller linked list to the larger list

worst case ⟶ the smaller list can have $\frac{n}{2}$ elements.

Take a deeper look:

Take a particular node ⟶

First time size ≥ 2
Second time size ≥ 4
Third time size $\geq 2^3$
⋮

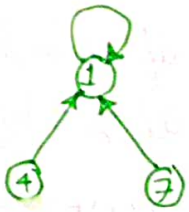
That means for each node, we can only update it $O(\log n)$ times.

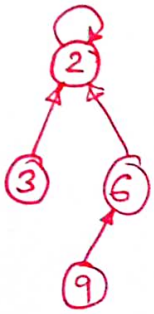
Total backpointer updates $O(\cancel{n^2}\ n\log n)$

Amortized analysis

Implentation: Rooted Tree

- Not a binary tree
- Inverted tree





MakeSet: $O(1)$

Find: $O(h) = O(n)$

Union: $O(1)$.

~~Find complexity~~

Total Complexity: $O(n\log n + m)$

Now we connect smaller tree to larger tree, $h \approx O(\log n)$

If you follow height balancing, a tree of height h must have atleast 2^h nodes.

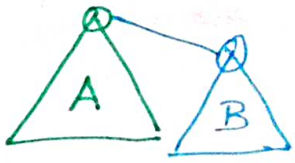
Height 1:  2 nodes

$A \rightarrow h(A)$ $\quad h(B)$

$\geq 2^{h(A)} \quad \geq 2^{h(B)}$

~~A becomes~~

A is larger than B $\Rightarrow$ B becomes subtree of A



$h = \max\{h(A), h(B)+1\}$

if $h = h(A)$, then $|A \cup B| \geq |A| \geq 2^{h(A)}$

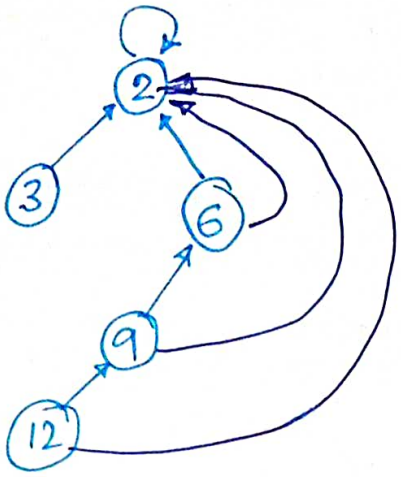
if $h = h(B)+1$, then ~~$|A \cup B| \geq |B| + |B| \geq 2^{h(B)+1}$~~.

$|A \cup B| \geq |B| + |B| \geq 2^{h(B)+1}$

Find: $\log n$

Total complexity: $O(m \log n + n)$

2. Path compression :-



Complexity: $O(m \log^* n)$

for $n \leq 2$, $\log^* n = 1$

for $n > 2$ $\quad 1 + \log^*(\lceil \log_2 n \rceil)$

for $n \leq 2^{65536}$ $\quad \log^*(n) \leq 5$

∴ Rooted Trees are default ds for Disjoint sets

NAME: ANUBHAV DHAR ROLL NO: 20CS30004

GRAPH PROBLEMS: GREEDY EDITION

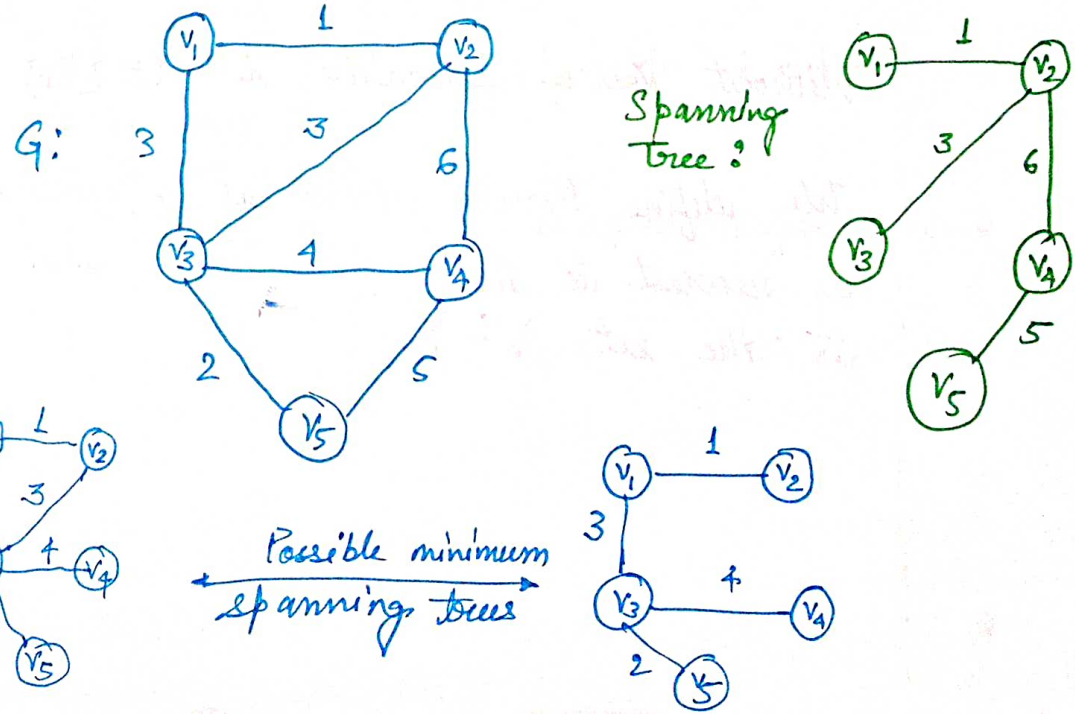
- Identification of minimum spanning tree in an undirected weighted graph.
- Single source shortest path finding in a directed weighted graph.

SPANNING TREES

A spanning tree is a connected subgraph of a graph G having all the nodes of G and is a tree.

Typically G is a weighted undirected graph.

A minimum spanning tree is a spanning tree of minimum aggregate edge weights (minimum weight)



NAME: ANUBHAV DHAR ROLL NO: 20CS30004

How to do this :—

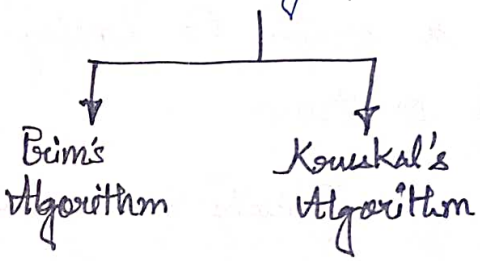
T is a spanning tree of G.

$\Rightarrow$ Vertices of T = Vertices of G.

Let $F \subseteq E$ be the edges of T.

Let $T(V, F)$ be the notation for the minimum spanning tree of $G(V, E)$

We need to identify some "locally optimal" strategy.



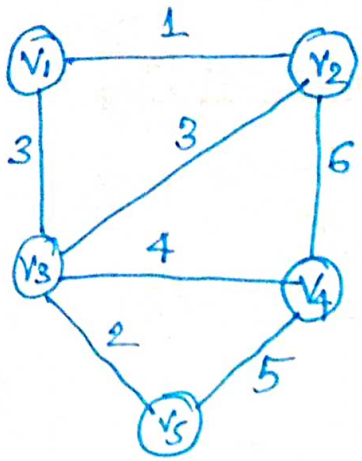
PRIM'S ALGORITHM

Start with an empty subset F
We form a subset of vertices $Y \subseteq V$, initialized with one arbitrary vertex.

Without Loss of Generality, let $Y = \{v_1\}$

We define Nearest Vertex as a vertex that is "nearest" to the vertices in Y and belongs to the set $V - Y$.

