Analysis of algorithms

Asymptotic analysis

In this analysis, we evaluate performance of algorithm in terms of input size (don't measure actual running time)

Asymptotic analysis is not perfect, but it is best way available for analyzing algorithm. for example, two sorting algorithm take $1000n\log n$ and $2n\log n$ time.

Both of these algorithm are asymptotically same (because both have same order of growth $n\log n$), by asymptotic analysis we can't judge which one is better as we ignore constant in asymptotic analysis.

Worst, Average and best cases

analysis on linear search :-

i) worst cases :- Calculate upper bound on running time.

for linear search, worst case happen when element searched is not present in array.

Time complexity $O(n)$

ii) Average Case analysis :- It is sum all time taken divide no. of inputs.

$$\text{Average Case time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{n+1} = \frac{\theta\left(\frac{(n+1)(n+2)}{2}\right)}{(n+1)}$$

$$= \theta(n)$$

iii) Best Case:- we calculate lower bound on running time of algorithm.

time Complexity of best case in linear serach:- O(1)
Element is present at first position.

Asymptotic Notations

Asymptotic analysis doesn't depend on machine specific Constant it always depend only on order of growth.

1. Θ (theta) Notation:- it bounds a function from above and below. So it define exact asymptotic behavior.

Θ (theta) Notation get from expression by drop low order term and ignore leading constant.

$3n^3 + 6n^2 + 6000 = \Theta(n^3)$

2. Big O notation:- it defines upper bound of algorithm. it bounds a function from above only.

Insertion Sort, take linear time in best case and quadratic time in worst case. we can safely say time complexity of insertion sort $O(n^2)$.

Note, $O(n^2)$ also cover linear time.

if we use O notation to represent time Complexity of insertion sort then we have to use two statements for best and worst case:-

i> worst Case time Compexity $O(n^2)$

ii> best " " $\Omega(n)$

3. Ω Notation :- Ω Notation provide lower bound of function.

Time Compexity of insertion Sort is $\Omega(n)$ because it is best Case (lower bound)

Properties of Asymptotic Notation :-

i> General properties :- if $f(n)$ is $O(g(n))$ then $a*f(n)$ is also $O(g(n))$ where a is Constant.

Example :- $f(n) = 2n^2+5$ is $O(n^2)$

$7*f(n) = 14n^2+35$ is $O(n^2)$

Similary, property also satisfy for theta (Θ) and Ω Notation.

ii> ~~Reflexive Properties :-~~

little o asymptotic Notations (strictly upper bound)

Analysis of loops

1. O(1) :- Time complexity of a function is O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

Example:- Swap() function has O(1) complexity.

A loop or recursion that runs a constant no. of time is considered as O(1).

```
for (int i=1; i<=c; i++)   // Here c is constant
{
  // some O(1) expression
}
```

Time Complexity O(1).

2. O(n) :- if loop variable is incremented / decremented by a constant amount.

Ex:-
```
for (int i=1; i<=n; i+=c)
{
  // some O(1) expression
}
```

3. $O(n^c)$:- Time complexity of nested loop is equal to number of times innermost statement executed.

Ex:-
```
for (int i=1; i<=n; i+=c)
{ for (int j=1; j<=n; j+=c)
{ // Some O(1) expression
}
```

Time Complexity $O(n^2)$

4). O(log n) :- Time Complexity is O(log n) if loop variable is multiply / divided by a Constant amount.

```
for (int i=1 ; i<=n ; i*=c)
  { // some O(1) expression
  }
for (int i=n ; i>0 ; i/=c)
  { // Some O(1) expression
  }
```

Series in first loop is $1, c, c^2, c^3 \ldots c^k$

$c^k = n \Rightarrow k = \frac{\log n}{\log c} = O(\log n)$.

5. O(log log n) :- if loop variables is reduced / increased exponentially by a Constant amount.

```
// Here c is constant greater than 1
for (int i=2 ; i<=n ; i=pow(i,c))
  { // Some O(1) expression
  }
// Here fun is Sqrt or Cuberoot or any other Constant root.
for (int i=n ; i>1 ; i=fun(i))
  { // Some O(1) expression
  }
```

$2, 2^c, 2^{c^2}, 2^{c^3}, 2^{c^4} \ldots 2^{c^k}$ ~~= n~~

$2^{c^k} = n$

$c^k = \log_2 n$

$k = \log_c \log_2 n \quad O(\log_c \log_2 n) = O(\log \log n)$

Solving Recurrences

i) ~~Subs~~ Three methods for solving recurrences :-

i) Substitution method :- We make a guess for soln and then we use mathematical induction to prove guess is correct or incorrect.

Example :- $T(n) = 2T(n/2) + n$

We guess $T(n) = O(n \log n)$, Now use induction to prove

We need to prove $T(n) <= c\, n \log n$

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

$<= 2\left[\frac{cn}{2} \log n/2\right] + n = n(c \log n - c \log 2 + 1)$

$<= nc \log n$

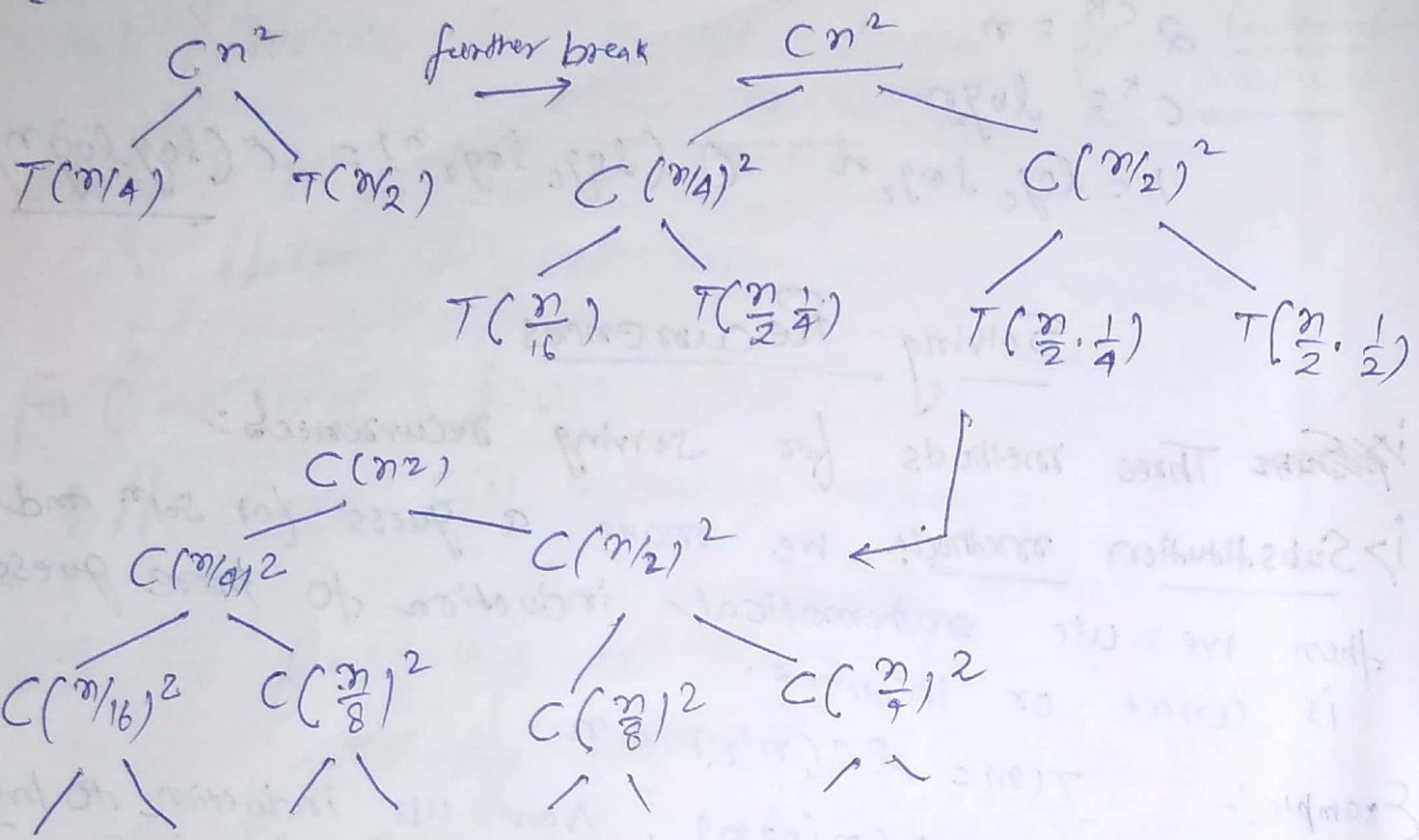
Hence, $T <= nc \log n$

$T = O(n \log n)$

ii) Recurrence Tree method :- In this method, we draw a recurrence tree and calculate time by every level of tree. finally we sum work done at all levels.

To draw recurrence tree, we start from given recurrence and keep drawing till we find a pattern among levels. Pattern is typically a arithmetic or geometric series.

Example:- $T(n) = T(n/4) + T(n/2) + cn^2$



$$T(n) = cn^2 + \frac{5}{16}n^2c + \frac{25}{256}cn^2 + \cdots$$

↓ first level ; ↓ second level ; ↓ third level

for get upper bound, we can sum upto infinite series.

$$T(n) = \frac{cn^2}{1 - 5/16} = \frac{16}{11}cn^2 = O(n^2)$$

Searching — Algorithms

1) Linear Search :-

```
serach (int arr[], int n, int x)
{ for (int i=0; i<n; i++)
    { if (arr[i]==x)
        return i;
    }
  return -1;
}
```

Time Complexity :- O(n)

Binary Search

Search a sorted array by repeatedly dividing search interval in half.

Time Complexity :- O(log n)

Algorithm :- i) Compare x with middle element.
ii) if x matches with middle element, return index.
iii) Else if x > mid element, then recur for right half.
iv) Else (x < mid element) recur for left half.

Recursive implementation of binary search

```
binarySearch (arr, 0, n-1, x);
int binarysearch (int arr[], int l, int r, int x)
{ if (r>=l)
  { int mid = (r+l)/2;
    if (arr[mid] == x)
        return mid;
    Else if (arr[mid] > x)
        return binarysearch (arr, l, mid-1, x);
    return binarySearch (arr, mid+1, r, x);
}
```

```
    return -1;
}
```

Iterative implementation of binary search

```
binarysearch(arr, 0, n-1, x);
int binarysearch(int arr[], int l, int r, int x)
{ while (l<=r)
  { int mid = (l+r)/2;
    if (arr[mid]==x)
        return mid;
    else if (arr[mid]<x)
        l = mid+1;
    else
        r = mid-1;
  }
return -1;
}
```

Time Complexity of binary search written as

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

$c \rightarrow$ constant

$T(n/2)$

$\therefore$ Time Complexity :- $O(\log n)$.

Jump Search

In this method check fewer elements in sorted array by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

What is optimal block size to be skipped ?

In worst case, we have to do n/m jumps and if last checked value is greater than the element to be searched for we perform $m-1$ comparision for linear search.

Total number of comparision in worst case be $((n/m) + m - 1)$. this value be minimum if $m = \sqrt{n}$

therefore, best step size is $m = \sqrt{n}$.

```
#include <bits/stdc++.h>
Using namespace std;
int jumpsearch ( int arr[], int x, int n)
{   int step = sqrt(n);
    int prev = 0;
   for(int i=0; i<n; i+= step)
     { if ( arr[i] > x)
          { for (int j= i-step+1; j< i ; j++)
             { if (arr[j] == x)
                 return j;
             }
           return -1;
          }
      else if (arr[i] == x)
         { return i;
         }
   }
   return -1;
}
```

Time Complexity :- $O(\sqrt{n})$

Auxiliary space :- $O(1)$

Note :- time Complexity of jump search is between Linear Search ($O(n)$) and binary search ($O(\log n)$)

Binary Search in C++ STL

binary_search (start_Ptr, end_Ptr, num) :-

function returns boolean true if element is present in container; else return false.

```
#include <bits/stdc++.h>
Using namespace std;
int main()
{  vector<int> arr = {10, 15, 20, 25, 30, 35};
   (binary_search (arr.begin(), arr.end(), 15)) ? exist : not exist;
}
```

Search an element in Sorted and rotated array

Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
Key = 3
Output : 8 (index number)

Algorithm :- i) Find middle point mid = (l+h)/2
ii) if key is present at middle point return mid.
iii) Else if arr[l ... mid] is sorted
   a) if key to be searched lies in range from arr[l] to arr[mid], recur for arr[l ... (mid-1)].
   b) Else recur for arr[(mid+1) ... h]
iv) Else (arr[(mid+1), ... h] must be sorted)
   a) If key to searched lies in range from arr[mid+1] to arr[h], recur for arr[mid+1, ... h].

b) Else recur for arr[l -- mid].

```
int search (int arr[], int l, int h, int key)
{   if (l>h) return -1;
    int mid = (l+h)/2;
    if (arr[mid] == key) return mid;
    if ( arr[l] <= arr[mid])
    {  if ( key >= arr[l] && key <= arr[mid-1] )
            return search ( arr, l, mid-1, key);
        return search (arr, mid+1, h, key);
    }
    if ( key >= arr[mid+1] && key <= arr[h])
        return search (arr, mid+1, h, key);
    return search (arr, l, mid-1, key);
}
```

Median of two Sorted arays of Same Size

There are two arrays A and B of size n each. Write algorithm to find median of array obtained after merging above 2 arays. Time Complexity should be O(logn).

input arr1[] = {1, 12, 15, 26, 38}
arr2[] = {2, 13, 17, 30, 45}

Output :- 16

Explanation :- array after merging :-

arr[] = {1, 2, 12, 13, 15, 17, 26, 30, 38, 45}

median = (15+17)/2 = 16.

Sorting Algorithms

1) Selection Sort :- Sort an array by repeatedly finding minimum element from unsorted part and putting it at begining.

arr[] = 64 25 12 22 11

// Find minm element in arr[0---4]
// Place it at begining
11 25 12 22 64
// Find minm element in arr[1---4]
// place it at begining of arr[1---4]
11 12 25 22 64
// find minm element in arr[2--4]
// place it at begining of arr[2--4]
11 12 22 25 64
// find minm element in arr[3-4]
// place it at begining of arr[3--4]
11 12 22 25 64

```
#include <bits/stdc++.h>
using namespace std;
 void swap (int *xp, int *yp)
 { int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void selectionsort(int arr[], int n)
 { int i, j, min_idx;
   for (int i=0; i<n-1; i++)
    {
```

```
        min_idx = i;
        for (j = i+1; j < n; j++)
            { if (arr[j] < arr[min_idx])
                min_idx = j;
            }
        Swap(&arr[min_idx], &arr[i]);
    }
}
void Printarray (int arr[], int size)
{ int i;
    for (i=0; i<size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
int main ()
{ int arr[] = {64, 25, 12, 22, 11};
    Selection sort (arr, n);
    Printarray (arr, n);
    return 0;
}
```

Time Complexity :- $O(n^2)$
Auxiliary space :- $O(1)$

Output :- 11, 12, 22, 25, 64

Bubble Sort

It works by repeatedly swapping the adjacent elements if they are in wrong order.

Example : 5 1 4 2 8

First Pass :-
5, 1, 4, 2, 8 → 1, 5, 4, 2, 8
1, 5, 4, 2, 8 → 1, 4, 5, 2, 8
1, 4, 5, 2, 8 → 1, 4, 2, 5, 8
1, 4, 2, 5, 8 → 1, 4, 2, 5, 8

Second Pass :-
1, 4, 2, 5, 8 → 1, 4, 2, 5, 8
1, 4, 2, 5, 8 → 1, 2, 4, 5, 8

1, 2, 4, 5, 8 ⟶ 1, 2, 4, 5, 8

1, 2, 4, 5, 8 ⟶ 1, 2, 4, 5, 8

Now, array is sorted, but our algorithm does not know if it is completed. Algorithm needs one whole pass without any swap to know it is sorted.

Third pass:-

No change.

```
void bubblesort (int arr[], int n)
{ int i, j;
  bool swapped;
  for (int i=0; i<n-1; i++)
  { swapped = false;
    for (j=0; j<n-1-i; j++)
    { if (arr[j] > arr[j+1])
      { swap (&arr[j], &arr[j+1]);
        swapped = true;
      }
    }
    if (swapped == false)
      break;
  }
}
```

Time Complexity: O(n*n)

Auxiliary space: O(1)

Insertion Sort

It works way we sort playing cards in our hands.

Algorithm

// Sort arr[] of size n

insertionsort(arr, n)

Loop from i=1 to n-1.

a) Pick element arr[i] and insert it into sorted sequence arr[0..i-1]

Example :- 4, 3, 2, 10, 12, 1, 5, 6 → 3, 4, 2, 10, 12, 1, 5, 6

2, 3, 4, 10, 12, 1, 5, 6 → 2, 3, 4, 10, 12, 1, 5, 6

2, 3, 4, 10, 12, 1, 5, 6 → 1, 2, 3, 4, 10, 12, 5, 6

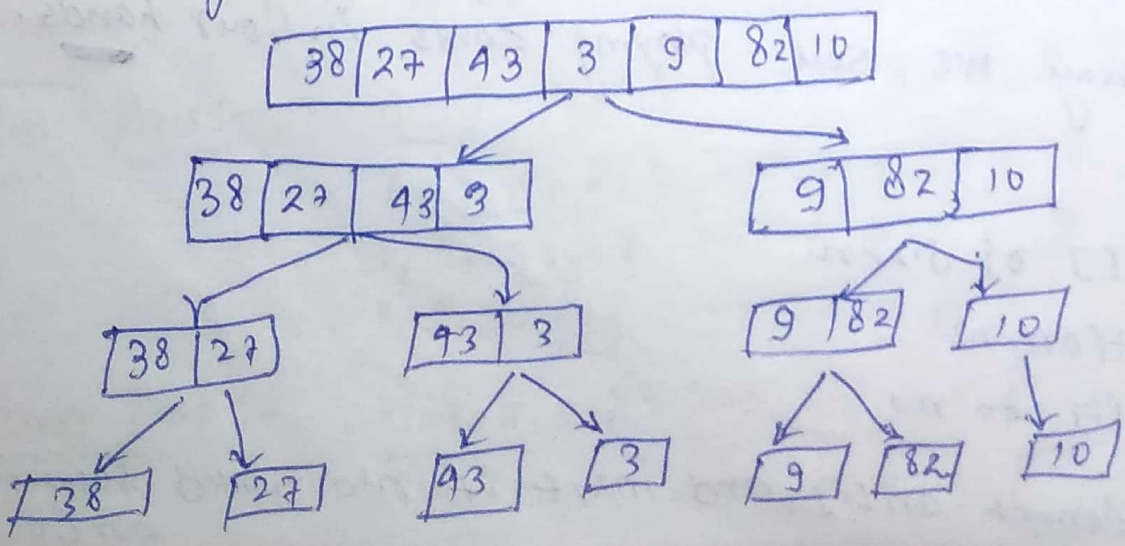
1, 2, 3, 4, 5, 10, 12, 6 → 1, 2, 3, 4, 5, 6, 10, 12

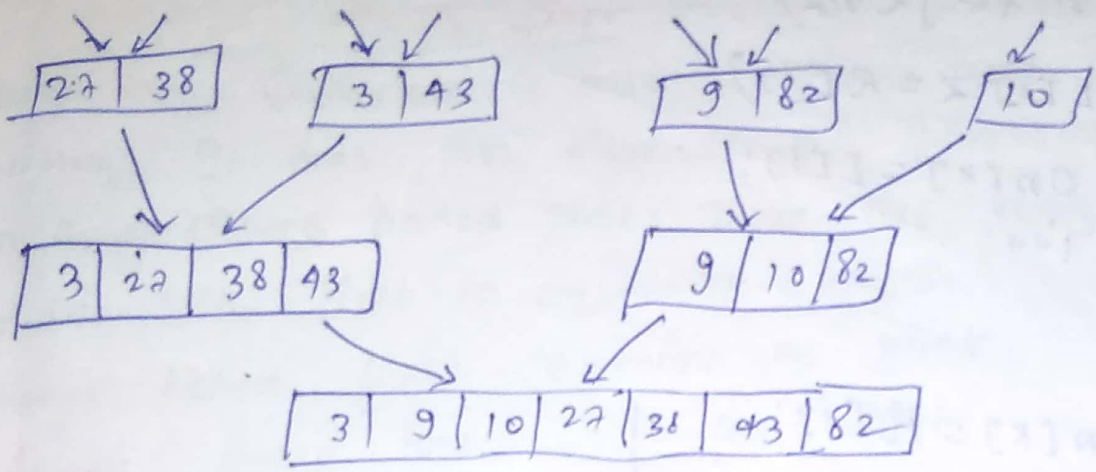
```
void insertionSort (int arr[], int n)
{  int i, key, j;
   for(i=1; i<n; i++)
    { key = arr[i];
      j = i-1;
      while ( j>=0 && arr[j] > key)
      { arr[j+1] = arr[j];
        j = j-1;
      }
      arr[j+1] = key;
    }
}
```

Time complexity :- O(n²)
Auxiliary space :- O(1)

Merge Sort

It is Divide and conquer algorithm. It divides input array in two half, call itself for two halves and then merge two sorted halves.





Algorithm:-

mergeSort(arr[], l, r)

if r > l

1> find middle element to divide array into two halves.

mid = (l+r)/2

2. call mergesort(arr[], l, mid) first half

3. call mergesort for second half

mergesort(arr[], mid+1, r)

4. merge two halves sorted in step2 and step3.

Call merge(arr, l, m, r)

merge() function merge two array arr[l...m] & arr[m+1,...r].

```
Void merge (int arr[], int l, int m, int r)
{ int i, j, k;
  int n1 = m-l+1;
  int n2 = r-m;
  int L[n1], R[n2];
  for (int i=0; i<n1; i++)
      L[i] = arr[l+i];
  for (int j=0; j<n2; j++)
      R[j] = arr[m+1+j];
  i=0;  j=0;  k=l;
```

```
while (i<n1 && j<n2)
{ if (L[i] <= R[j])
  { arr[k] = L[i];
    i++;
  }
  else
  { arr[k] = R[j];
    j++;
  }
  k++;
}
while (i<n1)
{ arr[k] = L[i];
  i++;
  k++;
}
while (j<n2)
{ arr[k] = R[j];
  j++; k++;
}
}
```

```
void mergesort (int arr[], int l, int r)
{ if (l < r)
  { int m=(l+r)/2
    mergesort (arr, l, m);
    mergesort (arr, m+1, r);
    merge (arr, l, m, r);
  }
}
```

Time Complexity :- $O(n \log n)$

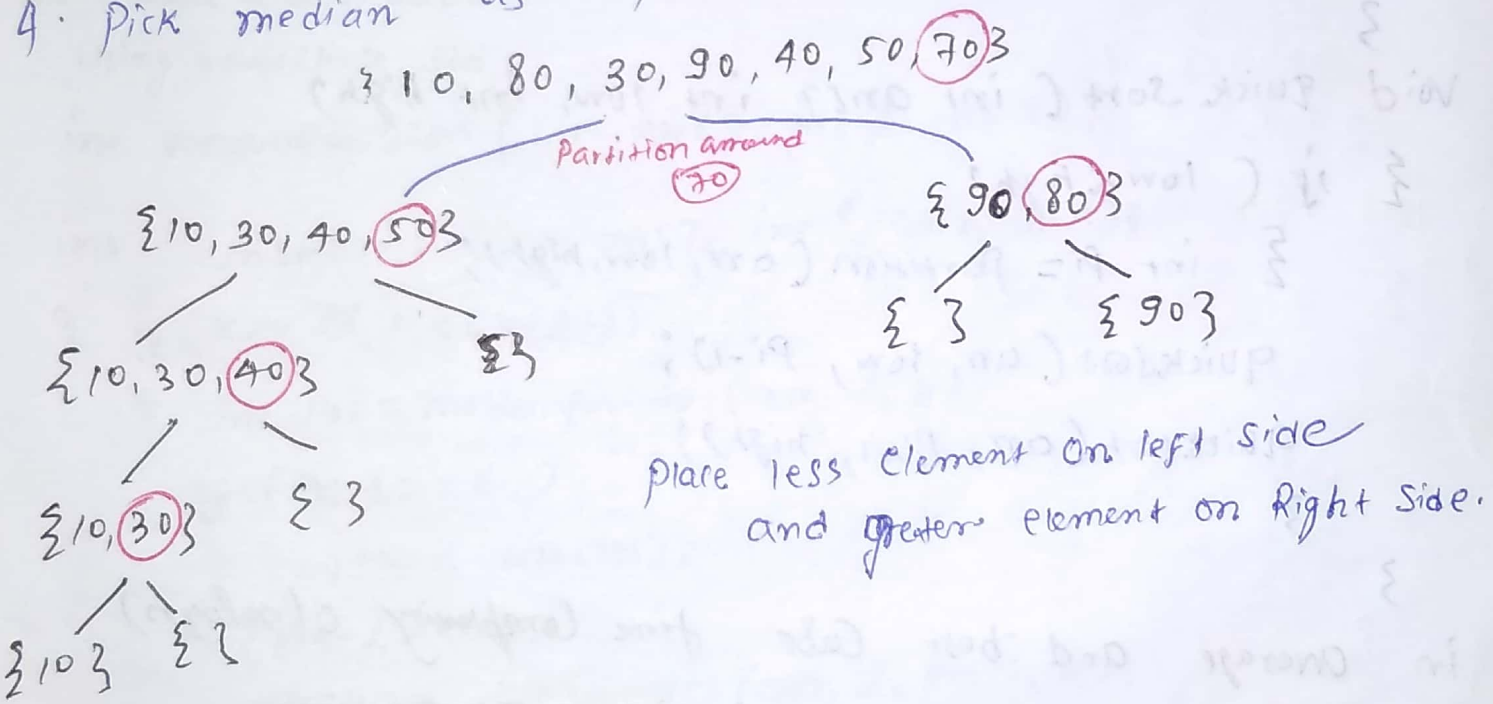
n :- for linear merge $\log n$ – two divide in half.

Space Complexity $O(n)$

Quick Sort

Like merge sort, Quick sort is also a divide and conquer algorithm. It picks an element as pivot and partitions given array around picked pivot. There are many different way to pick pivot in different ways.

1. Always picked first element as pivot
2. Always picked last " as pivot
3. pick a random element as pivot.
4. pick median as pivot.



Place less element on left side and greater element on Right side.

```
void Swap (int *a, int *b)
{ int temp = *a;
   *a = *b;
   *b = temp;
}
int Partition( int arr[], int low, int high)
{  int pivot = arr[high];
   int i = (low - 1);
```

```
for( int j= low; j<= high -1 ; j++)
  { if(arr[j] <=pivot)
    { i++;
      swap(&arr[i], &arr[j]);
    }
  }
  swap(&arr[i+1], &arr[high]);
  return (i+1);
}
Void quick sort ( int arr[], int low, int high)
{ if ( low < high)
    { int pi= partition (arr, low, high);
    quicksort (arr, low, pi-1);
    quicksort (arr, pi+1, high);
    }
}
```

in average and best case time complexity O(nlogn)
in most case time complexity $O(n^2)$
But it is better among all sorting algorithm.
because it take O(1) space rather O(n) space
in merge sort.

K^{th} Smallest / Largest Element in Unsorted Array / Expected Linear time O(n)

input :- arr[] = {7, 10, 4, 3, 20, 15}
K = 3

Output :- 7

input :- arr[] = {7, 10, 4, 3, 20, 15}
K = 4

Output :- 10

```
#include <bits/stdc++.h>
using namespace std;
int randompartition (int arr[], int l, int r);

int KthSmallest (int arr[], int l, int r, int K);
{ if (K>0 && K<=r-l+1)
  { int pos = randompartition (arr, l, r);
    if (pos-l==K-1)
        return arr[pos];
    if (pos-l>K-1)
        return KthSmallest (arr, l, pos-1, K);
    return KthSmallest (arr, pos+1, r, K-pos-1+l);
  }
  return INT_MAX;
}
void swap (int *a, int *b)
{ int temp = *a;
  *a = *b;
  *b = temp;
}
```

```
int partition(int arr[], int l, int r)
{ int x = arr[r], i=l-1;
  for(int j=l; j<r; j++)
  { if(arr[j]<=x)
    { i++;
      swap(&arr[i], &arr[j]);
    }
  swap(&arr[i+1], &arr[r]);
  return i+1;
}
int randomPartition(int arr[], int l, int r)
{ int n= r-l+1;
  int pivot = rand() % n;
  swap(&arr[l+pivot], &arr[r]);
  return partition(arr, l, r);
}
int main()
{ int arr[] = {12, 3, 5, 7, 4, 19, 26};
  Cout << KthSmallest(arr, 0, n-1, k);
  return 0;
}
```

Time Complexity :- Worst Case $O(n^2)$

Average Case $O(n)$.

Output :- 5

Count of index pairs with equal elements in an array

Given an array of n elements. The task is to count the total number of indices (i, j) such that arr[i] = arr[j] and i ≠ j. (i != j)

Example :- input arr[] = {1, 2, 1}
output :- 1
as arr[0] = arr[1] the pair of indices is (0, 1).

input :- arr[] = {1, 1, 1}
Output :- 3
As arr[0] = arr[1] = arr[2] the pair of indices is (0, 1), (0, 2) and (1, 2)

Algorithm :- Count the frequency of each number and find number of pairs with equal elements.
Suppose, a number x appears k time at index $i_1, i_2 \dots i_k$. Then pick any two index i_x and i_y. which is counted as 1 pair. So $^nC_2$ is number of pairs such that arr[i] = arr[j] = x.

```
#include <bits/stdc++.h>
using namespace std;
int CountPairs(int arr[], int n)
{ unordered_map<int, int> mp;
  for(int i=0; i<n; i++)
      mp[arr[i]]++;

  int ans=0
  for( auto it = mp.begin(); it!=mp.end(); it++)
  { int Count = it -> second;
    ans += ((Count) * (Count-1))/2;
  }
```

```
    return ans;
}
int main()
{   int arr[] = {1, 1, 2};
    cout << countpairs(arr, n) << endl;
    return 0;
}
```

Time Complexity :- O(n)

;;; Count Pairs in array that hold i*arr[i] > j*arr[j].

$0 < i < j < n$, Count pairs(arr[i], arr[j])

Example :- arr[] = { 5, 10, 10, 2, 4, 1, 6 }
(indexes: 0 1 2 3 4 5 6)

Output :- 5 (10, 2) (10, 4) (10, 1) (2, 1) (4, 1)

Data Structure & Algorithm

Backtracking Algorithm

1. The Knight's Tour Problem :-

Activity Selection Problem / Greedy Algo-1

Given n activities with their start and finish time.
Select maximum number of activities that can be performed by a single person, assuming person can only work on single activity at a time.

Example :- given activity are sorted by finish time

1) input :- start[] = {10, 12, 20};
finish[] = {20, 25, 30};

Output :- 2 {0, 2}

2) input :- start[] = {1, 3, 0, 5, 8, 5};
finish{} = {2, 4, 6, 7, 9, 9};

Output :- 4 {0, 1, 3, 4}

Greedy choice is to always pick next activity whose finish time is least among remaining activities and start time is more than or equal to finish time of previously selected activity.
We can sort activities according to their finishing time so we always consider next activities as minm finish time activity.

i) Sort the activities according to their finishing time.
ii) Select the first activity from sorted array and print it.
iii) Do following for remaining activities in sorted array.
...a) if the start time of this activity is greater than or equal to finish time of previously selected activity then select this activity and print it.

```
#include <bits/stdc++.h>
using namespace std;
Struct Activity
{ int start, finish;
};
bool activitycompare (Activity S1, Activity S2)
{ return (S1.finish < S2.finish);
}
Void printmaxActivities (Activity arr[], int n)
{ Sort (arr, arr+n, activitycompare);
  int i=0;
  Cout << "(" << arr[i].start << "," << arr[i].finish << ")";
  For (int j=1; j<n; j++)
  { if (arr[j].start >= arr[i].finish)
    { Cout << "(" << arr[j].start << ","
           << arr[j].finish << ")";
      i=j;
    }
  }
} int main ()
{ Activity arr[] = { {5,9}, {1,2}, {3,4}, {0,6},
                    {5,7}, {8,9} };
  int n = size of (arr) / size of (arr[0]);
  PrintmaxActivities (arr, n);
  return 0;
}
```

→ if this statement true then S_1 come first S_2 second.

Ex:- { (2,4), (3,9) }

A < 9

So:- after sorting

{ (2,4), (3,9) }

Time complexity :- O(n log n)

Output :-

(1,2), (3,4), (5,7), (8,9)

Every

unique

numer

integer

Egyptia

For a

First

remainir

Example

#includ

using

void

Greedy Algorithm for Egyptian fraction

Every positive fraction can be represented as sum of unique unit fractions. A fraction is unit fraction if numerator is 1 and denominator is a positive integer. Ex:- 1/3.

Egyptian fraction Representation of 2/3 is 1/2 + 1/6

" " " 6/14 is 1/3 + 1/11 + 1/231

" " " 12/13 is 1/2 + 1/3 + 1/12 + 1/156

For a given number of form 'nr/dr' where nr < dr. First find greatest possible unit fraction, then recur for remaining part.

Example :- for 6/14 we first find ceiling of 14/6 i.e 3.

So first unit fraction becomes 1/3,

then recur for (6/14 - 1/3) i.e 4/42.

```
#include <iostream>
using namespace std;
void printEgyptian (int nr, int dr)
{ if (dr == 0 || nr == 0)
      return;
  if (dr % nr == 0)   // if numerator divides denominator
                      // then simply divide.
    { cout << "1/" << dr/nr; return;
  if (nr % dr == 0)
    { cout << nr/dr;
      Return;
    }
  if (nr > dr)
  { cout << nr/dr << " + ";
    printEgyptian (nr % dr, dr);
    return;
  }
```

```
    int n = dr/nr + 1;
    Cout << " 1/" << n << " + ";
    printEgyptian( nr*n - dr, dr * n);
}
int main()
{ int nr = 6, dr = 14;
    printEgyptian(nr, dr);
    return 0;
}
```

Job Sequencing Problem

Given an array of job where every job has a deadline and associated profit if job is completed before deadline. It is also given that job takes single unit of time, so minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

i) Sort all jobs in decreasing order of profit.

ii) iterate on jobs in decreasing order of profit.
   for each job (a) find a time slot j, such that slot is empty and j < deadline and j is greatest. Put job in this slot and mark this slot filled.
   b) if no such j exists, ignore job.