NAME: ANUBHAV DHAR ROLL NO: 20CS30004



$Y = \{v_1\}$

The nearest vertex to Y from V−Y is v_2

- We keep adding nearest vertices Y from V−Y and add it to Y, consequently updating F
- Terminate when $|V| = |Y|$

In any greedy solution, we have

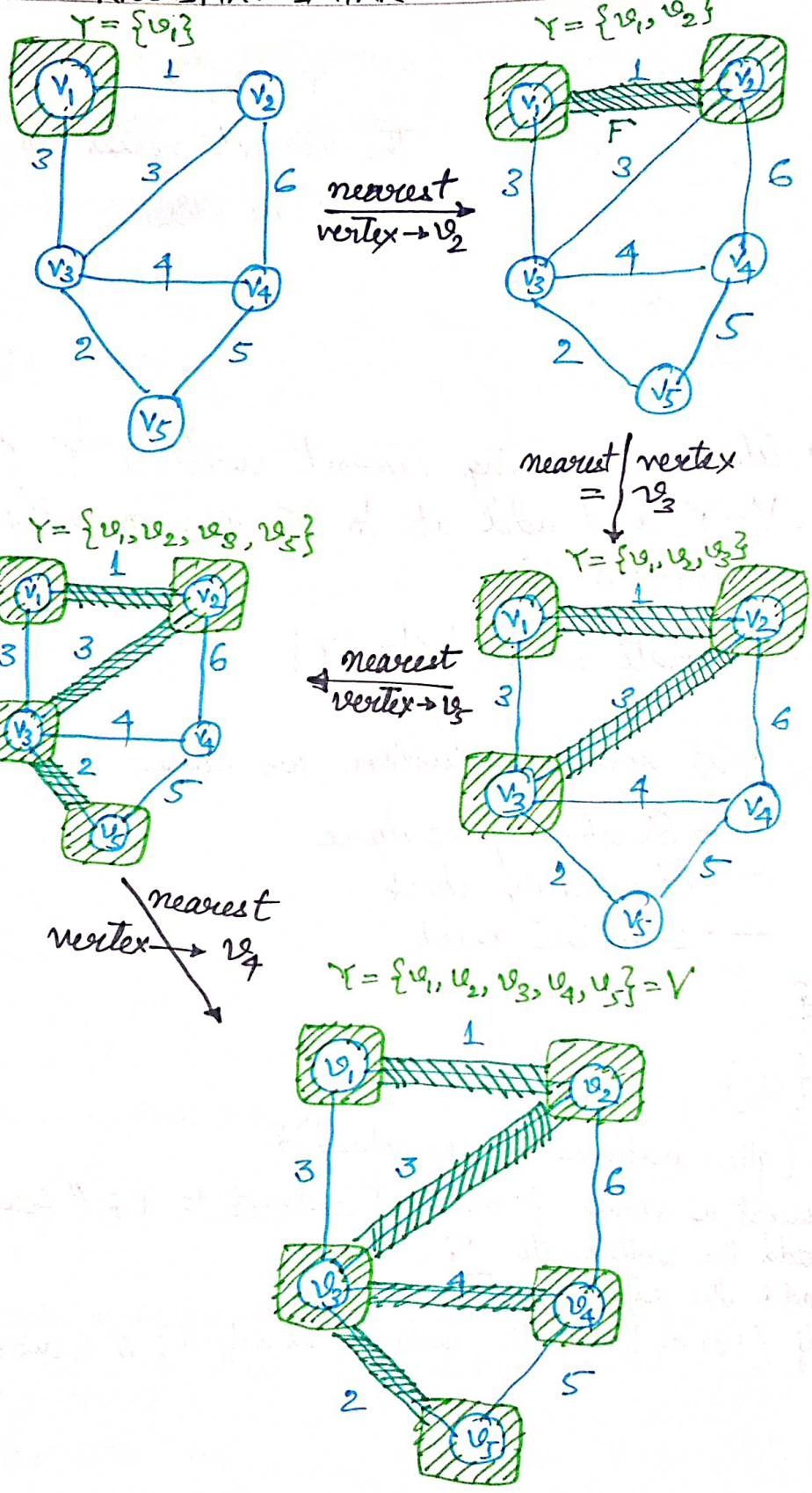
— Selection procedure
— Feasibility check
— Solution check

$F = \phi$

$Y = \{v_1\}$

while (the instance is not solved) {
select a vertex from V−Y nearest to Y; // Selection & feasibility
add the vertex to Y;
add the edge to F;
if (|V| == |Y|) the instance is solved; // Solution check
}

NAME: ANUBHAV DHAR ROLL NO.: 20CS30004



A minimum spanning tree

NAME: ANUBHAV DHAR ROLL NO.: 20CS30004

Coding Prim's Algorithm –

Adjacency Matrix:–

$$W[i][j] = \begin{cases} \text{weight of edge } v_i - v_j \\ \infty \text{ if } (v_i, v_j) \notin E,\ i \neq j \\ 0 \text{ if } i = j \end{cases}$$

nearest[i] := index of the vertex in Y nearest to the node v_i

distance[i] := weight of edge $(v_i, v_{nearest[i]})$

$Y = \{v_1\}$

nearest[i] = 1, $\forall i$, $i \neq 1$.

distance[i] = W[1][i], $\forall i$

In every step, Y is expanded with that vertex which has distance[i] that is smallest. Let us call index of this vertex as vnear.

The vertex indexed by vnear is added to Y

distance[vnear] = -1 [do not use it in subsequent iteration]

NAME: ANUBHAV DHAR ROLL NO.: 20CS30004

```
void prim(int n, int W[][], set_of_edges F){
    index i, vnear;
    int min;
    edge e;
    index nearest[2...n];
    int distance[2...n];
    F = ϕ;
    for(i=2; i<=n; i++){
        nearest[i] = 1;
        distance[i] = W[1][i];
    }
    repeat(n-1 times){
        min = ∞;
        if(0 ≤ distance[i] < min){
            min = distance[i];
            vnear = i;
        }
        e = edge(vnear, nearest[vnear]);
        add e to F;
        distance[vnear] = -1;
        for(i=2; i<=n; i++){
            distance[i] = W[i][vnear];
            nearest[i] = vnear
            if(W[i][vnear] < distance[i]){
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
            }
        }
    }
}
```

Complexity : $O(n^2)$

NAME: ANUBHAV DHAR ROLL NO: 20CS30004

$G = (V, E)$

$F \subset E$ is termed as a "promising subset" if edges could be added to F, to make an a minimum spanning tree.

Lemma 1:-

$G = \langle V, E \rangle$

Let F be a promising subset.

Let Y be the set of vertices that ~~connected~~ the edges in F.

If e is an edge of minimum weight that connects a vertex from Y to $V-Y$, then $F \cup \{e\}$ is promising.

Proof:-

Part 1:-

Since F is promising, we can assume in "future" it is going to lead us to a minimum spanning tree F', possibly through union of some edges.

$$F \subseteq F'$$

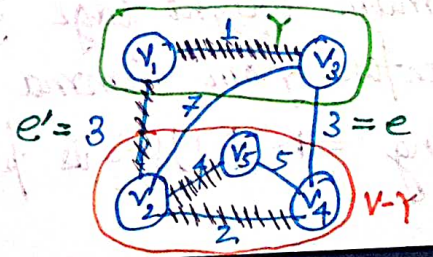
(V, F') will be the final MST at the end.

If $e \in F' \Rightarrow F \cup \{e\} \subseteq F' \Rightarrow F \cup \{e\}$ is promising.

Part 2:-

If $e \notin F'$

Now since (V, F') is an MST, $F' \cup \{e\}$ is a simple cycle and e will be a part of this cycle.



Now, $F' \cup \{e\} - \{e'\}$ will be a spanning tree again.

Now, due to minimality of e,

$$w(e) = w(e')$$

$\therefore F' \cup \{e\} - \{e'\}$ is a valid MST.

Further $e' \notin F$, because e' connects ~~as a vertex Y~~ a vertex from Y and a vertex from $V-Y$.

$$\Rightarrow F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$$

Hence $F \cup \{e\}$ is a promising subset as well.

[Proved] ∎

Theorem 1: Prim's Algorithm always produces an MST.

Equivalently, we may show that F is promising after execution of each step.

Proof:-

[Basis] Φ is trivially promising.

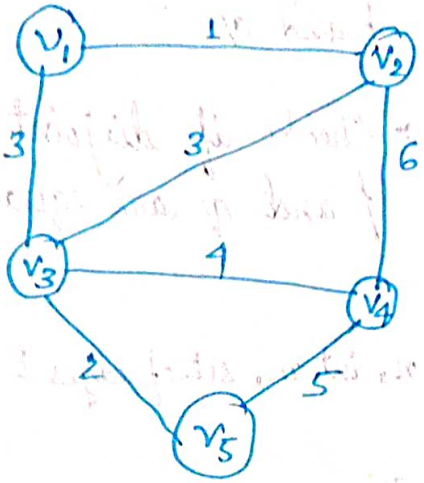
[Hypothesis] Let at any step, F constructed so far be promising.

[Induction] We need to show $F \cup \{e\}$ is promising. Now, by algorithm, e is the minimum weight edge from Y to $V-Y$. Therefore from Lemma 1,

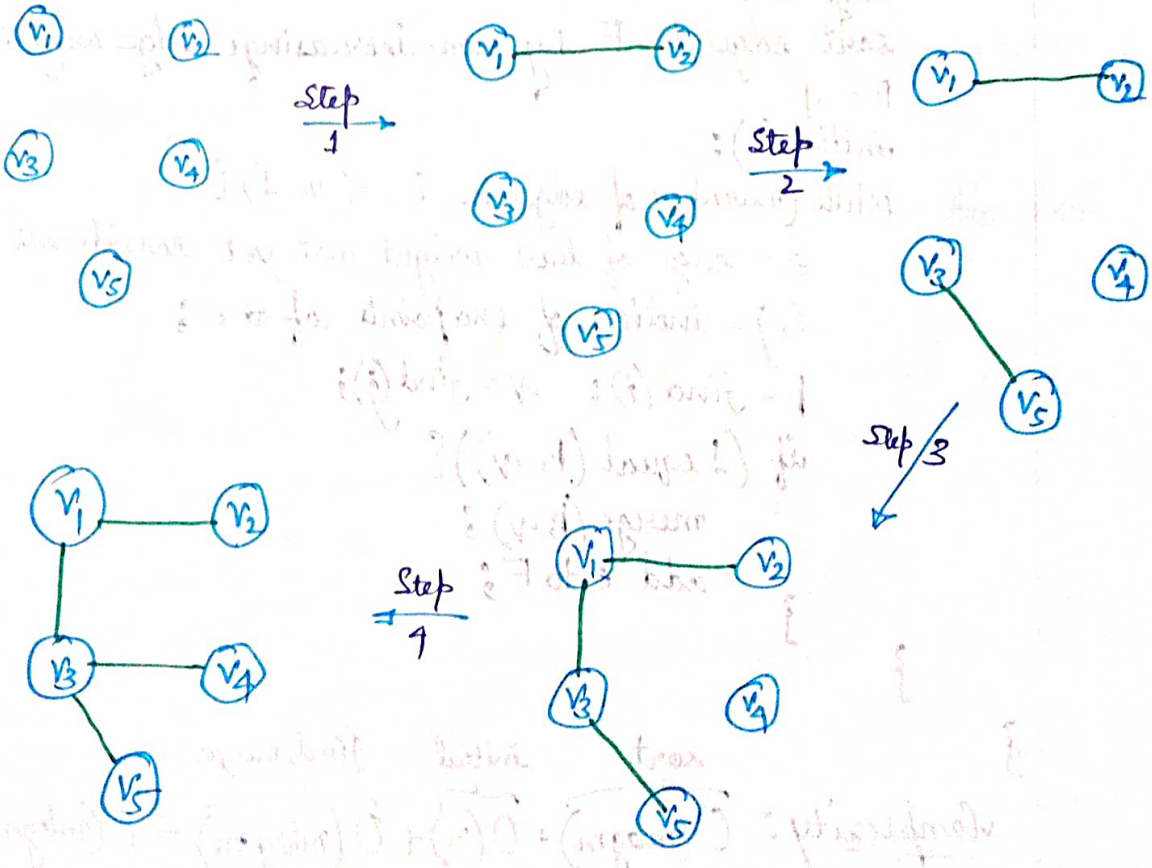
F is promising $\Rightarrow F \cup \{e\}$ is promising.

KRUSKAL'S ALGORITHM

- Make singleton disjoint sets using single nodes
- Sort edges according to increasing order of edgeweight.
- Connect edges nodes with disjoint set :
- Stop when every node is connected.



(v_1, v_2)
(v_3, v_5)
(v_1, v_3)
(v_2, v_3)
(v_3, v_4)
(v_1, v_5)
(v_2, v_4)



index i;
set_pointer p, q;
initial(n) → initializes n disjoint subsets indexed as 1 to n.

p ⇒ find(i) → makes p point to the set containing i

merge(p, q) → merges disjoint sets containing p and q

equal(p, q) → Checks if disjoint sets of p and q are equal.

```
void Kruskal (int n, int m, set_of_edges E, set_of_edges &F){
    index i, j;
    set_pointer p, q;
    edge e;
    sort edges in E by non-increasing edge weight;
    F = ϕ
    initial(n);
    while (number of edges in F < n-1){
        e = edge of least weight not yet considered;
        i, j = indices of endpoints of e;
        p = find(i); q = find(j);
        if (!equal(p, q)){
            merge(p, q);
            add e to F;
        }
    }
}
```

Complexity: $\overbrace{O(m\log m)}^{\text{sort}} + \overbrace{O(n)}^{\text{initial}} + \overbrace{O(m\log m)}^{\text{find, merge}} = O(m\log m)$

For sparse graph → $m \sim O(n)$;
For dense graph → $m \sim O(n^2)$; Kruskal → $O(n^2 \log n)$

For sparse graph ⟶ Kruskal
For dense graph ⟶ Prim's

DIJKSTRA'S ALGORITHM

(Single Source Shortest Path)

Y → The vertex from which the shortest path needs to be computed

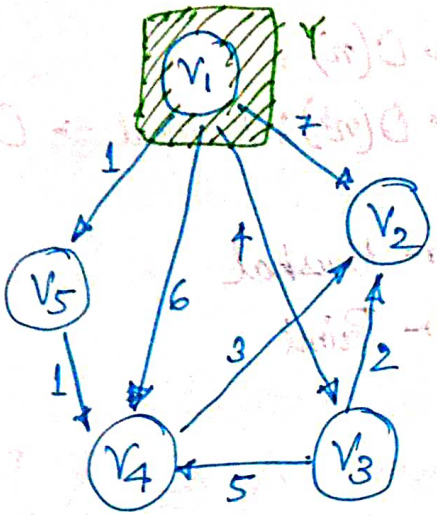
$Y = \{v_1\}$, $F = \phi$.

We add the nearest vertex v to v_1 to Y and (v_1, v) to F.

V~F → Check the paths from v_1 to the vertices V-Y having intermediate vertices Y.

A shortest path among these paths ~~the pat~~ will be the shortest.

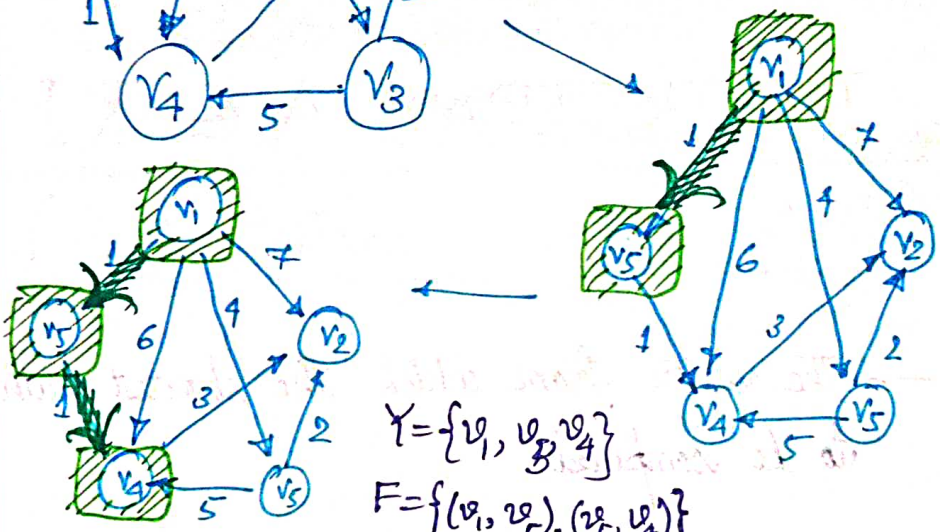
We add the end vertex to Y and the edge to F.



$Y = \{v_1\}$

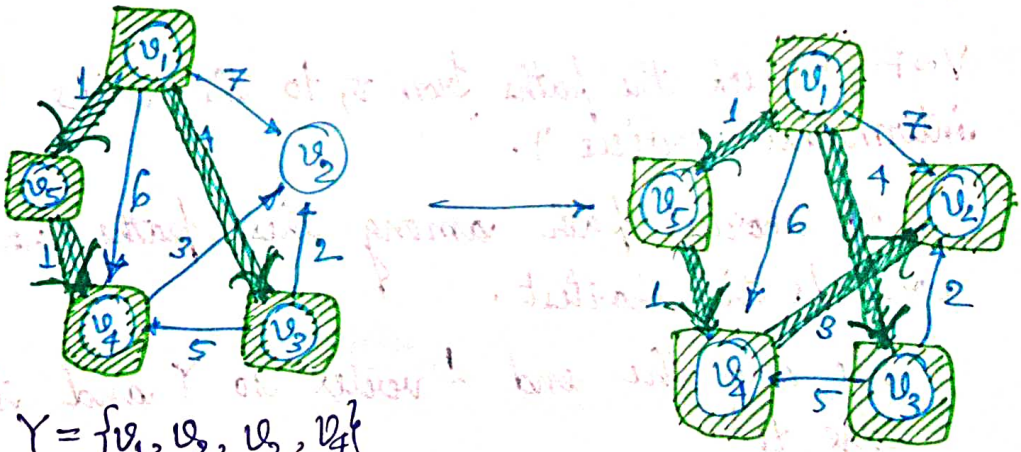
nearest vertex: v_5

$Y = \{v_1, v_5\}$, $F = \{(v_1, v_5)\}$



$Y = \{v_1, v_5, v_4\}$

$F = \{(v_1, v_5), (v_5, v_4)\}$



$Y = \{v_1, v_2, v_3, v_4\}$

$F = \{(v_1, v_5), (v_5, v_4), (v_1, v_3)\}$

$Y = \{v_1, v_2, v_3, v_4, v_5\} = V$

$F = \{(v_1, v_5), (v_5, v_4), (v_1, v_3), (v_4, v_2)\}$

touch[i] := index of the vertex v in Y such that edge (v, v_i) is the last on the current path from v_1 to v_i using vertices present in Y as intermediate.

length[i] := length of path from v_1 to v_i with vertices present in Y.

```
void dijkstra(int n, int W[][], set_of_edges F){
    index i, vnear;
    edge e;
    index touch[2...n];
    int length[2...n];
    F = φ;
    for(i=2; i<=n; i++){
        touch[i] = 1;
        length[i] = W[1][i];
    }
    repeat (n-1 times){
        min = ∞;
        for(i=2; i<=n; i++){
            if(0 ≤ length[i] <min){
                min = length[i];
                vnear = i;
            }
        }
        e = (touch[vnear], vnear);
        add e to F;
        for(i=2; i<=n; ++i){
            if(length[vnear] + W[vnear][i] < length[i]){
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;
            }
        }
        length[vnear] = -1;
    }
}
```

// Complexity : $T(n) = O(n^2)$

Knapsack Problem

Knapsack ← some weight of gold.