Input:-

JobID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output:- following is maximum profit sequence of jobs.

```
#include <bits/stdc++.h>
using namespace std;

struct Job
{ char id;
  int dead, profit;
};

bool Comparison (Job a, Job b)
{ return (a.profit > b.profit); }

void printJobScheduling (Job arr[], int n)
{ sort (arr, arr+n, comparision);
  int result[n];
  int slot[n];
  for (int i=0; i<n; i++)
      slot[i] = false;
  for (int i=0; i<n; i++)
  { for (int j= min (n, arr[i].dead)-1; j>=0; j--)
    { if (slot[j]==false)
      { result[j] = i;
        slot[j] = true;
        break;
      }
    }
  }
  for (int i=0; i<n; i++)
  { if (slot[i])
      cout << arr[result[i]].id << " ";
  }
}

Job arr[] = { {'a', 2, 100}, {'b', 1, 19}, ... };
```

by place
→ sort a first then b
if this statement true.

Job Sequencing Problem - Loss Minimization

We are given N jobs numbered from 1 to N. For each activity, let T_i denotes number of days required to complete job. for each day of delay before starting to work for job i, a loss of L_i is incurred. We are required to find a sequence to complete jobs so that overall loss is minimized. We can work on one job at a time.

Example input:- $L = \{3, 1, 2, 4\}$
$T = \{4, 1000, 2, 5\}$

Output:- 3, 4, 1, 2 we should first complete job 3, then job 4, 1, 2 respectively.

input :- $L = \{1, 2, 3, 5, 6\}$
$T = \{2, 4, 1, 3, 2\}$

Output:- 3, 5, 4, 1, 2

On this we have to sort the job according to ratio of L_i/T_i in descending order.

One more thing, to get most accurate result avoid dividing L_i by T_i. instead compare two ratio like a/b and c/d, $ad > bc$. for $a/b > c/d$. and pick them accordingly in lexigraphical order.

Job Selection Problem - loss minimization strategy /set 2

We are given a sequence of N goods of production number 1 to N. Each good has volume denoted by (V_i). The Constraint is that once a good has been completed its volume starts decaying at fixed percentage (P) per day.

All good decays at same rate and further each good take one day to complete.
We are required to find order in which goods should be produced so that overall volume of goods is maximized.

→ Approach is to make good first which have least volume.

let V₀ volume good made on i^{th} day
then, its volume remain after N day
$V(1-P)^{N-i}$.

input:- 4, 2, 151, 15, 1, 52, 12 P = 10%.

Output:- 222.503.

~~Huffman coding~~

e. Weighted job scheduling

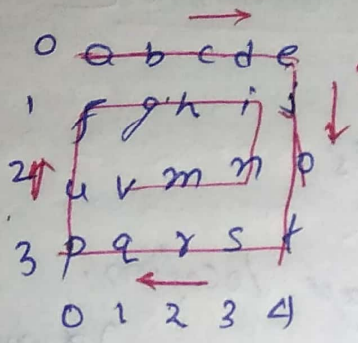
Find maxm profit if at a time only one job can be performed.

Ex:- {1, 2, 50} Ans:- 250 (job 1 and job 4)
{3, 5, 20}
{6, 19, 100}
{2, 100, 200}

```
struct job
{ int start, finish, profit;

};
bool compare (job a, job b)
{ return (a.finish < b.finish);

}
int latestNonConflict (job arr[], int i)
{ for (int j=i-1; j>=0; j--)
  { if (arr[j].finish <= arr[i].start)
      return j;

  }
  return -1;

}
```

```
int Profitmax (Job arr[], int n)
{ Sort (arr, arr+n, compare);
  int *table = new table[n];
  table[0] = arr[0].profit;
  for (int i=1 ; i<n ; i++)
  { int include = arr[i].profit;
    int l = lastNonConflict (arr, i);
    if (l!=-1)
        include += table[l];
    table[i] = max (table[i-1], include);
  }
int result = table[n-1];
delete[] table;
  return result;
}
```

Print matrix in Spiral form




```
Void Print_spiral ( m, n, mat[][] )
{ int i, k=0, l=0;
                  1 for row   2 for Columns.
  int last_row = m-1, last_col = n-1;
  while ( k <= last_row && l <= last_col )
  { for (i = l ; i <= last_col ; i++)
        Cout << mat[k][i] << ' ';
    k++;
    for ( i = k ; i <= last_row ; i++)
        Cout << mat[i][last_col] << ' ';
    last_col --;
    if ( k <= last_row)
    { for ( i = last_col ; i >= l ; i--)
        Cout << mat[last_row][i];
      last_row --;
    }
    if ( l <= last_col)
    { for ( i = last_row ; i >= k ; i--)
        Cout << mat[i][l];
      l++;
    }
  }
}
```

Check prime number

```
bool isprime(int n)
{
    if (n<=1)
        return false;
    if (n<=3)
        return true;
    if(n%2==0 || n%3==0)
        return false;
    for(int i=5; i*i<=n; i+=6)
    {   if (n%i==0 || n%(i+2)==0)
            return false;
    }
    return true;
}
```

time complexity :- $O(n^{1/2})$

Print all prime number 1 to n

Using above time complexity $O(n * n^{1/2}) = O(n^{3/2})$

```
for (int i=2; i<=n; i++)
    if( isprime(i))
        cout<<i<<endl;
```

→ Use sieve of Eratosthenes :-

Concept :- Create boolean array prime[n+1], initialize with true. then finally a value in prime[i] is false if i is not a prime. else prime true.

```
Void sieveofEratosthenes(int n)
{   bool prime[n+1] = {true};
    prime[0] = prime[1] = false;
```

```
for (int i=2 ; i*i<=n ; i++)
{ if (prime[i])
    { for (int p=i*i ; p<=n ; p+=i)
        prime[p] = false;
    }
}
for (int i=0 ; i<n+1 ; i++)
    if (prime[i])
        cout << prime[i] << " ";
}
```

time :- O(n log log n)

Job Sequencing Problem (Greedy)

I/P

	J_0	J_1	J_2	J_3
deadline	4	1	1	1
profit	70	80	30	100

O/P :- 170

deadline	2	2	3	3
profit	50	60	20	30

O/P :- 140

Rules :-

i) one unit time taken by every job

ii) only one job can be assigned at a time.

iii) time starts at 0.

```
Job
{ int dead;
  int profit;
}

bool Compare( Job a, Job b)
  { return (a.profit > b.profit);
  }

void max-profit ( Job arr[ ], int n)
  { sort(arr, arr+n, Compare) ;
    int max_time = 0;
    for ( int i=0; i<n ; i++)
        max_time = max( max_time, arr[i].dead);
    int profit = 0;
    bool slot [max_time ] = {false};
    for (int i=0 ; i<n ; i++)
     { if ( slot[ arr[i].dead -1 ] == false)
        { profit += adj[i].profit; slot [arr[i]. dead-1 ] = true; }
       Else
        { int k = arr[i].dead -2;
          While (k>=0 && slot[k] )
            { k--; }
```

```
if ( k<
  continue
else
  { slot
    profit
  }
}
} return
}
```

```
if (k<0)
   continue;
else
  { slot[k]=true;
    profit += adj[i].profit;
   }
}
} return profit;
}
```

→ Sort job according to profit

1	1	4	1
100	80	70	30

Slot[] :- | 100 | | | 70 | = 170

2	2	3	3
60	50	30	20

Slot[] :- | 50 | 60 | 30 | |

Ans :- 140.

3 = true; }

'\t' → for tab
'\n' → for newline

ascii Chart

0 — 48	A — 65	a — 97
1 — 49	B — 66	b — 98
2 — 50	C — 67	c — 99
3 — 51	D — 68	d — 100
4 — 52	E — 69	e — 101
:	F — 70	:
:	:	:
9 — 57	Z — 90	z — 122

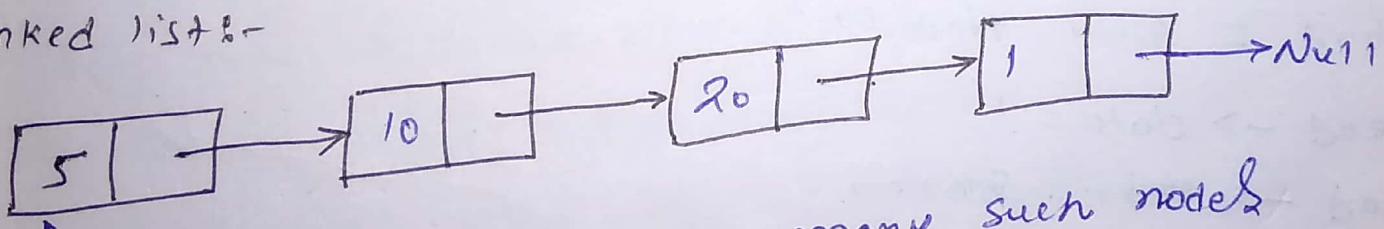
Palindrome :-
Word which is symmetrical from front and end.
9 1019
9 1419
Ompraksht shakarpno

Single linked list

it is a way to store a collection of elements. like an array these can be integers or characters. Each element in a linked list is stored in form of node.

Node :- | Data 10 | Next Pointer | } Node Collection of two sub-elements or parts.

Linked list :-



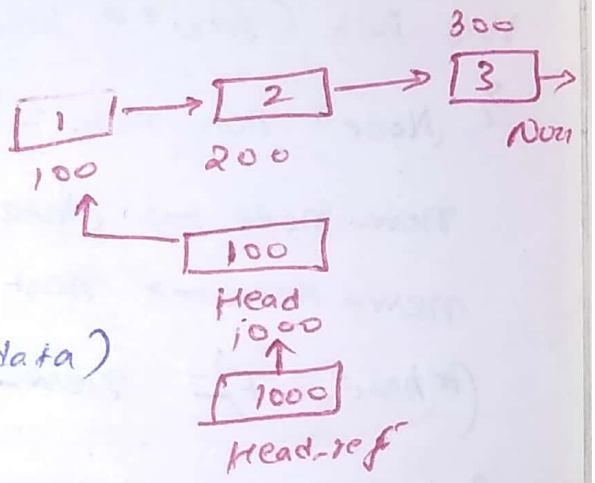
A linked list is formed when many such nodes are linked together to form a chain. First node is always used as a reference to traverse the list and is called Head. Last node points to null.

Creation of linked list with 3 Node:-

```
#include <bits/stdc++.h>
using namespace std;
class Node
{ public:
    int data;
    Node* next;
};
int main()
{
  Node* head = NULL;
  Node* second = Null;
  Node* third = NULL;
  // allocate three nodes in heap
  head = new Node();
  second = new Node();
  third = new Node();
  head -> data = 1;
  head -> next = second;
  second -> data = 2;
  second -> next = third;
  third -> data = 3;
  third -> next = NULL;
  return 0;        PrintList(head);
}
```

// for printing Contained of list write function
printlist ( )

```
Void printlist( Node * n)
{ while (n != NULL)
  {
  cout << n -> data << "  ";
  n = n -> next ;
  }
}
```



insert Node in linked list

1> At front of linked list

```
Void Push ( Node ** head_ref , int new_data)
{ /allocate node */
  Node * new_node = new Node ();
  // put data
  new_node -> data = new_data;
  new_node -> next = (* head_ref) ;
  (*head_ref) = new_node ;
```

Time Complexity O(1)

```
}
```

2> insert b/w node

```
#include <bits/stdc++.h>
Using namespace std;
Class Node{
Public:
    int data;
    Node * next;
};
Void Push (Node ** head_ref , int new_data)
{ Node* new_node = new Node();
  new_node -> data = new_data;
  new_node -> next = (* head_ref);
  (*head_ref)= new_node;
};
Void insertAfter (Node * Prev_node , int new_data)
{ Node* new_node = new Node();
  if (Prev_node == NULL)
  { cout << " the given previous node cannot be NULL";
    return;
  }
new_node -> data = new_data;
new_node -> next = Prev_node -> next;
Prev_node -> next = new_node;
}
```

time :- O(1)

```
void append (Node ** head_ref, int new_data)
{ Node * new_node = new Node ();
  Node * last = *head_ref;
  new_node -> data = new_data;
  new_node -> next = NULL;
  if (*head_ref == NULL)
    { *head_ref = new_node;
      return;
    }
  while (last -> next != NULL)
      last = last -> next;
  last ->next = new_node;
  return;
  }
void printlist ( Node *node)
{ while ( node != NULL)
  { cout<< " " << node -> data;
    node = node -> next;
  }
}
int main ()
{ Node *head = NULL;
  append (&head, 6);
  Push (&head, 7); push (&head, 2);
  Append (&head, 4); insert ( head -> next, 8);
  Printlist (head); return 0;
}
```

Output:-
1 2 8 6 4

// function to delete linked list

```
void deletelist (Node** head_ref)
{
    Node* current = *head_ref;
    Node* next;
  while (current != NULL)
    { next = current->next;
      free(current);
      current = next;
    }*head_ref = NULL;
}
```

Length of a linked list (iterative & Recursive)

iterative methode ——

```
int getCount (Node* head)
{ int count = 0;
  Node* current = head;
  while (current != NULL)
    { count++;
      current = current->next;
    }
  return count;
}
```

Call:- getCount(head);

Recursive Soln :-

```
int getCount (Node * head)
{ if (head == NULL)
    return 0;
  Else
    return 1 + getCount (head -> next);
}
```

Search an element in a linked list

1> iterative Soln :-

```
bool search (Node* head, int x)
{ Node* current = head;
  while (current != NULL)
  { if (current -> data == x)
      return true;
    current = current -> next;
  }
  return false;
}
```

Recursive Soln :-

```
bool search (Node *head, int x)
{ if (head == NULL)
    return false;
Else if (head -> data == x)
    return true;
Else
    return search (head -> next, x);
}
```

Function to get Nth node in linked List

GetNth() function takes a linked list and integer index returns data valued stored in node at given index.

1) iterative Soln :-

```
int getNth (Node *head , int index)
{ Node * Current = head;
  int Count = 0;
  while (Current != NULL)
  { if (count == index)
        return (Current -> data);
    Count ++;
    Current = Current -> next;
  }
  assert(0);
}
```

2) Recursive Soln :-

```
int getNth (Node * head ,int index)
{   int Count=0;
    if (head == NULL)
        assert(0);
    else if (index == Count)
        return head -> data;
    else return GetNth (head -> next, index-1);
}
```

function to get Nth node from end of linked list

```
Void printNthfromlast ( Node * head, int n )
{ int len=0 , i;
  Node* temp = head;
  while ( temp !=NULL) {
    temp = temp -> next;
      len++;
  }
  if (len< n)
    return;
temp = head;
for (i=1 ; i< len- n+1; i++)
   temp  = temp -> next;
   Cout<< temp -> data;
return;
}
```

Time Complexity :- O(n)

method-2 (Use two pointers)

```
void printNthfromlast ( Node * head, int n)
{ Node * main_ptr = head;
  Node* ref_ptr = head;
  int Count=0
  if (head !=NULL)
  { while (Count <n)
    { if (ref_ptr == NULL)
      { Cout<<n<<"is greater than no. of nodes";
        return ;
```

```
        ref_ptr = ref_ptr -> next;
        count++;
    }
while (ref_ptr != NULL)
{ main_ptr = main_ptr -> next;
  ref_ptr = ref_ptr -> next;
}
cout << main_ptr -> data;
}}
```

Print middle of given linked list if odd length if even length print second middle element

Method 1 :- Using length print length/2 + 1 Node ~~if len Odd~~

Method 2 :- Two pointer method (increase one pointer by 2, increase other " by 1)

↓ this give middle elemen

```
void printmiddle (Node *head)
{ Node * slow_ptr = head;
  Node * fast_ptr = head;
  if (head != NULL)
  { while ( fast_ptr != NULL && fast_ptr -> next != NULL)
    { fast_ptr = fast_ptr -> next -> next;
      slow_ptr = slow_ptr -> next;
    }
    ~~printf( slow~~ cout << slow_ptr -> data;
}}
```

Method-3

```
Void PrintMiddle (Node * head)
{ int Count = 0;
  Node * mid = head;
  While (head != NULL)
  { if (Count%2)
        mid = mid -> next;
    ++Count;
    head = head -> next;
  }
  if (mid != NULL)
    cout << mid -> data;
}
```

Change mid if Count is odd so it takes half step of head.

Function Counts Number of times a given int occurs in linked list.