↓

max wt. = W

Each item has a price and a weight.

Maximize total value of items that you steal.

$$S = \{item_1, item_2, item_3, \ldots, item_n\}$$

w_i = weight of item i

p_i = profit/value of item i.

W = max weight the knapsack can hold.

0-1 Knapsack: You either take an item or don't.

$\sum_{item_i \in A} p_i$ is maximized subject to $W \geq \sum_{item_i \in A} w_i$.

Greedy strategy: Take items of largest profit.

If the large profit items have very large weight, then it might not be optimal.

W = 30 lb

25 lb ⟷ 10$

10 lb ⟷ 9$

10 lb ⟷ 9$

Greedy: 10$ optimal: 18$

Greedy strategy 2: Take item with maximum $\frac{p_i}{w_i}$

5 lb ⟷ \$50
10 lb ⟷ \$60
20 lb ⟷ \$140

$W = 30$ lb.

item$_1$ → $\frac{\$50}{5} = \$10$
item$_2$ → $\frac{\$60}{10} = \$6$
item$_3$ → $\frac{\$140}{20} = \$7$

Greedy → \$190

optimal → \$200

(This works for fractional knapsack)

DP Solution (Yay!):

A be optimal for $S = \{item_1, item_2, \ldots, item_n\}$

Case I: $item_n \notin A$.

A is optimal for $S - \{item_n\}$

Case II: $item_n \in A$

Then $A - \{item_n\}$ is optimal for $S - \{item_n\}$ with weight not exceeding ~~$W - w_n$~~ $(W - w_n)$.

For $i > 0$ and $w > 0$

$P[i][w]$ be the optimal profit obtained when choosing items only for the first i items under the restriction that the total weight cannot exceed 20.

$$P[i][w] = \begin{cases} \max(P[i-1][w], p_i + P[i-1][w-w_i]) & w_i \le w. \\ P[i-1][w] & w_i > w \end{cases}$$

Max profit $\rightarrow$ $P[n][W]$

HEAPS

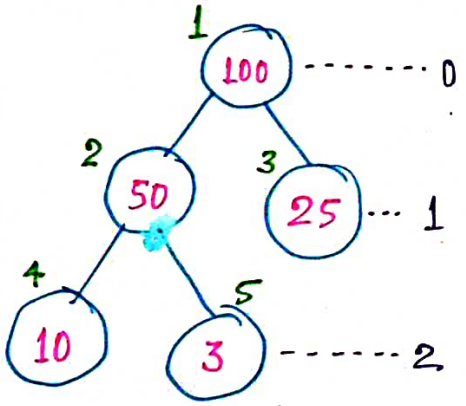
Also known as Priority Queue

A heap is a binary tree with the following property:—

* Structure: It is an almost complete binary tree (All but the last label must be completely full); and in the last level, nodes appear to the left).

* Values: Any node u, if it is parent is p, $val(p) \geq val(u)$ [for max heap]

(In min heap $val(p) \leq val(u)$)



Max heap

A max heap of max level l has
2^i nodes in level i, $0 \leq i < l$
and $[1, 2^l]$ nodes in level l

x → left child → $2x$
x → right child → $2x+1$
x → parent → $\lfloor \frac{x}{2} \rfloor$

So we can represent a max heap with an array. (Due to the structure property)

X	100	50	25	10	3
0	1	2	3	4	5

Priority Queue ADT (Heaps):-

Jobs: Each job has a priority number —

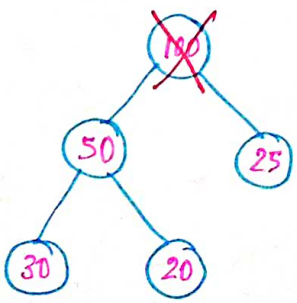
① Find the job with max priority — Find Max
② Remove the job with max priority — Delete Max
③ Insert the new job in approximate position. — Insert.

	Find Max	Delete Max	Insert
Unsorted array	$O(n)$	$O(1)$	$O(1)$
Sorted array	$O(1)$	$O(1)$	$O(n)$
Heaps	$O(1)$	$O(\log n)$	$O(\log n)$

Max Heap:- [A heap on array A[]]

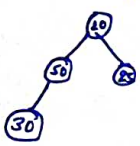
① Find Max: A[1] ($O(1)$ operation)

② Delete Max:



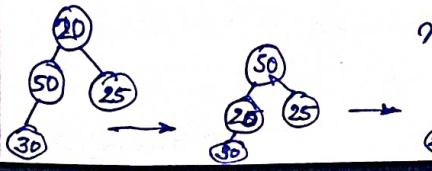
- Delete 100
- Copy the value of the last leaf node to root

```
A[1] = A[size];
size--;
```



(Subsequently we don't change the structure)

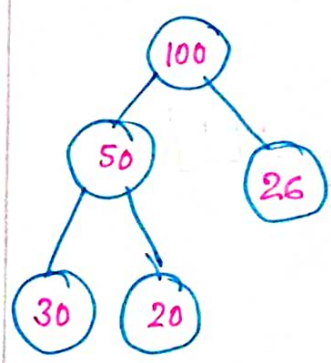
- Start from the root, swap this with the max valued child, if A[max_child] > A[i]



(Max heap); $O(\log n)$ swaps.

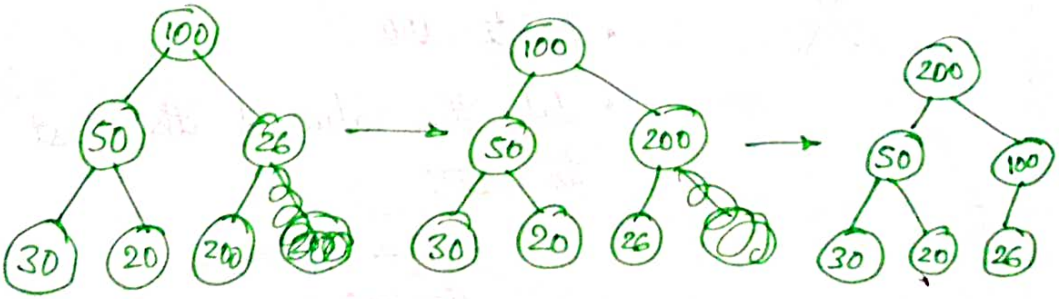
```
deleteMax (H, size) {
        H[1] = H[size];
        size--;
        nodeIndex = 1;
        while (true) {
            l = 2 * nodeIndex;
            r = 2 * nodeIndex + 1;
            max = r;
            if (l > size) return;
            if (r > size) max = l;
            else if (H[l] > H[r]) max = l;
            else max = r;
            if (H[nodeIndex] > H[max]) return;
            swap (H[nodeIndex], H[max]);
            nodeIndex = max;
        }
}
```

⊛ Insert



Insert 200

* Insert at last vacant leaf.
* compare with parent, swap if necessary.



```
insertHeap(H, size, newVal) {
        H[size+1] = newVal;
        nI = size+1;
        size++;
        while (true) {
            if (nI == 1) return;
            p = nI / 2;
            if (H[p] >= H[nI]) return;
            swap(H[p], H[nI]);
            nI = p;
        }
}
```

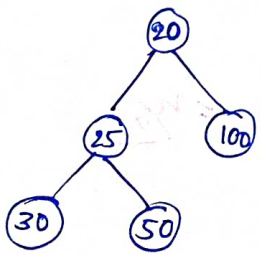
FindMax → $O(1)$
DeleteMax → $O(\log n)$
Insert → $O(\log n)$

Creating a Heap:—

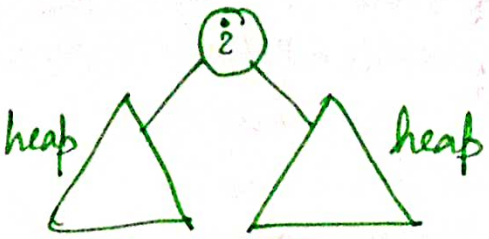
Scenario 1: We get all n elements one-by-one
Insert repeatedly → $\Theta(n \log n)$

Scenario 2: All elements are given (in arbitrary order)

X 20 25 30 50 100



* All leaf nodes are heaps
* Go from size/2 ... 1, at each ~~operation~~ index, ensure that the subtree rooted at an arbitrary vertex is a heap.