Method-1 iterative Soln :-

```
int Count (Node * head, int Search_int)
{ Node * Current = head;
  int Count = 0;
  while (Current != NULL)
  { if (Current -> data == Search_int)
        Count ++;
    Current = Current -> next;
  }
  Return Count;
}
```

Time Complexity O(n)

Auxiliary space O(1)

Method 2 :- Using Recursion

```
int frequency = 0;  → global variable declare
int Count (Node * head, int Search_element)
{ else if (head -> data == Search_element)
        frequency ++;
  if (head == NULL)
      Return frequency;
  else return Count (head -> next, Search_element);
}
```

without global variable :-

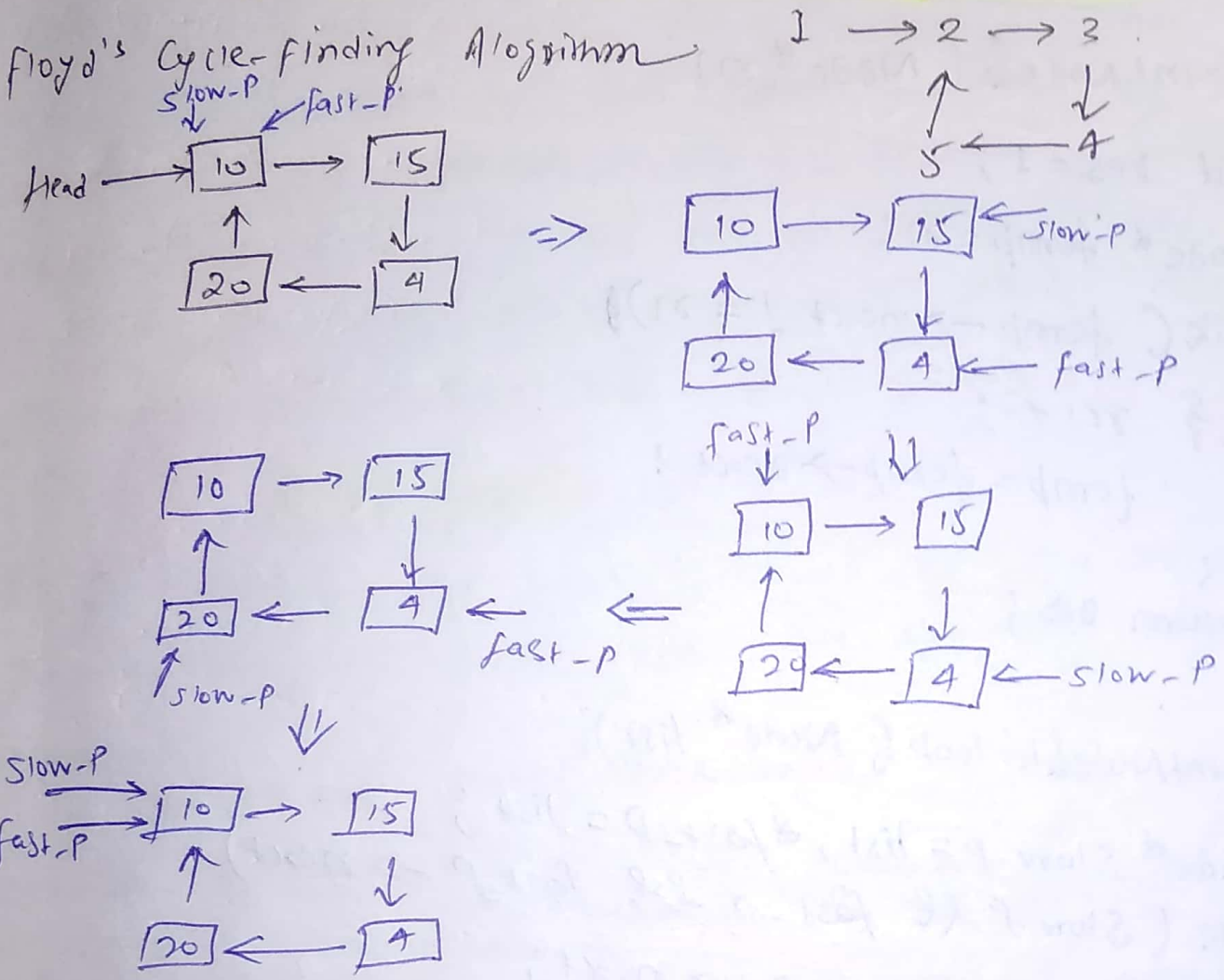
```
int Count (Node * head, int Search)
{ if (head == NULL)
      Return 0;
  Else if (head -> data == Search)
      return 1 + Count (head -> next, Search);
  Else
      return Count (head -> next, search);
}
```

Time Complexity in each O(n).

Space " O(n)

Detect loop in linked list

Floyd's Cycle-finding Algorithm



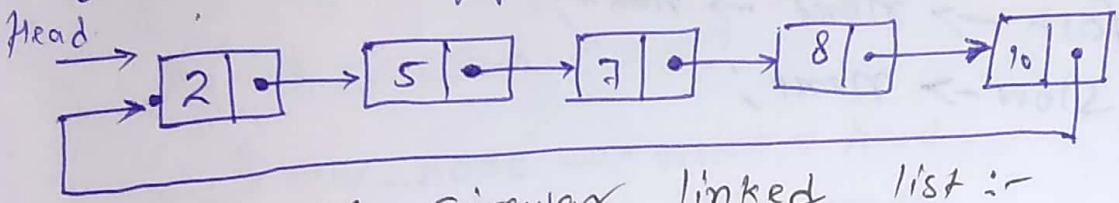

```
int detectloop (Node* list)
{ Node* slow_p = list, *fast_p = list;
  while (slow_p && Fast_p && fast_p -> next)
  { slow_p = slow_p -> next;
    fast_p = fast_p -> next -> next;
    if (slow_p == fast_p)
        Return 1;
  }
  return 0;
}
```

find length of loop in linked list

```
int countNodes(Node* n)
{ int res = 1;
  Node* temp = n;
  while(temp -> next != n)
    { res++;
      temp = temp -> next;
    }
  return res;
}
int countNodesinloop(Node* list)
{ Node* slow_p = list, *fast_p = list;
  while(slow_p && fast_p && fast_p -> next)
    { slow_p = slow_p -> next;
      fast_p = fast_p -> next -> next;
      if(slow_p == fast_p)
        return countNodes(slow_p);
    }
  return 0; // indicate that their is no loop
}
```

Circular linked list

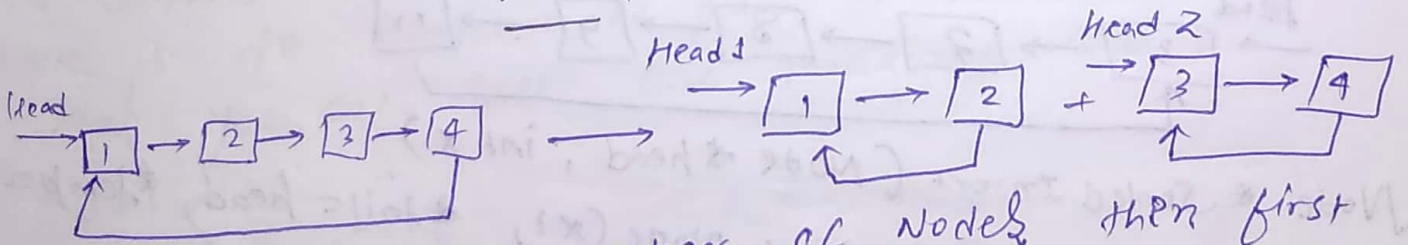
A linked list where all nodes are connected to form a circle. there is no NULL at end. It can be singly or doubly Circular linked list.



traversal of Circular linked list :-

```
Void printlist (struct Node * first)
{ struct Node * tmp = first;
  if (first != NULL)
  { printf ("%d", temp -> data);
    temp = temp -> next;
  while (temp != first)
  { printf ("%d", temp -> data);
    temp = temp -> next;
  }
  }
}
```

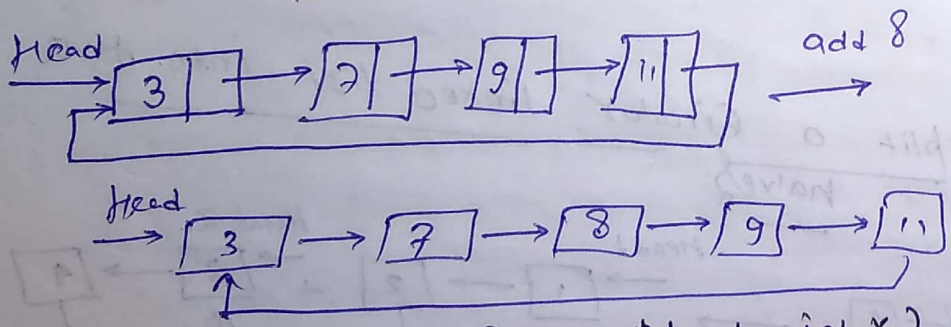
Split a Circular linked list into two halves



if there are odd number of Nodes then first list Contain One extra Nodes.

```
Void Split list ( Node *head, Node **head1_ref, Node **head2_ref)
{ Node *slow = head, *fast = head, *temp = NULL;
  while (fast -> next -> next != head && fast -> next != head)
  { fast = fast -> next -> next;
    slow = slow -> next;
  }
*head2_ref = slow->next;
if (fast -> next == head)
      fast -> next = *head2_ref;
  else fast -> next -> next = *head2_ref;
slow -> next = head;
*head1_ref = head;
}
```

Insert New data into Circular linked list




```
Node * Sorted Insert (Node *head , int x)
{ Node * new_node = new Node (x), *tail = head, *temp = head;
  while (tail -> next != head)
      tail = tail -> next;
```

ref) if (head -> data >= x)

{ new_node -> next = head;

head) tail -> next = new_node;

return new_node;

}

if (tail -> data <= x)

{ new_node -> next = head;

tail -> next = new_node;

return head;

}

while (temp->next != head)

{ if (temp -> next -> data >= x)

{ new_node -> next = temp -> next;

temp -> next = new_node;

return head;

}

temp = temp -> next;

}

Josephus Circle using Circular linked list

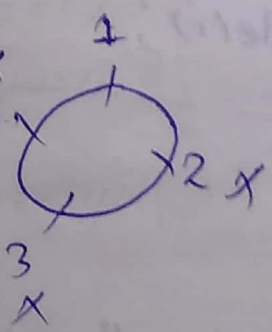
Given the total number of persons n in a circle and a number m which indicates that m person are skipped and m^{th} person is killed in circle. Task is to choose the place in initial circle so that you are last one remaining and so survive.

head;

input:- n=4

m=2

Output :- 1



input :- n=5

m=3

Output :- 4

i) Create a Circular linked list of size n.
ii) Traverse through linked list and one by one delete every mth node until there is one node left.
iii) Return value of only left node.

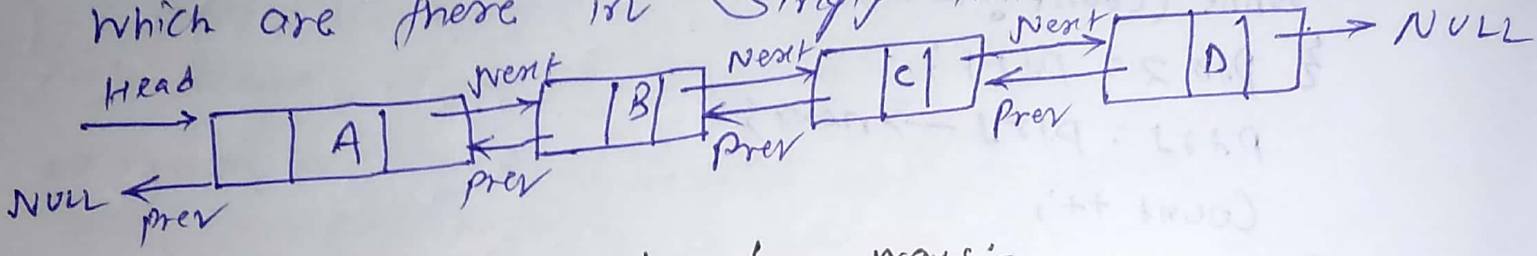
```
#include <bits/stdc++.h>
using namespace std;
Struct Node
{ int data;
  Struct Node * next;
};
Node *newNode (int data)
{ Node * temp = new Node;
  temp -> next = NULL;
  temp -> data = data;
  return temp;
}
Void getJosephusPosition ( int m, int n)
{ Node *head = newNode(1);
  Node *prev = head;
  for (int i=2; i<=n; i++)
    { prev -> next = newNode(i);
      Prev = prev -> next;
    }
  Prev -> next = head;
//while only one node is left in linked list
 Node *ptr1 = head, *ptr2 = head;
```

```
while (ptr1 -> next != ptr1)
{ //find m-th node
   int count=1;
   while (count != m)
   { ptr2 = ptr1;
     ptr1 = ptr1 -> next;
     count ++;
   }
  //Remove m-th node
  ptr2 -> next = ptr1 -> next;
  free (ptr1);
  ptr1 = ptr2 -> next;
}
Printf ("%d", ptr1->data);
}
int main ()
{  int n=14, m=2;
   getjosephusposition (m,n);
   return 0;
}
int jos (int n; int k)
{  if (n==1)
      return 0;
   else return (jos (n-1, k)+k) %n;
}
```

Time complexity : O(m*n)

Doubly Linked List (intro and insertion)

It contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list: next



A node can be added in four ways:-

i) At front of Double linked list.

ii) After a given node.

iii) At end of DLL.

iv) before a given node.

Add a Node at front (5 step process)

```
Void Push (struct Node ** head_ref, int new_data)
{ struct Node * new_node = new Node ;
    new_node -> data = new_data;
    new_node -> next = * head_ref;
    new_node -> prev = NULL;
    if ( (*head_ref) != NULL )
        *head_ref -> Prev = new_node;
    (*head_ref) = new_node;
}
```

Add a Node after a given node (7 step)

```
Void insertAfter ( struct Node * Prev_node, int new_data)
{   if ( Prev_node == NULL)
        Return ;
    Struct Node * new_node = new Node ;
```

```
new_node -> data = new_data;
new_node -> next = prev_node -> next;
new_node -> prev = prev_node;
prev_node -> next = new_node;
// change previous of (new_node's next node)
if (new_node -> next != NULL)
    new_node -> next -> prev = new_node;
}
```

Add a node at end (7 step)

```
Void append (struct Node **head_ref, int data)
{ struct Node * new_node = new Node;
  new_node -> data = data;
  new_node -> next = NULL;
  struct Node * last = *head_ref;
  if (*head_ref == NULL)  // if linked list is empty.
                          //make new node as head;
  { new_node -> prev = NULL;
    *head_ref = new_node;
    return;
  }
  while (last -> next != NULL)
      last = last -> next;

  last -> next = new_node;
  new_node -> prev = last;
  return;
}
```

Add a node before a given node.

```
Void insertbefore (struct Node **head_ref, struct Node *next_node,
                   int new_data)
{
  if (next_node == NULL)
      return;
  Struct Node * new_node = new Node;
  new_node -> data = new_data;
  new_node -> prev = next_node -> prev;
  next_node -> prev = new_node;
  new_node -> next = next_node;
  if (new_node -> prev != NULL)
      new_node -> prev -> next = new_node;
  Else
      *head_ref = new_node;
}
```

Delete a node in Doubly linked list

Deletion of node in doubly linked list divided in three main Categories :-

i) Deletion of head node
ii) Deletion of middle "
iii) Deletion " end "

```
Void deleteNode (Node **head_ref, Node *del)
{
  if (del == *head_ref)
    { *head_ref = pos -> next;
      pos -> next -> prev = NULL;
      free (pos); return; }
```

```
if (pos -> next == NULL)
    { pos -> prev -> next = NULL;
      free(pos);
      return;
    }
pos -> prev -> next = pos -> next;
pos -> next -> prev = pos -> prev;
free(pos);
return;
}
```

node,

Reverse a doubly linked list

```
Void reverse (Node **head_ref)
{ Node *temp = NULL;
  Node *Current = *head_ref;
  while (Current != NULL)
  { temp = Current -> prev;
    Current -> prev = Current -> next;
    Current -> next = temp;
    Current = Current -> prev;
  }
  // Before Changing the head, check for case like empty
     // list and list with only one node.
  if (temp != NULL)
      *head_ref = temp -> prev;
}
```

Here Concept is to Swap prev and next pointers of all nodes, Change prev of head (or start) and Change head pointer in end.

Merge Sort for Doubly linked list

```
Node *merge (Node *first, Node *second)
{ if (first == NULL)
      return second;
  if (second == NULL)
      return first;
  if (first -> data < second -> data)
  { first -> next = merge (first->next, second);
      first -> next -> prev = first;
      first -> prev = NULL;
      return first;
  }
  else
  { second -> next = merge (first, second -> next);
    second -> next -> prev = second;
    second -> prev = NULL;
    return second;
  }
Node *split (Node *head)
  { Node *fast = head, *slow = head;
    while (fast -> next && fast -> next -> next)
    { fast = fast -> next -> next;
      slow = slow -> next;
    }
    Node *temp = slow -> next;
      slow -> next = NULL;
      return temp;
  }
```

```
Node *mergesort(Node *head)
{   if (!head || !head->next)
        return head;
    Node *second = split(head);
    head = mergesort(head);
    second = mergesort(second);
    return merge(head, second);
}
```

Find Pairs with given sum in doubly linked list

Given a Sorted doubly linked list of positive distinct elements, the task is to find pairs in doubly LL whose sum is equal to given X. without using an extra space.