$H[i]$ may violate heap property. Do something to correct it (analogous to deleteMax)

```
for (i = size/2; i >= 1; --i){
        maxHeapify(H, size, i);
}
```

```
maxHeapify(H, size, i);
    nI = i;
    while (true){
        l = 2*nI;  r = 2*nI+1
        if (l > size) return;
        ⋮ } Analogous to deleteMax
}
```

Suppose $n = 2^{l+1}$

$l=0$	o	1	$\frac{n}{2^{l+1}}$
$l=1$	o o	2	
$l=2$	o o o o	2^2	
⋮	⋮		
$l=$	o o ⋯ o o	$2^l \approx n/2$	

Now, $2^l \le n \le 2^{l+1} - 1$

For each node in Level k, $0 \le k \le L-1$, we ~~sum~~ run MaxHeapify; each taking $O(l-k)$ time.

$\therefore$ Complexity of Buildheap $\le c \sum_{k=0}^{l-1} 2^k (l-k)$

Now, $\sum_{k=0}^{l-1} 2^k (l-k)$

$\le \sum_{k=0}^{l-1} \frac{n}{2^{l-k}} (l-k)$ $\quad [\because 2^l \le n]$

$= \sum_{h=1}^{l} \frac{nh}{2^h}$ $\quad [\text{where } h = l-k]$

$S \le \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots\cdots$

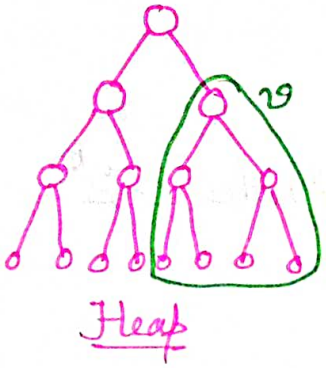
$\therefore \frac{S}{2} = -\frac{1}{2^2} - \frac{2}{2^3} - \cdots$

$\Rightarrow \frac{S}{2} = \frac{1}{2} + \frac{1}{2^2} + \cdots = 1 \Rightarrow S = 2$

$\therefore \sum_{h=1}^{l} \frac{nh}{2^h} \le n \sum_{h=1}^{\infty} \frac{h}{2^h} = 2n$

$\Rightarrow$ Complexity is $O(n)$

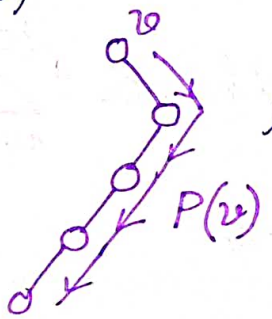
Visual Proof :-



Let v be a node & $T(v)$ be the subtree rooted at v.

Complexity of max heapify at v = Height$(T(v))$

Consider the inorder successor of v, let the path be $P(v)$



$\underset{O(\log n)}{\text{length}[P(v)]} = \text{Height}[T(v)]$

(except 2, where length $[P(v)] = \text{Height}[T(v)] - 1$)

$\therefore$ Complexity of Build Heap $= O\left(\sum_{\text{non-leaf}} \text{length}[P(v)]\right)$

Now, for any two nodes, u & v

suppose $P(u)$ & $P(v)$ do not share an edge. Then $O\left(\sum_{\text{nonleaf}} \text{length}(P[v])\right) = O(\#\text{edges}) = O(n)$

See that if $P(u)$ & $P(v)$ share an edge then successor(u) = successor(v) [Contradiction]

MakeHeap $- O(n)$

BuildHeap	$O(n)$
Insert	$O(n \log n)$
Findmax	$O(1)$
Delete Max	$O(\log n)$

Sorting Using Heaps :—

Given n integers —

1. Create Heap
2. Find Max
3. Delete Max
4. Go to 2 if heap is not yet empty

∴ "Heapsort" takes time $O(n \log n)$

This can be done "inplace", that is with $O(1)$ extra space.

We can similarly perform "AVL-Sort" with AVL Trees.

Data structure facilitating "Search"

- {Sequential Search} (comparison)
- {Binary Search} (comparison)
- {DFS / BFS} (graph)
- {Tree structures for searching items (AVL, BST)} (comparison)

All item searches done so far are comparison based. However we will now reference an item based on "key value".

HASHING

Hash Function: A hash function h, maps keys of a particular type to integers in a fixed interval $[0 \ldots, N-1]$

$h(x) \rightarrow$ hash value of the key x.

Example: $h(x, y) = (5x + 7y) \bmod N$

(Here the hash function depends on two integers)

~~Hash Fun~~

Hash Table: Given a hash function h, the array (table) of size N which stores the value (k, e) at index $h(k)$ of the array, is called the Hash Table.

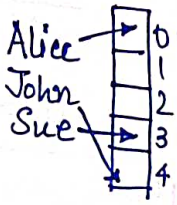
Example: $h(x) = x \bmod 5$.

<6, tea> → 1
<2, coffee> → 2
<14, chocolate> → 4

	key	element
0		
1	6	tea
2	2	coffee
3		
4	14	chocolate

Searching can be done in $O(1)$

Example 1: $h(w)$ = (length of word w) mod 5



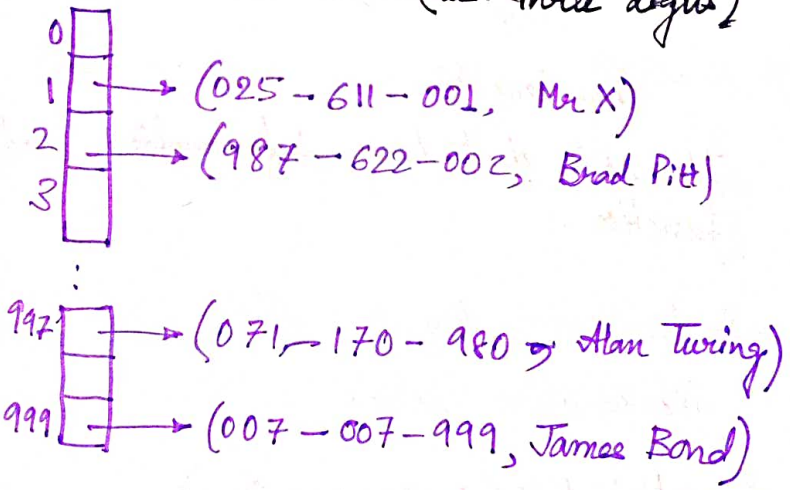
COLLISIONS :-

Joe "collides" with Sue

N → very large (still collisions are inevitable)

(Example 1 is a very bad choice)

Example 2: hash function → AADHAR number!!!!
(last three digits)



Choice of hash function
Collision handling ..

How Probable are Collisions?

Party → p persons

What is the probability that two people have the same birthday.

$N = 365$

$q(p, N)$ = prob. that no collisions

$$= \frac{N}{N} \cdot \frac{N-1}{N} \cdots \frac{N-p+1}{N}$$

$$= \frac{(N-1)(N-2)\cdots(N-p+1)}{N^p}$$

$q(p, N)$ → for $p = 23$, prob. of collision > 0.5

So our aim is to design "good" hash function, and a "good" collision handling technique.

Example: key → integer

$h(K) = (int)K \bmod N$

long long K.

We still don't have ~~information~~ good hash function.

We must try to utilize maximum information latent in K.

Example: $K = \langle x_0, x_1, \cdots, x_{k-1} \rangle$

$(int)\langle x_0, x_1, \cdots, x_{31} \rangle + (int)\langle x_{32}, \cdots, x_{n-1} \rangle$

Example: $S\phi$ = ~~"tmpo"~~ $S[0], S[1], \cdots, S[n]$

$S[0] + S[1] + \cdots + S[n] \pmod N$

Collisions: "stop", "tops", "spot", "pots"

Polynomial Hash Codes :-

$W_k = \langle x_0, x_1, \cdots, x_{k-1} \rangle$

↓

Consider each position — $X[i]$

$h(W_k) = x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-1}$

Experimentally we know that $a \in \{33, 37, 39, 41\}$ are "good" values of a, giving ~ 7 collisions in 50,000 randomly chosen English words (Hence this is used in Java and other languages).

Does 'N' have to do anything with all this?

N should be prime; that way it has a higher likelihood to "spread" the key values.

Example :-

key = {200, 205, 210, ..., 600}, N = 100 or 101

N=100:-

205, 305, 405, 505 ; 210 310 410 510

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

5 5 5 5 10 10 10 10

A single hashcode would collide with three others.

N=101:-

205, 305, 405, 505, 210, 310, 410, 510

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

3 2 1 0 8 7 6 5

$$h(k) = (s_0 a^{k-1} + s_1 a^{k-2} + \cdots + s_{k-2} a^1 + s_{k-1} a^0) \% N$$

Hourner's Rule:-

$$h(k) = \{(((((s_0 a + s_1) a + s_2) a + s_3) a + \cdots) a + s_{k-1}\} \% N$$

This way, it will be easier to have an iterative computation algorithm

```
int hash(char v[], int N){
    int h=0; int a=33;
    for(int i=0; i<v.length; ++i){
        h = (a*h + v[i]) % N;
    }return h%N;
}
```

COLLISION HANDLING

- Chaining
- Open Addressing
  - Linear Probing
  - Double Hashing.

Chaining:-

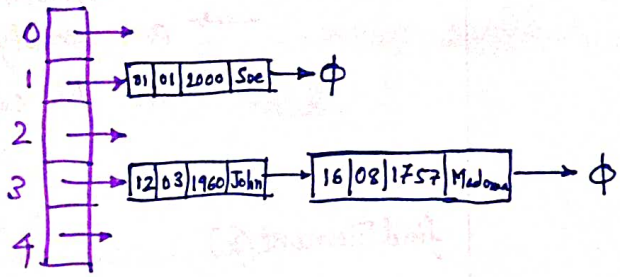
Each cell in the hash table is linked with a linked list, (a chain). Colliding items are stored in the list outside the table.

Example:

key = birthday, name

h(k) = (birthday + month) % 5

01|01|2000 , Sue
12|03|1960 , John
16|08|1757 , Madonna



Worst case: All entries became a part of the same chain.

Open Addressing:-

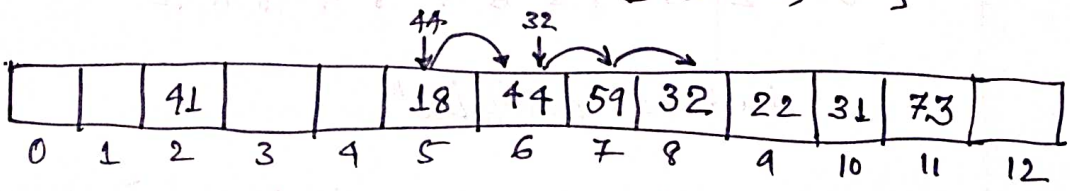
The colliding items are placed in a different cell of the table

(1) Linear Probing:-

* Colliding items are just put inside the "next" available cell.
* Test if cells are free to be inserted to (Probing)

Example: $h(x) = x \bmod 13$

{18, 41, 22, 44, 59, 32, 31, 73}



Colliding items will lump together and cause new collisions

* **Searching:**

→ find (k)

→ Start from the cell $h(k)$, keep probing.

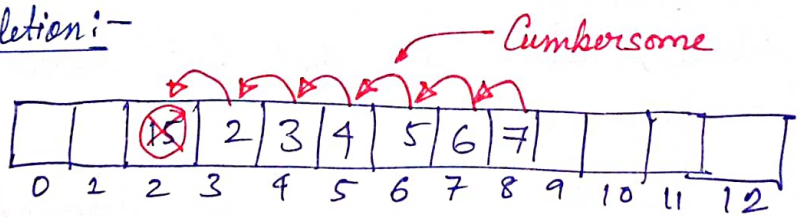
→ an ~~intern~~ item k is found.

→ an empty cell is found.

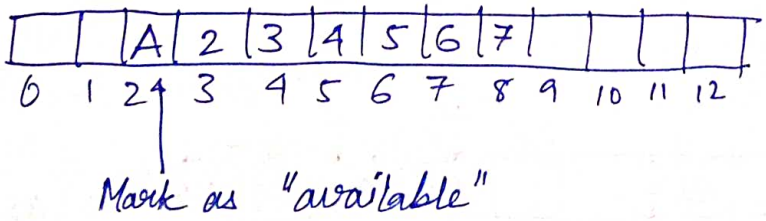
→ All N cells have been probed.

```
findElement(k)
    i = h(k)
    p = 0
    while p < N do
        c = A[i]
        if c == ϕ return No_search_key
        if c.key == k return c.element
        i = (i+1) % N
        p = p + 1;
    return No_search_key.
```

* **Deletion:-**

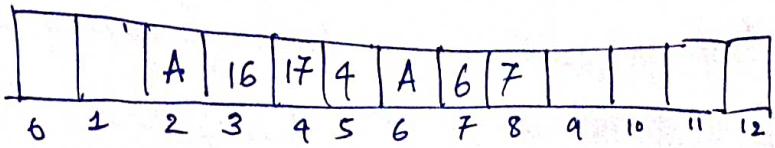


Delete 15



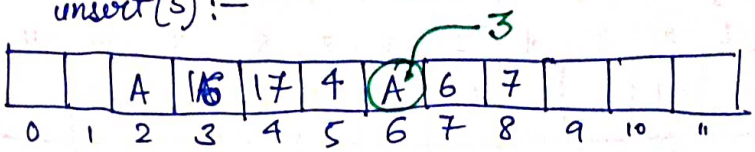
* Occasional rearrangement is required. to clean up "A"s (reorder)

* Insertion



Insertion: Start with $h(k)$, probe consecutively until an empty cell or an A-cell is found.

insert(3):-



* Insert
* Delete
* Find

} Complete Collision Handling.

Rather than probing the "next cell", probe the i-th cell,

* Linear increament
* Changing Direction
* ~~Doubling~~ Double hashing.

DOUBLE HASHING :-

Keep a secondary hash function $d(k)$. then insert at $(h(k) + jd(k)) \% N$ on j-th collision.

Example :-

$N = 13$, $h(k) = k \bmod 13$, $d(k) = (7 - k) \bmod 7$

0	1	2	3	4	5	6	7	8	9	10	11	12
31		41			18	32	59	73	22	44		

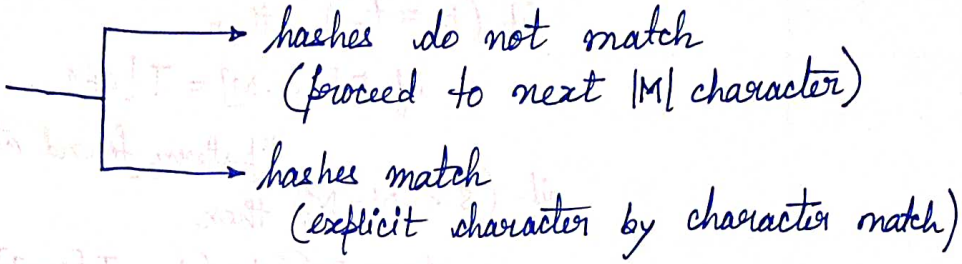
	k	h(k)	d(k)	Probe	
1	18	5	3	5	5+0x3
2	41	2	1	2	2+0x1
3	22	9	6	9	9+0x6
4	44	5	5	5, 10	5+0x5=5 5+1x5=5+5=10
5	59	7	4	7	7+0x4
6	32	6	3	6	6+0x3
7	31	5	4	5, 9, 0	5+0x4=5 5+1x4=9 5+2x4=13≅0
8	73	8	4	8	8+0x4

This empirically guarantees that the values are spread across "better" and collisions are minimized.

STRING MATCHING : RABIN-KARP ALGORITHM

Find the presence of a pattern M in a sequence of characters (the string)

* Compute the hash value of the pattern M.
* Compute the hash value of the next |M| characters of the sequence.
* Compare the hash.



M char sequence ⟶ M digit number in base b.
(where b is the alphabet size)

text subsequence : $t[i \ldots i+X-1]$

$$X(i) = t[i] * b^{M-1} + t[i+1] * b^{M-2} + \ldots + t[i+M-1]$$
$$X(i+1) = t[i+1] * b^{M-1} + t[i+2] * b^{M-2} + \ldots + t[i+M]$$

$$X(i+1) = X(i) * b + t[i+M] - b^M t[i]$$

[The arithmetic is done in modulo q, q is prime]

$$[x \bmod q + y \bmod q] \bmod q = (x+y) \bmod q$$
$$[x \bmod q] \bmod q = x \bmod q$$

$$h(i) = \left[\left(t[i] * b^{M-1} \bmod q\right) + \left(t[i+1] * b^{M-2} \bmod q\right) + \ldots + \left(t[M+i-1] \bmod q\right)\right] \bmod q$$

$$\therefore \quad h(i+1) = \left(h(i) * b \bmod q - t[i] * b^M \bmod q + t[i+M]\right) \bmod q$$

Pseudocode :=

n = length(T), ~~M~~ = length ~~(M)~~ (P)
$h = d^{(M-1)} \bmod q$
$p = 0$, $t_0 = 0$
for $i = 0$ to M, do :
$p = (d * p + P[i]) \bmod q$
$t_0 = (d * t_0 + T[i]) \bmod q$
for $s = 0$ to $N-M$, do -
if $(p = t_s)$ then
if $P[1 \ldots M] = T[s+1, \ldots, s+M]$ then
"pattern found at shift" s
if $(s < N-M)$ then
~~$t_{s+1} = ((d * (t_s - T[s+1])) h + T[s+M+1]) \bmod q$~~
$t_{s+1} = (d * (t_s - T[s+1] * h) + T[s+M+1]) \bmod q$

Example :=

T = "aabbcaba" ; P = "cab" ; $q = 3$; $d = 26$

hash (cab) = $(2 \times 26^2 + 0 \times 26^1 + 1 + 26^0) \bmod 3$
$= 0$
hash (aab) = 1
hash (abb) = 0 || collision (explicit matching required)
.
.
.