Input :- 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9

X = 7

Output :- (6,1), (2,5)

Expected time complexity :- O(n)

Auxiliary space :- O(1)

```
Void pairSum(Node *head, int x)
{   struct Node *first = head;
    "    "    *second = head;
    while(second->next != NULL)
        second = second->next;
    bool found = false;
    while(first != NULL && second != NULL &&
          first != second && second->next != first)
    {   if((first->data + second->data) == x)
        {   found = true;
            cout << first->data << ", " << second->data
                 << endl;
            first = first->next;
```

```
        second = second -> prev;
    }
    else
    { if ((first -> data + second -> data) < x)
            first = first -> next;
      Else
            second = second -> prev;
    }
}
if (found == false)
    cout << "No Pair found";
}
```

(1) first part

Dynamic Programming

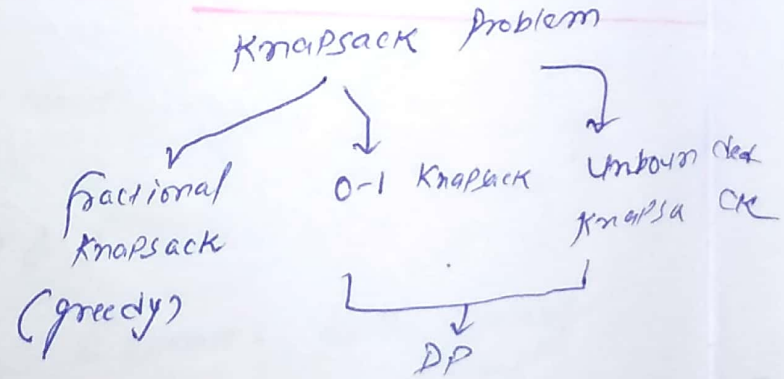
- i) Choice [Recursion call thro terms]
- ii) Optimal (max^m, min^m, ...)

Recursive
- Memoziation
- Top-down

1) 0-1 Knapsack (6) → Only one no. of item.
2) Unbounded Knapsack (5) → item number not limited
3) Fibonanci (3)
4) LCS (15)
5) LIS (10)
6) Kadane's Algorithm (6)
7) matrix chain multiplication (7)
8) DP on trees (4)
9) DP on grid (14)
10) Others (5)

0-1 Knapsack Problem

- i) Subset Sum
- ii) Equal sum partition
- iii) Coun of Subset sum
- iv) Minm Subset Sum diff
- v) Target Sum
- vi) No Of Subset Sum given difference



input wt[] :-

1	3	4	5

val[] :-

1	4	5	7

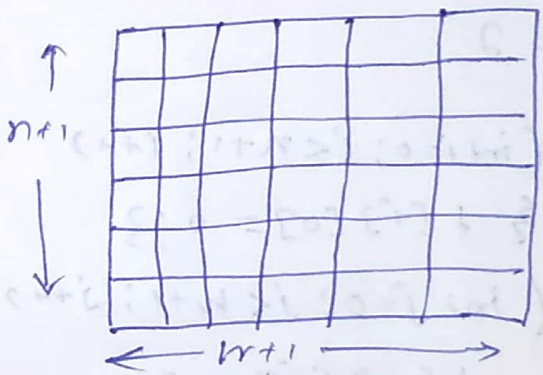
O/P → max^m profit

W = 7

```
int knapsack ( int wt[], int val[], int W, int n)
{   if (n==0 || W==0)
        return 0;
    if ( wt[n-1] <= W)
        { return max (val[n-1] + knapsack (wt[], val[],
                                  W-wt[n-1], n-1)
                      knapsack (wt[], val[], W, n-1); }
    Else
        return knapsack (wt[], val[], W, n-1);
}
```

Memoization :-

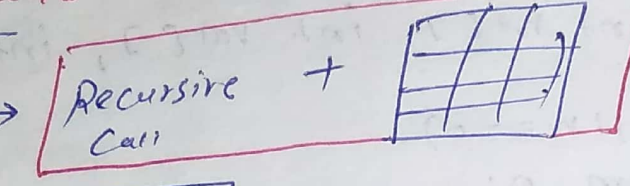
```
int t[n+1][W+1];
memset (t, -1, sizeof(t));
```

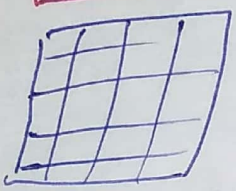



```
int knapsack (int wt[], int val[], int W, int n)
{   if (n==0 || W==0)
        return 0;
    if (t[n][W] != -1)
        return t[n][W];
    if (wt[n-1] <= W)
        return t[n][W] = max (val[n-1] + knapsack (wt, val,
                                  W-wt[n-1], n-1),
                              knapsack (wt, val, W, n-1));
    else return t[n][W] = knapsack (wt, val, W, n-1);
}
```

memoization demerits :- full stack size by recursive call.

Top-down approach :-

Memoized (dp) → Recursive Call +  → stack overflow

top-down →  only matrix.

↓

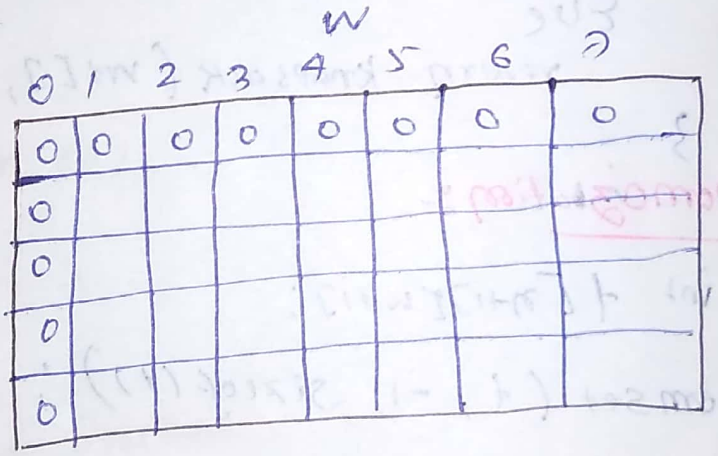
Best step-1 initialization

step-2 iterative function

wt[] = [• 1 3 4 5] → n = 4

val[] = [1 4 5 7]

W = 7




```
for(int i=0; i<n+1; i++)
  { t[i][0] = 0; }
for(int j=0; j<W+1; j++)
    t[0][j] = 0;
  for(int i=1; i<n+1; i++)
    for(int j=1; j<W+1; j++)
    { if(wt[i-1] <= j)
        t[i][j] = max(val[i-1] + t[i-1][j-wt[i-1]]
                      , t[i-1][j]);
      else
      t[i][j] = t[i-1][j];
    }
```

return t[n][W];

Subset Sum Problem

arr[] : 2 3 7 8 10] Present or Not

Sum = 11

Does any subset exist in given array so that its Sum is 11.

t[n+1][W+1] = t[6][12]

arr size	0	1	2	3	4	5	6	7	8	9	10	11
0	T	F	F	F	F	F	F	F	F	F	F	F
1	T											
2	T											
3	T											
4	F											
5	T											

Empty subset

Sum

```
for (int i=0; i<n+1; i++)
    t[i][0] = true;
for (int j=1; j<W+1; j++)
    t[0][j] = false;
for (int i=1; i<n+1; i++)
for (int j=1; j<W+1; j++)
{ if (arr[i-1]<=j)
    t[i][j] = t[i-1][j-arr[i-1]] || t[i-1][j];
  Else
    t[i][j] = t[i-1][j];
}
    Return t[n][Sum];
```

Equal Sum Partition

arr[] = {1, 5, 11, 5}

O/p : True / False

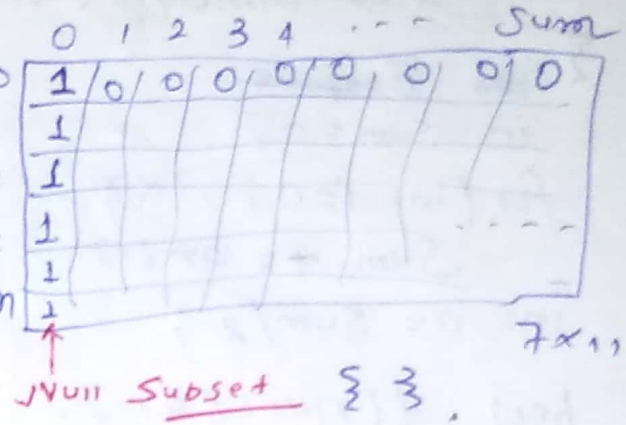
```
boolean SubsetSum (int arr[], int sum, int n)
{ bool t[n+1][sum+1];
  for(int i=0; i<n+1; i++)
      t[i][0] = true;
  for(int j=1; j<sum+1; j++)
      t[0][j] = false;
  for(int i=1; i<n+1; i++)
  for(int j=1; j<sum+1; j++)
    { if(arr[i-1]<=j)
        t[i][j] = t[i-1][j] || t[i-1][j-arr[i-1]];
      Else
        t[i][j] = t[i-1][j];
    }
return t[n][sum];
}
boolean CaualSumpartition (int arr[], int n)
  { int Sum = 0;
    for(int i=0; i<n; i++)
      Sum = Sum + arr[i];
    if (Sum % 2 == 0)
    return SubsetSum (arr, sum/2, n);
    Else return false; }
```

Count of Subset Sum of given Sum

input :- arr[] 2 3 5 6 8 10

Sum : 10

t[n+1][sum+1] = t[7][11]




```
int count(int arr[], int sum, int n)
{ int t[n+1][sum+1];
for(int i=0; i<n+1; i++)
    t[i][0] = 1;
for(int j=1; j<sum+1; j++)
    t[0][j] = 0;
for(int i=1; i<n+1; i++)
 for(int j=1; j<sum+1 ; j++)
  { if(arr[i-1] <= j)
      t[i][j] = t[i-1][j] + t[i-1][j-arr[i-1]];
    else
      t[i][j] = t[i-1][j];
  }
 return t[n][sum];
}
```

Minimum Subset Sum difference

arr[] : 1 6 11 5 ⇒ {1+6+5} {11}

12 − 11 = 1

OP :- 5

P_1 { 3 } S_1 P_2 { 3 } S_2 $S_1 - S_2 = min$

min abs($S_1 - S_2$)

we have to find position of array into two so that it sum ~~of~~ difference become minimum.

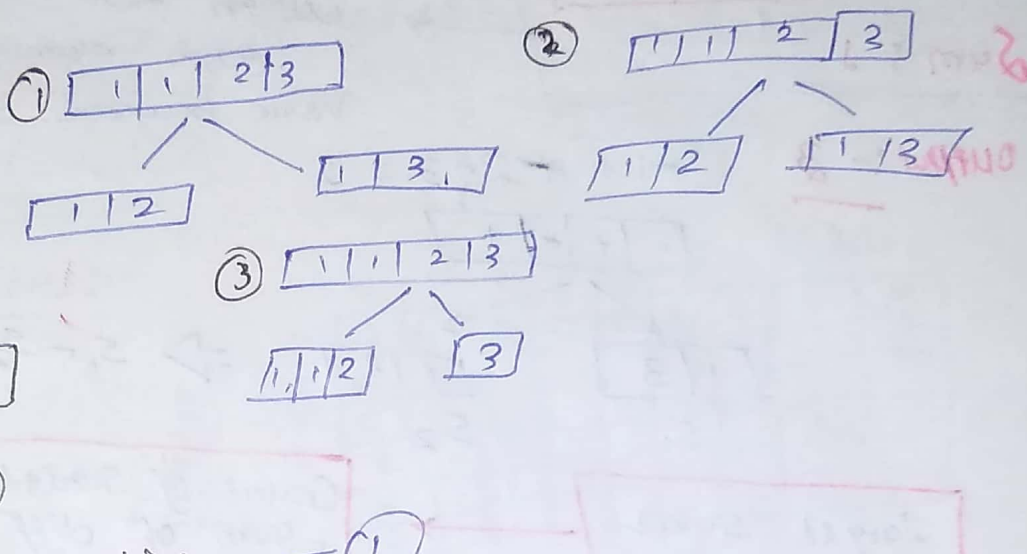
```
int SubsetSumdiff (int arr[], int n)
{
  int Sum = 0;
  for (int i=0; i<n; i++)
     Sum += arr[i];
  int a = Sum/2;
  bool t[n+1][a+1];
  for (int i=0; i<n+1; i++)
        t[i][0] = true;
  for (int j=1; j<a+1; j++)
        t[0][j] = false;
  for (int i=1; i<n+1; i++)
   for (int j=1; j<a+1; j++)
    { if (arr[i-1] <= j)
        t[i][j] = t[i-1][j] || t[i-1][j-arr[i-1]];
      else
        t[i][j] = t[i-1][j];
    }
int result;
 for (int i=a; i>=0; i--)
    if (t[n][i] == true)
       { result = i;
         break;
       }
return Sum - 2*result;
}
```

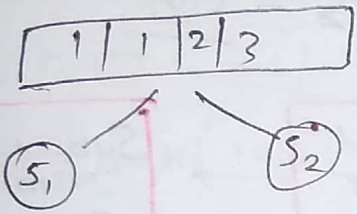
Count the number of Subset Sum given difference

Input arr[] = | 1 | 1 | 2 | 3 |

diff = 1;

Output = 3





Sum(S1) - Sum(S2) = diff — (1)

Sum(S1) + Sum(S2) = Sum(arr) — (2)

2 Sum(S1) = diff + Sum(arr)

$\Rightarrow Sum(S_1) = \frac{diff + Sum(arr)}{2}$

Count of no. of Subset Sum of given diff = Count of Subset Sum of (S_1)

```
int Sum = (diff + Sum of arr) / 2 ;
if ((diff + sum of arr) % 2) == 0
    return Count of subset sum (arr, sum);
Else return 0;
```

Target Sum :-

arr: | 1 | 1 | 2 | 3 |

Sum : 1

→ Count no of ways to assign '+' Or '-' in front of each element so that expression value become Sum value.

output :- 3 $-1+1+2+3 = 1$

| - | + | - | + |

| 1 | 3 | S_1 | | 1 | 2 | S_2

$\Rightarrow S_1 - S_2 = diff$

target sum → Count of Subset Sum of diff given → Subset sum of S_1

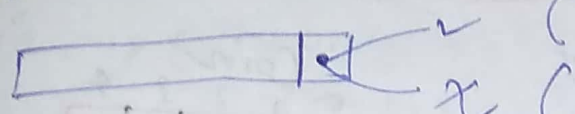
```
int sum = (diff + sum of arr) / 2 ;
if ((diff + sum of arr) / 2) == 0
    return Count of Subset sum (arr, sum);
Else return 0;
```

Unbounded Knapsack (5)

↳ Infinite no. of copy of each element.

1) Rod cutting
2) Coin change 1 (maximum no. ways)
3) Coin change 2 (minimum no. of coin used)
4) maximum ribbon cut

gn Unbounded knapsack



(can be processed later again)

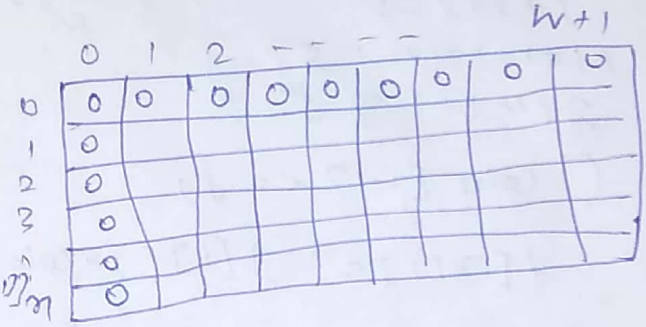
(processed)

Input wt[] :- | 1 | | | |

val[] :- | | | | |

W = _ :-

t[n+1][W+1]



Unbounded knapsack :-

```
if ( wt[i-1] <= j)
t[i][j] = max( val[i-1] + t[i][j-wt[i-1]],
               t[i-1][j] );
else
t[i][j] = t[i-1][j];
```

Rod Cutting

maximum profit by proper cutting ~~property~~ of rod.

Length[] :- | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

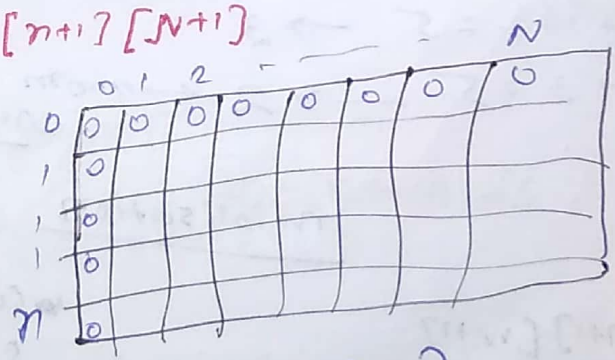
Price[] :- | 1 | 5 | 8 | 9 | 10 | 17 | 19 | 20 |

N :- 8

↓

Length of rod

t[n+1][N+1]




```
if ( Length[i-1] <= j)
t[i][j] = max( price[i-1] + t[i][j-length[i-1]], t[i-1][j] )
else
t[i][j] = t[i-1][j];
```

Coin change-1 (maxm no of ways)

Coin[]: | 1 | 2 | 3 | — infinite no. of supply

Sum :- 5

| 2 | 3 |, | 1 | 1 | 3 | } 5 ways.

1+2+2=5
1+1+1+1+1=5
2+1+1+1=5

Sum \ Coin	0	1	2	3	4
0	1	1	1	1	1
1	0				
2	0				
3	0				
4	0				
5	0				

```
if ( coin [i-1] <= j)
    t[i][j] = t[i][j-coin[i-1]] + t[i-1][j];

Else  t[i][j] = t[i-1][j];
```

Coin-change 2 (minm no. of coin used)

Coin[]: | 1 | 2 | 3 |

Sum : 5

1+1+1+1+1=5 → 5
1+1+1+2=5 → 4
1+1+3=5 → 3
1+2+2=5 → 3
2+3=5 → 2 ← minm no. of coin.

initialisation

t[n+1][w+1]

t[n+1][sum+1]

t[4][6]

Coin[]:-

Sum :-

	0	1	2	3	4	5
0	∞	∞	∞	∞	∞	∞
1	a					
2	a					
3	a					

(sum)

INT_MAX = ∞ - 1

(n)

```
for(int j=1; j< sum+1 ; j++)
  { if( j% arr[0] ==0)
        t[1][j] = j/arr[0];
    else
        t[1][j] = INT_MAX -1;
  }

for (      )
for(       )
  { if( coin[i-1] <=j)
    t[i][j] = min ( t[i][j-coin[i-1]] +1,
                    t[i-1][j] );
    else
      t[i][j] = t[i-1][j];
  }
```

Longest Common Subsequence (LCS)

1) Longest Common Substring
ii) Print Longest Common subsequence
iii) Shortest Common Supersubsequence
iv) print SCS
v) minm no. of insertion and deletion to make string a to string b.

vi) Longest repeating subsequence.
vii) Length of largest subsequence of a which is substring in b.
viii) subsequence pattern matching
ix) Count how many times a appears as subsequence in b.

x) Longest palidromic Subsequence

xi) " " Substring

xii) Count of palidromic Substring

xiii) minm no. of deletion in a string to make it a palidromic.

xiv) minm no. of insertion in string to make it a palidromic.

Longest Common Subsequence / LCS

I/P

x :- a b c d g h — n → length of string

y :- a b e d f h r — m

LCS :- a b d h

O/P :- 4

in Substring character must be continuous.

Recursion :-

```
int LCsequence(string x, string y)
    if (n==0 || m==0)
        return 0;

int LCS(string X, string Y, int n, int m)
{
    if (n==0 || m==0)
        return 0;
    if (X[n-1]==Y[m-1])
        return 1+LCS(X,Y,n-1,m-1);
    else
        return max(LCS(X,Y,n-1,m), LCS(X,Y,n,m-1));
}
```

memoization :-

```
t[n+1][m+1]
int t[n+1][m+1];
memset(t, -1, sizeof(t));  ⟶ in main
int Lcs(string x, string y, int n, int m)
{
    if (n==0 || m==0)
        return 0;
    if (t[n][m]!=-1)
        return t[n][m];
    if (x[n-1]==y[m-1])
        return t[n][m] = 1+ Lcs(x,y,n-1,m-1);
    else
        return t[n][m] = max(Lcs(x,y,n,m-1), Lcs(x,y,n-1,m));
}
```

Top down :-

```
for(int i=0; i<n+1; i++)
    t[i][0] = 0;
for(int j=1; j<m+1; j++)
    t[0][j] = 0;
for(int i=1; i<n+1; i++)
for(int j=1; j<m+1; j++)
{   if(x[i-1]==y[j-1])
        t[i][j] = 1+ t[i-1][j-1];
    else
        t[i][j] = max(t[i-1][j], t[i][j-1]);
} return t[n][m];
```

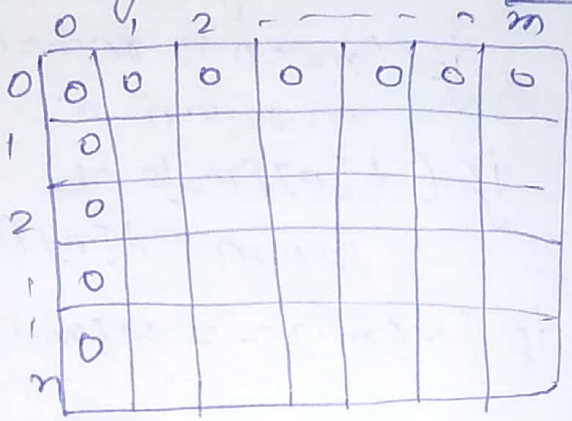
Longest Common Substring

I/P x: a b c d e → n
y: a b f c e → m

Continious Subset

O/P :- 2 (ab) ← Length of Longest Common Substring

t[n+1][m+1]




```
for(int i=1; i<n+1; i++)
  for(int j=1; j<m+1; j++)
    { if(x[i-1]==y[j-1])
        t[i][j]=1+t[i-1][j-1];
      else
        t[i][j]=0;
    }
  }
int max=INT_MIN;
for(int i=1; i<n+1; i++)
  for(int j=1; j<m+1; j++)
    if(max<t[i][j])
      max=t[i][j];

return max;
}
```

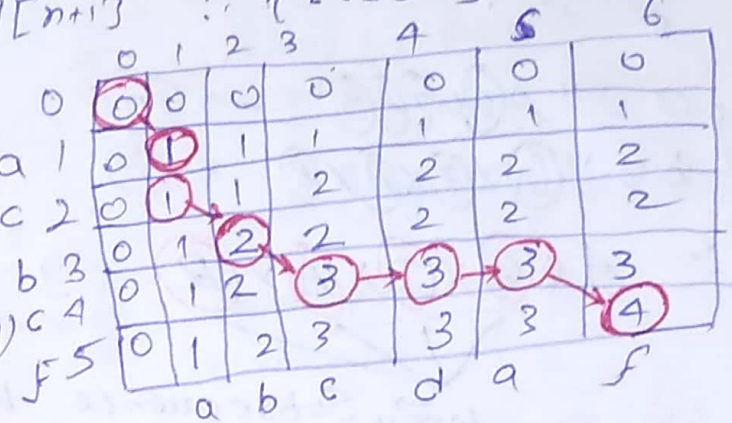
Print LCS (subsequence) b/w two strings

I/P x: a a b c f → m = 5

y: a b c d a f → n = 6

O/P :- abcf

t[m+1][n+1] :: t[6][7]




```
int i = m, j = n; String s;
while (t[i][j])
{ if (x[i-1] == y[j-1])
  { s.push_back(a[i-1]);
    i--;
    j--;
  }
  else
  { if (t[i-1][j] > t[i][j-1])
    { i--;
    }
    else
      j--;
  }
}
reverse(s.begin(), s.end());
return s;
```

Shortest Common Supersequence length

i/p a = "geek" → (m) we have to merge a and b
b = "eke" → (n) so that it contain both string.

o/p "geeke".

I/p { a = "AGGTAB" → (m)
b = "GXTXAYB" → (n)

m n LCS
↑ ↑ ↑
o/p :- AGGXTXAYB = 9 = 6+7-4 = 9

Subsequence written only once, in output
longest

So, length of shortest common supersequence
= m+n - longest subsequence length

Code :-
return (m+n) - LCS(a, b, m, n);

Minimum no. of insertion and deletion to make string a to string b

I/p :- a : heap → m = 4
b : pea → n = 3

o/p :- p e a = 3 = insertion :- n - LCS
deletion :- m - LCS
= 4+3-2(2)
= 3

Return m+n - 2 * LCS(a, b, m, n);

Longest Palidromic Subsequence (LPS)

I/P s: agbcba s' = abcbga

O/P 5

(a) g (b) (c) (b) (a)

String s';
s' = reverse (s.begin(), s.end());

s = string (s.rbegin(), s.rend());

return Lcs (s, s', s.length(), s'.length());

↓
Longest Common Subsequence.

Minm no of deltion in string to make its Palidromic.

I/P s = "agbcba" → n = 6

O/P :- 6 - 5 = 1.

s' = "abcbga"

Lcs = a b c b a → 5

Output :-

return n - LPS(s)

↓
Longest palidromic Subsequence.

Print Shortest Common Supersequence

I/P a: a c b c f → m=(5)
b: a b c d a f → n=6

O/P :- "a c b c d a f" , worst:- "a c b c f a b c d a f"
LCS :- a b c f

Similar Question print LCS (longest Common Subsequence)

	ϕ	a	b	c	d	a	f
ϕ	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

LCS t[6][7]

```
int i=m;
int j=n;
String s.
while (i>0 && j>0)
{ if (a[i-1]==b[j-1])
    { s.push_back(a[i-1]);
      i--; j--;
    }
  else { if (t[i][j-1] > t[i-1][j])
         { s.push_back(b[j-1]);
           j--;
         }
       else { s.push_back(a[i-1]);
              i--;
            }
       }
}
```

```
}
}
while ( i>0)
  { s.push_back ( a[i-1]);
      i--;
   }
while (j>0)
  { s.push_back (b[j-1]);
    j--;
  }
return string (s.begin(), s.end())
}
```

Longest Repeating Subsequence length

Str = "AABEBCDD" → (m)

o/p :- 3 (ABD)

(A) (A) (B) E (B) C (D) (D)

ABD repeat 2 times.

Str = "AABEBCDD" → (m)

Str = "AABEBCDD" → (n)

```
for (int i=1; i<m+1; i++)
  for(int j=1; j<m+1; j++)
    { if ( str[i-1] == str[j-1] && i!=j)
        { t[i][j] = 1+ t[i-1][j-1];
        }
      else
        { t[i][j] = max ( t[i-1][j], t[i][j-1]);
        }
    }
    return (t[m][m]);
```

Sequence Pattern matching

I/P a = "AXY"
b = "ADXCPY"

O/P - T/F → boolean.

(A) D (X) C P (Y) → true

lcs :- "AXY"

if length of lcs == a.length()
then return true;
else return false;

Minimum no. of ~~deletion~~ insertion in string to make it palindromic

I/P :- S = "(a)c(b)(c)(b)d(a)"

O/P :- 2.

(a)d(b)(c)(b)c(a)

longest palidromic subsequence
:- "abcba"

min^m of deletion = S.length() - length of LPS

min^m no. of insertion = min^m of deletion
= S.length() - length of LPS.

(a) c d (b) (c) (b) d c (a)
↑ insertion ↑ insertion

2nd part

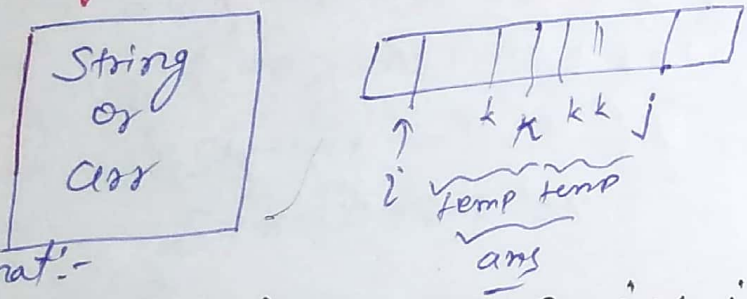
Matrix Chain multiplication

↳ Nuttanix

1. mcm
2. Printing mcm
3. Evaluate Expr to True / Boolean parenthesisation
4. min / max value of an Expr
5. Palindromic partitioning
6. Scramble String
7. Egg Dropping problem

mcm

identification :-



30, 50
10, 30

format :-

```
int Solve ( int arr[], int i, int j)
{ if (i>j)
    return 0;
  for ( int k = i; k<j ; k++)
  { // Calculate temp answer
    temp_ans = Solve (arr, i, k) + Solve (arr, k+1, j)
    ans ← fun (temp_ans);
  }
  return ans;
}
```

mcm

arr[] = { 40 20 30 10 30 }

A_1 A_2 A_3 A_4

40×20 20×30 30×10 10×30

$A_1A_2A_3A_4$ Calculate this by minm cost (less multiplication)

$A_{a\times b} B_{b\times c} = C_{a\times c}$

↓

Cost $(a\times c)\times b = acb$

multiply such that cost is minm.

A → 10 * 30

B → 30 * 5

C → 5 * 60

ABC

↓

(AB)C

Cost = 10×30×5 + 10×5×60

= 4500

A(BC)

Cost = 10×30×60 + 30×5×60

= 22000

Ex arr[]: 40 20 30 10 30

i ↑ j

A_1 → 40×20

A_2 → 20×30

A_3 → 30×10

A_4 → 10×30

A_i = arr[i-1] × arr[i]

= 40 × 20

```
int Solve ( int arr[], int i, int j)
{
    if (i >= j)
        return 0;
    for(int k = i; k <= j-1; k++)
    {
        int temp1
```

```
int min = INT_MAX;
for (int k = i; k <= j-1; k++)
{ int temp = Solve(arr, i, k) + Solve(arr, k+1, j)
                + arr[i-1] * arr[k] * arr[j];
  if (min > temp)
    min = temp;
}
return min;
}
```

memoized code bottom up

```
int t[Size+1][Size+1];
memset(t, -1, sizeof(t)); <- in main;
Solve(arr, 1, size-1) <- in main,
int solve(int arr[], int i, int j)
{ if (i == j)
    return 0;
  if (t[i][j] != -1)
    return t[i][j];
  int min = INT_MAX;
for (int k=i; k<j; k++)
  { int temp = Solve(arr, i, k) + Solve(arr, k+1, j)
                + arr[i-1] * arr[k] * arr[j];
    if (min > temp)
      min = temp;
  }
  return t[i][j] = min; }
```

Palidromic partitionaty

min^m no. of partition of string to make resultant substring palidroom.

I/p :- nitin

O/p :- 2


```
int t[size+1][size+1];
memset(t, -1, sizeof(t));
Solve(arr, 0, size-1);
```
→ in main()

```
int Solve(int arr[], int i, int j)
{ if(i>=j)
    return 0;
  if(t[i][j] != -1)
    return t[i][j];
  if(isPalindrom(arr, i, j))
    return 0;
  int min = INT_MAX;
  for(int k = i; k <= j-1; k++)
  { int temp = Solve(arr, i, k) + Solve(arr, k+1, j) + 1;
    if(min > temp)
      min = temp;
  }
  return t[i][j] = min;
}
```

Allocate minimum no. of pages

arr[]: 10 20 30 40 → number of page in each book

K: 2 → no. of students.

→ Array sorted not required.

```
#include <bits/stdc++.h>
using namespace std;
bool isValid(int arr[], int n, int k, int ans)
{ int student=1; int sum=0;
  for(int i=0; i<n; i++)
  { sum += arr[i];
    if(sum>ans)
    { student++;
      sum=arr[i];
    }
    if(student>k)
      return false;
  }
  return true;
}
int solve(int arr[], int n, int k, int m, int sum)
{ int start=m;
  int e=sum; int ans=INT_MAX;
  while(start<=e)
  { int mid=start+(e-start)/2;
```

```
    if (isValid(arr, n, k, mid))
    { ans = min(ans, mid);
      e = mid - 1;
    }
    else
    { start = mid + 1;
    }
  }
return ans;
}
int main()
{ int k, n; cin >> k >> n;
  int arr[n]; int m = INT_MIN, sum = 0;
  for (int i = 0; i < n; i++)
    { cin >> arr[i];
      m = max(m, arr[i]);
      sum += arr[i]; }
  cout << solve(arr, n, k, m, sum) << endl;
}
return 0;
}
```

longest increasing subsequence

3 4 -1 0 6 2 3 = 4

2 5 1 8 3 = 3

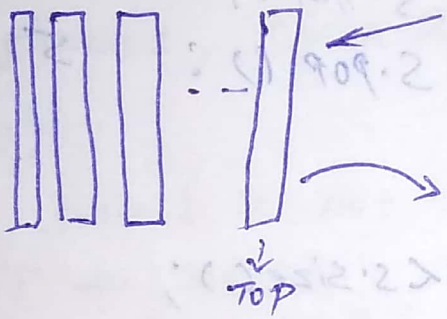
```
int t[n+1]; int j; t[0]=0;
for(int i=0; i<n; i++)
    t[i+1]=1;
for(int i=1; i<n; i++)
{   j=0;
    for(; j<i; j++)
    {  if(arr[i] >= arr[j])
           t[i+1] = max(t[i+1], t[j+1]+1);
    }
}
return max_element(t+1, t+n+1);
}
```

```
for(int i=0; i< colour.size(); i++)
    cout << colour[i] << "\n";
}
```

Stack Data Structure

linear data structure follows particular order in which operation are performed. Order may be LIFO (Last in first out) or FILO (First in last out)

Stack insertion and deletion happen on same side

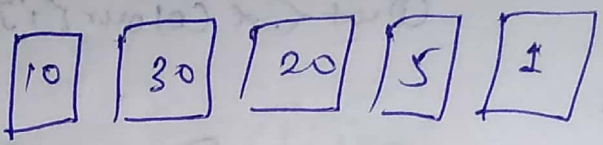


Last in, First out

- empty() :— Returns whether stack is empty — time complexity O(1)
- size() :- Returns size of stacks — time complexity O(1)
- top() :- Returns ~~reference to~~ top most element of stack — Time complexity O(1)
- push(g) :- Add element 'g' at top of stack. time complexity : O(1)
- pop() — deletes top most element of stack time Complexity : O(1)

```
#include <bits/stdc++.h>
using namespace std;
Void showstack (stack<int> s)
{ while (!s.empty())
  { cout << ' ' << s.top();
    s.pop();
  }
```

```
    cout << "\n";
}
int main ()
{ stack <int> s;
  s.push(10);
  s.push(30);
  s.push(20);
  s.push(5);
  s.push(1);
  cout << "the stack is:";
  showstack(s);
  cout << "\n s.size() : " << s.size();
  cout << "\n s.top() : " << s.top();
  cout << "\n s.pop() : ";
  s.pop();
  showstack(s);
  return 0;
}
```