NAME: ANUBHAV DHAR ROLL NO: 20CS30004

SORTING

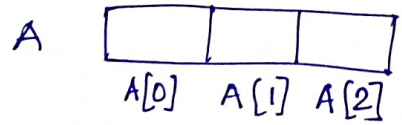
(*) Insertion $O(N^2)$ Worst Case
(*) Quick Sort $O(N^2)$ Worst Case
(*) Merge Sort $O(N \log N)$
(*) Heap Sort $O(N \log N)$
(*) AVL Sort $O(N \log N)$

Can we do better than $O(n \log n)$?
(If n elements are arbitrary)

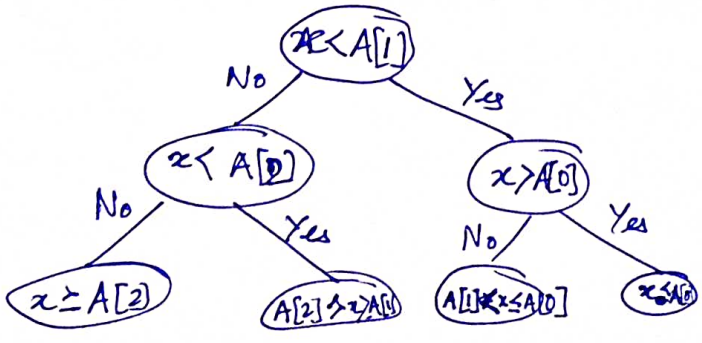
* We depend on comparison operation for sorting (This we call comparison based sorting)
* Let us assume, we only charge for comparisons.
* No. of comparisons can help establish a lower bound.
* We want to show that this lower bound of $\Omega(n \log n)$

To Prove :-

Searching an element in an array A[ ] of n elements (sorted) using binary search ($O(n \log n)$) We need to show that we cannot do better than $\Omega(\log n)$



Binary Search (x)



NAME: ANUBHAV DHAR ROLL NO.: 20CS30004

The decision tree of n nodes will have n+1 answers

Decision Tree	Algorithm
* Internal node	* One comparison operation
* leaf node	* Answer found
* root-to-leaf path	* algorithm execution for a particular input
* Length of path	* Running time
* Height of tree	* Worst case running time.

In any algorithm, the number of possible outputs will not change (either all possible n+1 outputs)

$\Rightarrow$ No. of leaf nodes are n+1.

The height of a binary tree with ~~height~~ n+1 leaf nodes is $\Omega(\log n)$.

$\therefore$ The worst case runtime is $\Omega(\log n)$

* For sorting, the internal nodes will be

$$A[i] < A[j]$$

* For sorting, the leaf nodes will be some permutation (e.g. $A[5] < A[3] \leq \cdots < A[8]$)

∴ Number of leaf nodes → $n!$

∴ Worst case running time is

$$\Omega(\log(n!)) = \Omega(n \log n)$$

To do better than $\Omega(n \log n)$, we need to use more information.

Integer Sorting:-

$A[1] \ldots\ldots A[n]$ unsorted.

→ range of values.
min max

n integers in range $[0, 1000]$

We create $cnt[_{0} \quad]_{1000}$

[e.g. $A[i] = 5$; $c[5] += 1$]

Counting Sort:

```
for i = 1 → n
    C[A[i]]++;
for i = 0 → 1000
    print i, C[i] times.
```

Runtime : $O(n)$

Restriction:
* All integers
* Range should be known & have size $O(n)$.

NAME: ANUBHAV DHAR ROLL NO.: 20CS30004

"But generally when we need to sort items, each item may have multiple fields"

Name	Marks
Bishal	53
Rohan	34
Mridul	45
Mohan	34

* Pick up another field and break ties.
* For ties place them ~~one which~~ in the order of input.

"Any sorting algorithm that does this (preserves order) is a stable sorting algorithm."

* Stable Sort :-

A sorting that ensures that for keys with the same value, the order of the keys in the output array will be the same as in the input array is called a stable sorting algorithm.

Quick Sort cannot be implemented to be a stable sort.

Counting Sort ⟶ Stable Sort

Input: Array of size n with multiple fields having the range of keys in $\{0, 1, 2, \ldots, K-1\}$ where $K = O(n)$

The input:

1A	0M
2B	0P
0B	2R
1B	2S
0C	2S
0D	1S
0K	0S

* One solution is to use array of linked list.
* Reusing our previous algorithm

$C[i] \triangleq$ Number of times we encounter i in $A[\,]$

C	7	3	3

Tle store the output in array $B[\,]$.

B [ ]

* Make pass through A $\forall\ j \in \{0, 1, \ldots, k-1\}$
⟶ $O(nk)$, Highly inefficient.

* B [x x x x x x x | x x x | x x x]
0 1 2

So we allocate positions for each value.

Now we redefine $C[i] \triangleq$ index in B where i should be printed.

C[] then

7	3	3

⟶ C[] now

0	7	10

Step 1: Right shift $O(k)$

```
for(i=K-1; i>0; --i)   C[i] = C[i-1];
```

Step 2: Cumulative Sum $O(k)$

```
for(i=1; i<K; ++i)   C[i] += C[i-1];
```

Now, when we traverse A, we put this element in B[C[A[j].key]]; & we increment C[A[j].key]

Working Example:-

1A	2B	0B	1B	0C	0D	0K
0M	~~0M~~	0P	2R	2S	1S	0S

B

0B	0C	0D	0K	0M	0P	0S	1A	1B	1S	2B	2R	2S	X
0	1	2	3	4	5	6	7	8	9	10	11	12	13

C[]:

0	1	2
0	7	10
0	8	10
0	8	11
1	8	11
1	9	11
2	9	11
3	9	11
4	9	11
5	9	11
6	9	11
6	9	12
6	9	13
6	10	13
7	10	13

Now what if we have $K = n^2$.

Each integer in ~~Of~~ a as —

$$a = a_0 n + a_1.$$

where $a_0 = \left\lfloor \frac{a}{n} \right\rfloor$; $a_1 = a \% n$

Now $a_0 \in \{0, 1, \ldots, n-1\}$

$a_1 \in \{0, 1, \ldots, n-1\}$

Sort according a_0 (Counting Sort)
Sort according a_1 (Counting Sort)
} (Order matters)

Example:—

$n = 10$

Input	a_0 - Sort	a_1 Sort
52	30	07
33	41	17
07	52	27
41	62	30
62	33	33
27	75	41
75	07	47
17	27	52
47	17	62
30	47	75

Complexity : $O(n)$ [Running counting sort twice]

Generalizing this idea:

$K = O(n^d)$ d is a constant.

then we can sort in $O(dn)$. In general if d is small enough (not comparable to $\log n$), the it is $O(n)$. It is called "Radix Sort"

BUCKET – SORT

* For an input of size n,
* We divide the input into n buckets, and each bucket is expected to have ~~$O(n)$~~ $O(1)$ inputs.
* Sort each bucket individually,
* Concatinate each bucket to get final answer.

Example: For inputs in $[0, \ldots, 1000]$, we break buckets as —

$[0, 100]$, $[101, 200]$, $\ldots$, $[901, 1000]$

Expected complexity : $O(n)$

Worst case complexity : Will depend on how we sort the individual ~~algorithms~~ buckets.

Characteristics of Sorting Algorithm

Time Complexity	Stable Sort	In-place
	QuickSort ✗	QuickSort ✓
		HeapSort ✓
		MergeSort ✓
		Counting Sort ✗
		~~Bc~~ RadixSort ✗

* Given an array of n elements, what is the lowerebound on constructing a binary search tree from here.

→ $\Omega(n)$

$\Omega(n \log n)$ (Best Lower bound)

AVL Tree → $O(n \log n)$

Assume you have a better algorithm with complexity T, [T is better than $O(n \log n)$]

Then we can have a sorting algorithm in $T + O(n)$ (Run BST algorithm + Inorder traversal) So Impossible!

ORDER STATISTIC

The i-th order statistics of a set of elements is the i-th smallest number

3, 8, 5, 2, 11, 7 ⟶ 3rd order statistics is 5

3, 3, 8, 5, 2, 11, 7 ⟶ 3rd order statistics is 3

The i-th order statistics is the i-th position after sorting.