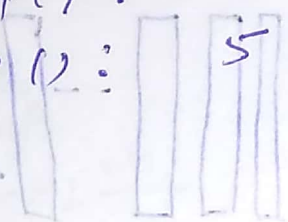
Output:-

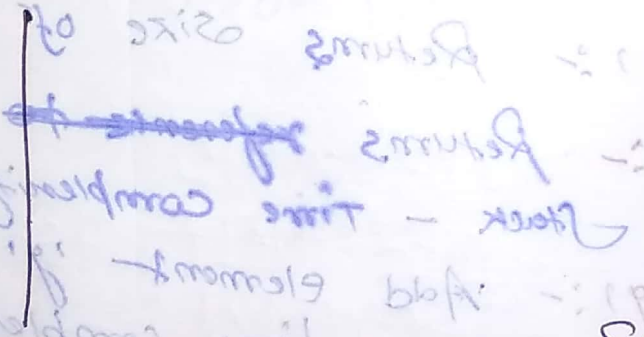
The stack is : 1 5 20 30 10

S.size() : 5

S.top() : 1

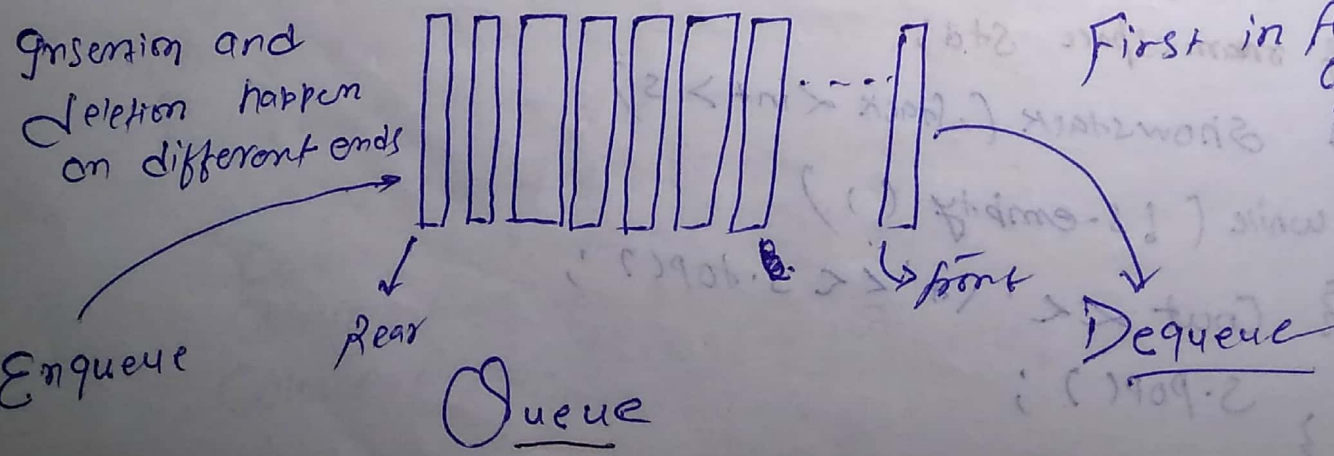
S.pop() :





Implementation of Queue using Stacks

Given, Stacks data structures and we have to implement Queue using pop() and push() operation.



Queue can be implemented using two stacks. Stack 1 and stack 2. q (queue) can be implemented into two ways :-

Method-1 (By making enqueue operation costly) :-

This method make sure that oldest entered element is at always top of stack 1, so that dequeue operation just pops from stack 1. To put element at top of stack 1, stack 2 is used.

enQueue (q, x):

→ while stack 1 is not empty, push everything from stack 1 to stack 2.

→ push x to stack 1 (assuming size of stack is unlimited)

→ push everything back to stack 1

Here time complexity O(n)

dequeue (q):

→ if stack 1 is empty then error

→ pop an element from stack 1 and return it.

time complexity O(1)

```
#include <bits/stdc++.h>
using namespace std;
Stack Queue
{ Stack<int> s1, s2;
  void enQueue (int x)
  { while (!s1.empty())
    { s2.push(s1.top());
      s1.pop();
```

```
    S1.push(x);
    while (!S2.empty())
    { S1.push(S2.top());
      S2.pop();
    }
}
int deQueue()
{ if(S1.empty())
  { cout << "Q is Empty";
    exit(0);
  }
  int x = S1.top();
  S1.pop();
  return x;
}
};
int main()
{ Queue q;
  q.enQueue(1);
  q.enQueue(2);
  q.enQueue(3);
  cout << q.deQueue() << '\n';
  cout << q.deQueue() << '\n';
  cout << q.deQueue() << '\n';
  return 0;
}
```

Time Complexity :-

Push operation : O(N)

Pop operation : O(1)

Auxiliary space : O(N)

use of stack for storing values.

Output :- 1
2
3

Stack introduction and Identification

7 Question :-

1> Nearest	Greater	to left	→ same concept
2> Nearest	"	to Right	(Next largest element)
3> "	Smaller	" Left	(Nearest Smaller ")
4> "	"	to Right	

5> stock Spain Problem
6> maximum area of Histogram

7> max Area of Rectangle in binary matrix

6 question :-

8> Rain water trapping
9>. Implementing a min Stack
10>. Implementing Stack using Heap
11>. The Celebrity problem
12>. Longest valid parenthesis
13>. Iterative ToH (Tower of Height) implementation

Identification :- i> array
of stack
question ii> $O(n^2)$ ← brute force.

Second loop → I loop dependent on i

then we always use Stack in this question for better Solution.

Next largest element / Nearest greater to Right

arr[] : 1 3 2 4

o/p :- 3 4 4 -1

arr[] : 1 3 0 0 1 2 4

o/p :- 3 4 1 1 2 4 -1

```
for(int i=0; i<n; i++)
  for(int j=i+1; j<n; j++)
  {
  }
```

-> So this can be implemented using stacks.


```
Vector<int> v;
Stack<int> s;
for(int i=n-1; i>=0; i--)
{ if( s.size()==0)
      v.push_back(-1);
  Else if ( s.top() > arr(i) )
    { v.push_back(s.top());
    }
  Else if ( s.top() <= arr(i) )
    { while( s.size()>0 && s.top() <= arr(i) )
        { s.pop();
        }
      if ( s.size()==0)
          v.push_back(-1);
      else
          v.push_back ( s.top());
    }
```

S.push(arr[i]);

}

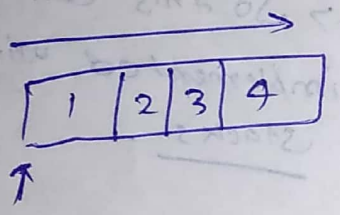
Reverse vector v;

Nearest greater to left

arr[]: 1 3 2 4

↑ ↑ ↑ ↑

-1 -1 3 -1



Brute:- ✓ dependent on i.

```
for(int i=0; i<n; i++)
    for(int j=i-1; j>=0; j--)
    {

    }
```

Change → i> left to Right

i> no need for changes.

```
vector<int> v;
stack<int> s;
for(int i=0; i<size; i++)
{ if( s.size() == 0)
      v.push_back(-1);
  else if( s.top() > arr[i])
      v.push_back( s.top());
  else if( s.size()>0 && s.top()<=arr[i])
  { while( s.size()>0 && s.top()<=arr[i])
    { s.pop();
    }
    if( s.size()==0)
      v.push_back(-1);
    else
      v.push_back( s.top());
  }
  s.push(arr[i]);
}
```

Nearest Smaller to left / Nearest Smaller element

arr[]:- 4 5 2 10 8

-1 4 -1 2 2

Stack vector

```
for(int i=0; i<size; i++)
{   if(S.size()==0)
        v.push_back(-1);
    Else if(S.size()>0 && S.top()<arr[i])
        v.push_back(S.top());
    Else if(S.size()>0 && S.top()>=arr[i])
    {   while(S.size()>0 && S.top>=arr[i])
        {   S.pop();
        }
        if(S.size()==0)
            v.push_back(-1);
        Else
            v.push_back(S.top());
    }
    S.push(arr[i]);
}
```

Nearest Smaller to Right

Traverse from right

Reverse ✓

arr[] : 4 5 2 10 8

Stack ←

Normal →

```
for(int i= size-1; i>=0; i--)
{   if(S.size()==0)
        v.push_back(-1);
```

```
Else if( S.top() <= arr[i])
        V.Push_back(S.top());
Else
{   while( S.size()>0 && S.top()>=arr[i])
        {S.pop()
        }
    if(S.size()==0)
      V.push_back(-1);
    Else
      V.push_back(S.top());
    }
  S.push(arr[i]);
}
reverse(V.begin(), V.end());
}
```

Stock Span Problem

arr: 100 80 60 70 60 75 85

Day 1 1 1 2 4 6

Consecutive smaller or equal elements → before i

↓ Span value of day i

left to Right

We have to find Nearest greater to left

100 80 60 70 60 75 85

80 70 75

Nearest greater to left

100	80	60	70	60	75	85
0	1	2	3	4	5	6
-1	100	80	80	70	80	100

6-0 = 6

4-3 = 1

5-1 = 4

⇒ arr :-

0	1	2	3	4	5	6
100	80	60	70	60	75	85

V :-

-1	0	1	1	3	1	0

← Nearest greater to left

1 1 1 2 1 4 6

Stack <int> s ⟶ stack < pair < int, int > > s;

NGE, index

vector <int> v;

for (int i=0; i < size; i++)

{ if (s.size() == 0)

v.push_back(-1);

else if (s.top().first > arr[i])

v.push_back (s.top().second);

else

{ while (s.size() > 0 && s.top().first <= arr[i])

{ s.pop(); }

if (s.size() == 0)

v.push_back (-1);

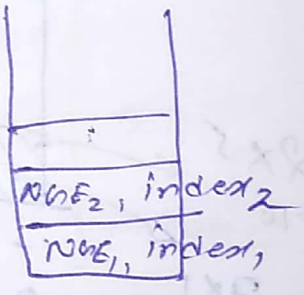
else

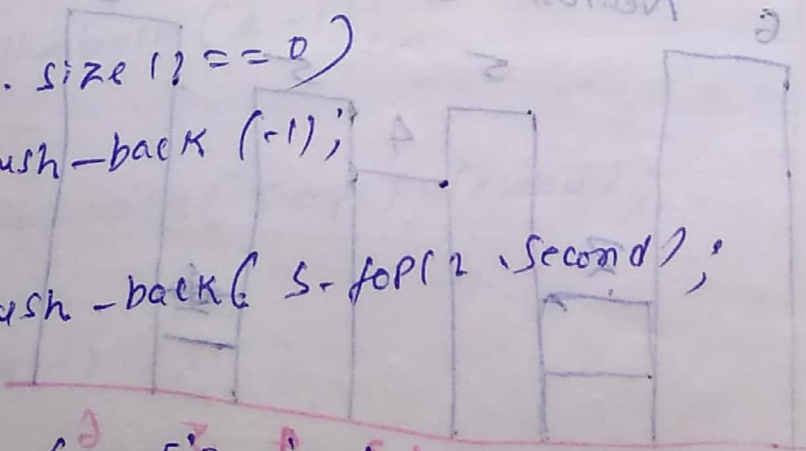
v.push_back (s.top().second);

}

s.push (arr[i], i);

}





```
for(int i=0; i<size(); i++)
{ v[i] = 1 - v[i];
}
return v;
}
```

Sum up :-

Stock Span Problem → Consecutive Smaller or equal to left → NGL index → Next Greater Left → i - NGL index.

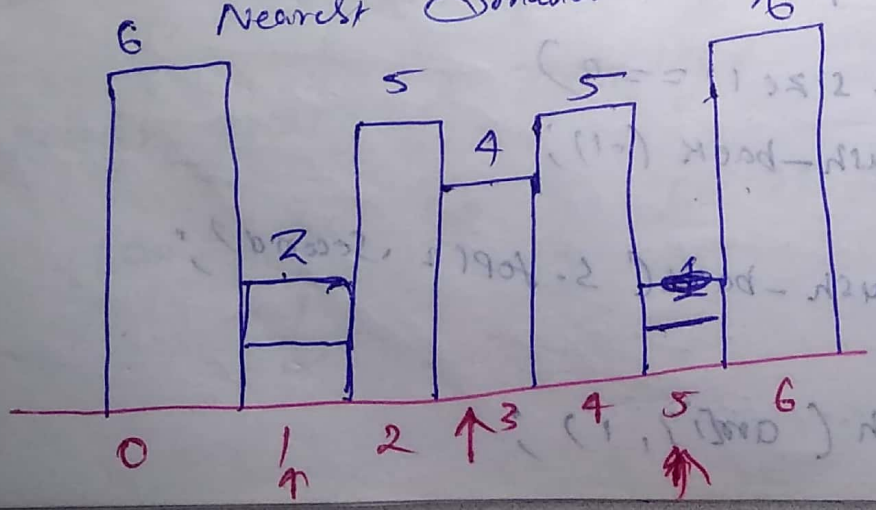
maximum area of histogram

arr[] : 6 2 5 4 5 1 6 → Height

For any building expend upto greater or equal to height on both side.

↓

Nearest smaller to left index
Nearest Smaller to right index



	0	1	2	3	4	5	6	
arr:	6	2	5	4	5	1	6	
left:	-1	-1	1	1	3	-1	5	← NSL index
Right:	1	5	3	5	5	7	7	← NSR index

Answer :- **Right - left - 1 = width[i]**

Area[i] = arr[i] (height) × width[i]

```
Stack < pair < int, int >> S1, S2
Vector < int > left
vector < int > Right
vector < int > width
For (int i=0 ; i<size; i++)
{ if ( S.size() ==0 )
    left.push_back(-1);
  Else if ( S.top().first < arr[i])
    left.push_back(S.top().second);
```

→ NSL

```
  Else
  { while ( S.top().first >= arr[i] && S.size() >0)
      { S.pop();
      }
    if (S.size() ==0)
      left.push_back(-1);
    Else
      left.push_back(S.top().second);
  }
  S.push(arr[i], i);
}
```

```
for(int i=size-1 ; i>=0 ; i--)
{ if(s.size()==0)
    right.push_back(7);
  Else if (s.top().first < arr[i])
    right.push_back(s.top().second);
  Else
    { while(s.size()>0 && s.top().first >= arr[i])
      { s.pop(); }
      if(s.size()==0)
        right.push_back(size);
      Else
        right.push_back(s.top().second);
    }
  s.push(arr[i], i);
}
reverse(right.begin(), right.end());
}
for(i=0; i<size; i++)
  width[i] = Right[i] - Left[i] - 1;
int max = width[0] * arr[0];
for(int i=1; i<size; i++)
{ if(max < width[i] * arr[i])
    max = width[i] * arr[i];
}
```

NSR

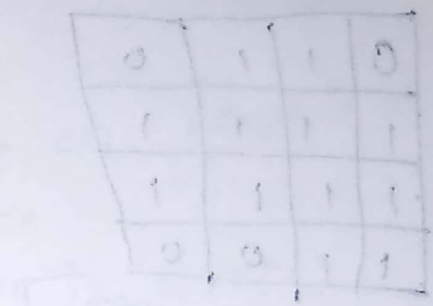
area

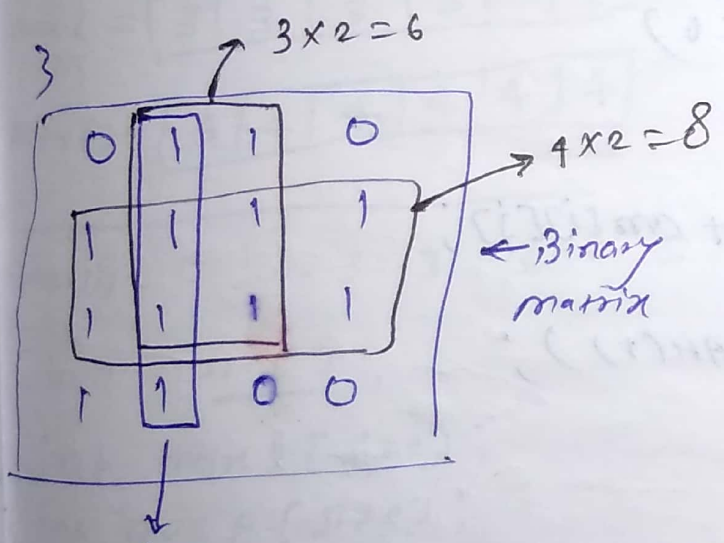
return max ;

7. Maximum area of Rectangle in binary matrix

Maximum area of histogram code :-

```
MAH ( int arr[], int size )
{ right[] → NSR (arr, size)
  left[] → NSL (arr, size)
  width[i] = Right[i] - Left[i] - 1
  area[i] = arr[i] * width[i]
  return max area[i]
```



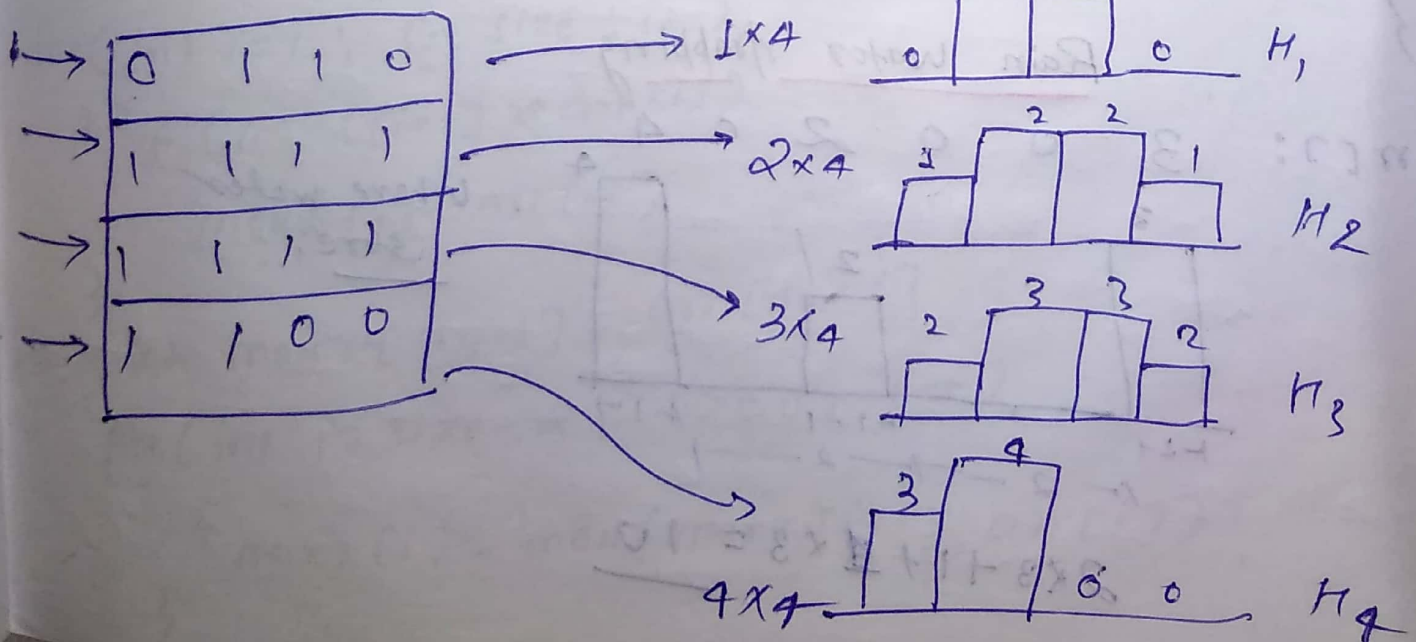


difference :-

2D array

Binary number.