Very Often we use ⟶

1. Minimum ($i = 1$)
2. Maximum ($i = n$)
3. Medium ($i = \frac{n}{2}$ if n is even; average of $i = \frac{n-1}{2}$ & $i = \frac{n+1}{2}$, n is odd)

Does this Depend on i?

⟶ We atleast have an $O(n \log n)$ [Sort and search]

For $i = 1$: Finding minimum, $n-1$ comparisons.

```
min = A[1]
for j = 2 → n
    if A[j] < min
        min = A[j]
```

Why cannot we do better than $n-1$ comparisons.

"We want to make a cricket pair up, n teams such that one team loses."

We cannot do better than $n-1$, since otherwise there will be two teams which haven't lost (for t games, there will be t losses)

Finding min and max ~~in~~ together:-

* Simple algo. (Find min then find max) in $2n-2$ comparisons
* Break into chunks of 2, $\{A[0], A[1]\}, \{A[2], A[3]\} \cdots$
  1. Compare $A[i]$ and $A[i+1]$ (i even)
  2. Compare max and $A\left[\underset{k \in \{i, i+1\}}{\text{argmax}} \{A[k]\}\right]$
  3. Compare min and $A\left[\underset{k \in \{i, i+1\}}{\text{argmin}} \{A[k]\}\right]$

$\therefore$ The number of comparisons $\approx \frac{3}{2} n$

Finding 2nd order statistics

* 1. Find first order statistics
  ~~2. Find second order statistics~~
  2. Remove it
  3. Find second order statistics.

$\therefore$ The number of comparisons $\approx 2n$

Complexity $O(2n)$

$\therefore$ For k-th order statistics: $O(kn)$ [Inefficient]

$\therefore$ It will be $O(n^2)$ for median.

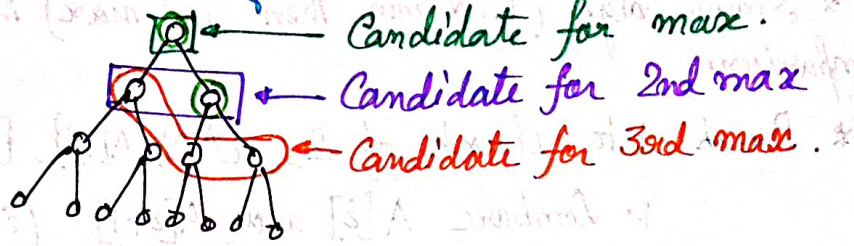
$\therefore$ For a general k, we can sort it in $O(n \log n)$ and proceed.

* Using Heaps:-

heap $\longrightarrow O(n + k \log k)$

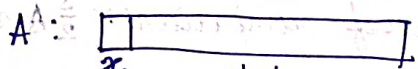
[Keep heap of size k]

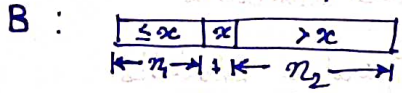
The Heap Algorithm :-



Worst Case O(n) algorithm :-

QuickSort : Recap —





Extending this Idea :—

if $k = n_1 + 1$ → pivot is answer.

if $k < n_1 + 1$ → find (recursively) k-th element in left partition

if $k > n_1 + 1$ → find (recursively) k-th element in right partition.

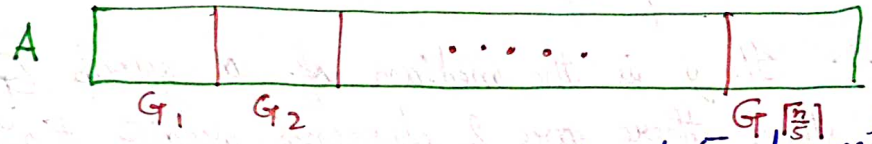
If we somehow ensure that the partitions are (roughly balanced).

Blum Floyd Pratt Rivest Tarjan

* Same algorithm as we discussed.
* Interesting idea is to select the pivot.
* Finally: ① Each partition has elements $\geq \frac{3n}{10}$, $T\left(\frac{3n}{10}\right)$
  ② Recursive call to the selection algorithm $T\left(\frac{n}{5}\right)$

$$T(n) = T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n) \longrightarrow T(n) = O(n)$$

"These 5 people have done something with 5"

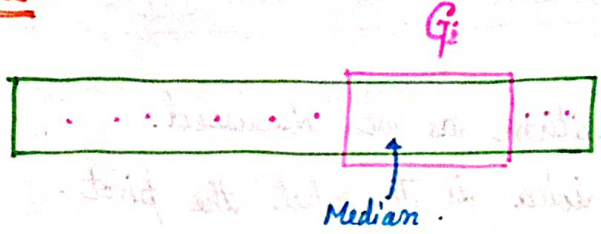


1. Divide the array into $\frac{n}{5}$ groups of 5 elements each.
2. Find median of these $\frac{n}{5}$ groups. Even $O(k^2)$ algorithm would work in $O(5n)$.
3. Recursively call find median to the list of these obtained median, which will be used as a pivot.
4. Partition the array and recursive.

This Guarantees that the left and right partition have atleast $\frac{3n}{10}$ elements

"This algorithm is also called Median of Median Algorithm"

Analysis:



Let median of medians be x.

1. x is a median of $\frac{n}{5}$ medians.
2. there are about $\frac{n}{5} \times \frac{1}{2} = \frac{n}{10}$ elements which are greater & $\frac{n}{10}$ elements which are less than x, as well as they are medians of some group.
3. If y is the median of a group G_j, then there are 2 elements greater than y and two elements less than y.
4. Therefore each partition will have atleast $\frac{3n}{10}$ elements.

Complexity of Selection.

* Finding the median of all groups : $O(n)$
* Finding median of medians : $T\left(\frac{n}{5}\right)$
* ~~Partitioning~~ Partitioning : $O(n)$
* Recursion : $T\left(\frac{7n}{10}\right)$

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

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

$\therefore$ We can now have a guaranteed $O(n \log n)$ algorithm for quicksort.

What is special about the number 5.

* Let us take 3,

Number of elements less than the median of medians $\rightarrow 2 \cdot \frac{n}{6} = \frac{n}{3}$. $\Rightarrow T\left(\frac{2n}{3}\right)$ for the larger partition.

$$\therefore T(n) = \underbrace{O(n)}_{\text{median finding in each chunk}} + \overbrace{T\left(\frac{2n}{3}\right)}^{\text{recursion}} + \underbrace{T\left(\frac{n}{3}\right)}_{\text{median of median finding}} \Rightarrow T(n) = O(n \log n)$$

☹

* Let us take 7;

Number of elements less than the median of median $\rightarrow \frac{n}{14} \times 4 = \frac{2n}{7}$. $\rightarrow T\left(\frac{5n}{7}\right)$ for larger partition.

$$\therefore T(n) = O(n) + T\left(\frac{5n}{7}\right) + T\left(\frac{n}{7}\right) \Rightarrow T(n) = O(n)$$

☺

Further $\frac{n}{7}$ elements are thrown.

* For 9, we throw $\frac{n}{6}$ elements!

However, for groups of size, k, we find the median in $O(k^2)$ for small k.

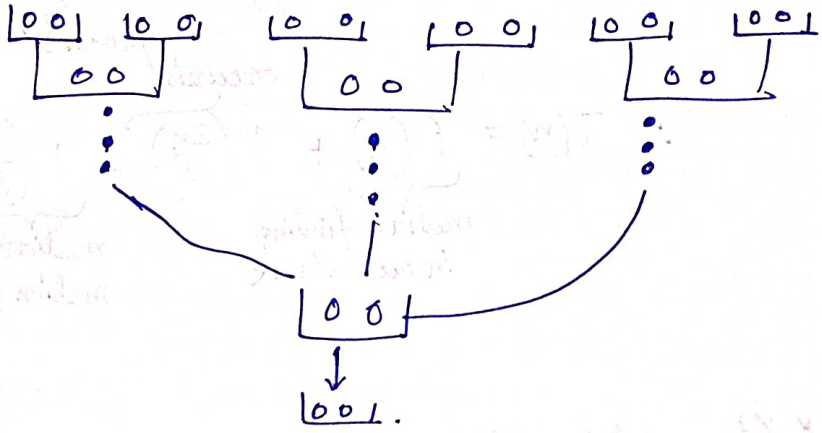
$\therefore$ For all $\frac{n}{k}$ chunks, we have $O\left(k^2 \cdot \frac{n}{k}\right)$

$= O(kn)$.

So this increase has more impact compared to the decrease.

Second Order Statistics

If we find max of ~~elem~~ the set and then max of the remaining elements. Consider the following algorithms.



Winner played $\log_2 n$ matches.

The Runner up, must have lost ONLY to the winner, and hence must be a part of these $\log_2 n$ matches.

$\therefore$ We need $\underbrace{(n-1)+(\lceil \log_2 n \rceil - 1)}$

minimum number of comparisons to find 2nd best