Compress 2D array to 1D array



ans = max { max(H1) → 2, max(H2) → 4, max(H3) → 8, max(H4) → 6 } → max = 8

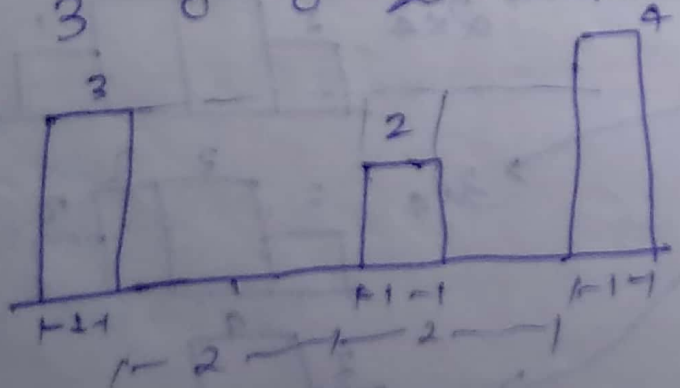
	j=0	1	2	3
i=0	0	1	1	0
1	1	1	1	1
2	1	1	1	1
3	1	1	0	0

n x m
arr[][]

```
Vector<int> v;
for(int j=0; j<m; j++)
    v.push_back(arr[0][j]);
int max = mah(v);
for(int i=1; i<n; i++)
{ for(int j=0; j<m; j++)
    { if(arr[i][j]==0)
        v[j]=0;
      else
        v[j] = v[j] + arr[i][j];
    }
  max = max(max, mAH(v));
}
return max;
}
```

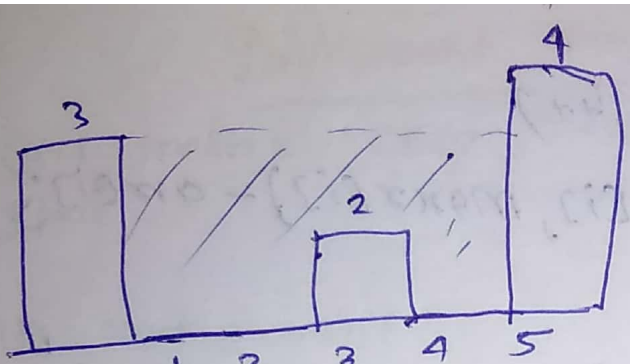
Rain water trapping

arr[]: 3 0 0 2 0 4



where water store.

2×3 + 1 + 2×3 = 10



Σ water on above each building = 0 + 3 + 3 + 1 + 3 = 10

water on any i^{th} buiding dependent on Largest building on Left and Right side:

water[i] = min(left side maxm, right side maxm) − arr[i]

3 0 0 2 0 4

maxl = 3 3 3 3 3 4

maxr[] = 4 4 4 4 4 4

water[i] = 0 3 3 1 3 0 = 10

0 1 0 2 1 0 1 3 2 1

0 1 1 2 2 2 2 3 3 3

3 3 3 3 3 3 3 3 2 1

Given :- arr[],

```
int maxl[size];
int maxR[size];
maxl[0] = 3;
for(int i=1 ; i< size ; i++)
  { if (maxl[i-1] < arr[i])
      maxl[i] = arr[i];
  }
maxr[size-1] = arr[size-1]
for(int i= size-2; i>=0; i--)
  { maxr[i] = max(maxr[i+1], arr[i]);
  }
```

```
int water[size];
for(int i=0 ; i<size ; i++)
    water[i]= min(maxl[i], maxr[i])-arr[i];

int sum=0;
for(int i=0; i<size ; i++)
    sum += water[i];
```

Return sum;

Implement minstack with extra space - O(n)

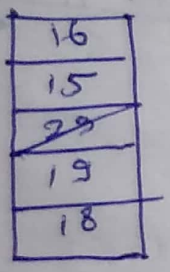
18 19 29 15 16

Find minm element in stack.

Stack (S)

Supporting Stack (SS)

top of it always contain minimum value to give min output in O(1) time.



Stack → push
→ pop

```
stack<int> s;
stack<int> ss;

void push(int a)
{
  s.push(a);
  if(ss.size()==0)
    ss.push(a);
  else if(a<=ss.top())
    ss.push(a);
  return;
}
```

```
int pop()
{ if(s.size()==0)
    return -1;
  if(s.top()==ss.top())
    ss.pop();
  s.pop();
  return s.top();
}
```

```
int getmain()
{
  if(ss.size()==0)
    return -1;
  return ss.top();
}
```

Implement minstack without using extra O(1) space

O(1) means any number of variable.

Constant

(2*x-min)
(2*min-y)

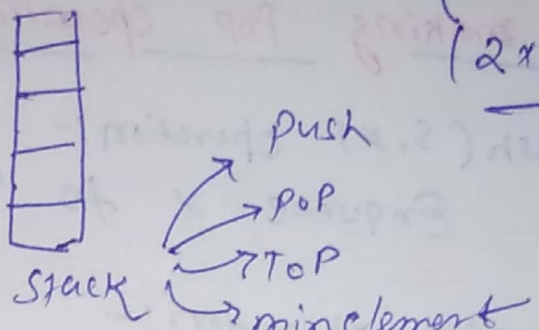
int minElement;

~~minElement = 0;~~

int getMin()
{ if (s.size() == 0)
    return -1;
  return minElement;
}

Void push(x)
{ if (s.size() == 0)
    minElement = x;
  if (x >= minElement)
    s.push(x);
  Else
  { s.push(2x - minElement);
    minElement = x;
  }
}

int top()
{ if (s.size() == 0)
    return -1;
  else { if (s.top() >= minElement)
           return s.top()
         else if (s.top() < min Element)
           return minElement;
  } }



void pop()
{ if (s.size() == 0)
    return;
  Else if (s.top() >= min Element)
    s.pop();
  Else if (s.top() < min element)
  { minE = 2*minE - s.top()
    s.pop();
  }
}

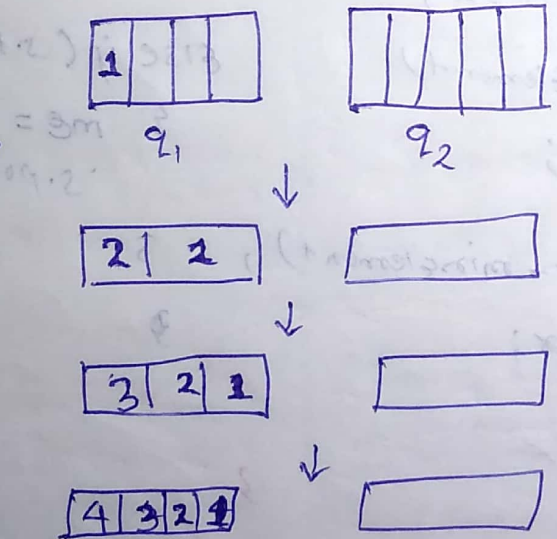
Implementation of Stack using Queue

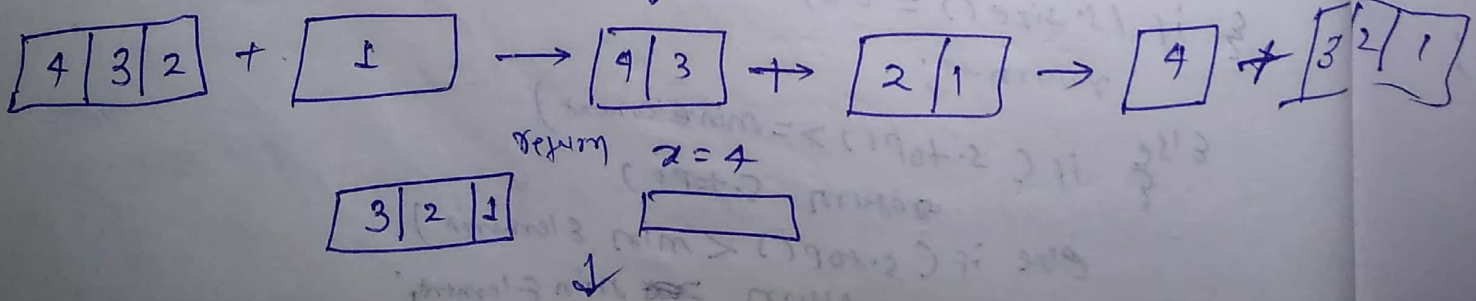
1.> By making Pop operation Costly :-

1. Push(S, x) operation:-
   Enqueue x to q_1 (assuming size of q_1 is unlimited)

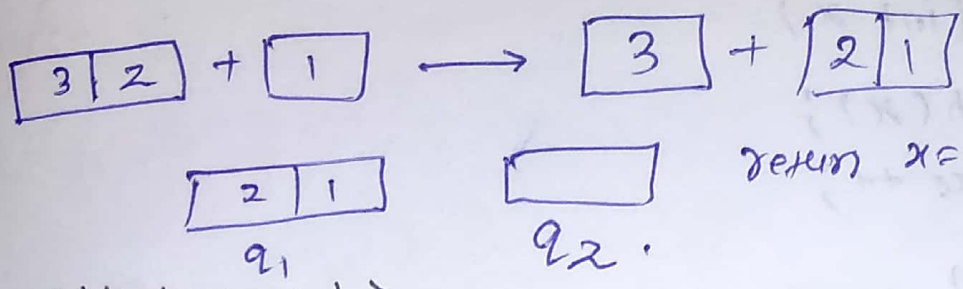
2. Pop(S) operation:-
   → One by one dequeue everything except the last element from q_1 and enqueue to q_2.
   → Dequeue last item of q_1, the dequeued item is result, store it.
   → Swap names of q_1 and q_2
   → Return item stored in step 2.

S.push(1)
S.push(2)
S.push(3)
S.push(4)
S.pop()
S.pop()





return x = 4




```
#include<bits/stdc++.h>
using namespace std;
class Stack
{
  queue<int> q1, q2;
  int curr_size;
public:
  Stack()
  { curr_size = 0;
  }
  void pop()
  { if (q1.empty())
      return;
    while (q1.push != 1)
    { q2.push(q1.front());
      q1.pop();
    }
    q1.pop();
    curr_size--;
    queue<int> q = q1;
    q1 = q2;
    q2 = q;
}
```

```
Void push(int x)
{ q1.push(x);
  curr_size++;
}
int top()
{ if (q1.empty())
    return -1;
  while (q1.size() != 1)
  { q2.push(q1.front());
    q1.pop();
  }
  int temp = q1.front();
  q1.pop();
  q2.push(temp);
  queue<int> q = q1;
  q1 = q2;
  q2 = q;
  return temp;
}
int size()
{ return curr_size;
}
};
```

```
int main()
{ Stack S;
  S.push(1);
  S.push(2);
  S.push(3);
  S.push(9);
  Cout << S.size() << endl;
  Cout << S.top() << endl;
  S.pop();
  Cout << S.top() << endl;
  S.pop();
  return;
```

Output:-

4
4
3

Implement Stack using single Queue

Push (s, x)
→ Size of q be s
→ Enqueue x to q
→ One by one Dequeue s items from queue and enqueue item.

Pop (s)
→ Dequeue an item from q.

```
#include<bits/stdc++.h>
using namespace std;
Class Stack
{ queue<int> q;
  Public:
     void push(int val);
     Void pop();
     int top();
     bool empty();
};
Void Stack::push(int val)
{ int s=q.size();
  q.push(val);
  for(int i=0; i<s; i++)
  { q.push(q.front()); q.POP();
  }}
```

```
void Stack::pop()
{ if(q.empty())
    return;
  q.pop();
}
int Stack::top()
{ return (q.empty()) ? -1 : q.front();
}
bool stack::empty()
{ return (q.empty());
}
int main()
{ Stack s;
  s.push(10);
  s.push(20);
  cout<<s.top()<<endl;
  s.pop();
  s.push(30);
  s.pop();
  cout<<s.top()<<endl;
  return 0;
}
```

Output:-

20
10

Stack (Infix to Postfix)

Infix expression:- expression of form a op b.
when operator is in-between every pair of operands.

Postfix expression:- expression of form a b op
when operator is followed by every pair of operands.

a+b*c+d ⟶ postfix a b c * + d +
infix a+b*c+d

```
#include<bits/stdc++.h>
using namespace std;
int prec(char c)
{ if(c=='^')
   return 3;
 else if(c=='*'||c=='/')
   return 2;
 else if(c=='+'||c=='-')
   return 1;
 else
   return -1;
}
void infixToPostfix(string s)
{ std::stack<char> st;
   st.push('N');
   int l=s.length();
   string ns;
```

```
For(int i=0; i<l; i++)
{ if((s[i]>='a' && s[i]<'z') || (s[i]>='A' && s[i]<='Z'))
    ns+=s[i];
  else if (s[i]=='(')
     St.push('(');
  else if (s[i]==')')
  { while (St.top()!='N' && st.top()!='(')
      { char c = St.top();
        St.pop();
        ns+=c;
      }
    if(s.top()=='(')
       St.pop();
    }
  else
    { while (st.top()!='N' && prec(s[i])<=prec(St.top()))
      { char c = St.pop();
        St.pop();
        ns+=c;
      }
      St.Push(s[i]);
    }
}
while(St.pop()!='N')
{ char c = St.top();
  St.pop();   ns+=c;
}
```

```
    cout << ns << endl;
    }
int main()
{ string exp = "a+b*(c^d-e)^(f+g*h)-i";
  infixToPostfix(exp);
  return 0;
}
```

Output :- abcd^e-fgh*+^*+i-

Prefix to Infix Conversion

Prefix :- if operator appears in expression before operands.

Example :- *+AB-CD

= (A+B)*(C-D) infix

Algorithm :-

→ Read prefix expression in reverse order (from right to left)

→ if symbol is an operand, then push it onto stack

→ if symbol is an operator, then pop two consecutive operands from top of stack. Create a string by concatenating the two operands and operator b/w them

String = (operand 1 + Operator + operand 2)

Push resulting string into stack.

→ Repeat above step until end of prefix expression.

```
String PreToInfix ( String Pre_exp)
{ Stack <string> s;
  int length = Pre_exp.size();
for (int i= length-1; i>=0; i--)
{ if ( pre_exp[i] = '+' || pre_exp[i]= '-' ||
       Pre-exp[i] = '*' || Pre_exp[i]= '/')

  { string op1 = s.top(); s.pop();
    ||     op2 = s.top();  s.pop();
    String temp = "(" + op1 + Pre_exp[i] + op2 + ")";
    s.push (temp);
  }
 else
  { s.push ( String (1, Pre_exp[i]));
  }
}
return s.top();
}
```

Prefix to Postfix

Algorithm :-

→ read prefix in reverse order. (Right to left)
→ if symbol is operand, push it onto stack
→ if symbol is Operator, pop two operands from stack create a string

string = operand1 + operand2 + operator

Push it back to stack

→ Repeat until end of prefix expression.

```
String PreToPost ( String Pre_exp)
{ stack <string> S;
int length= Pre_exp.size();
for(int i= length-1; i>=0; i--)
{ if (isoperator.(Pre_exp[i]))
  { String op1 = S.top(); S.pop();
    String op2 = S.top(); S.pop();
  String temp= op1 + op2 + Pre_exp[i];
  S.push(temp);
  }
else { S.push ( String (', Pre_exp[i]));
  }
} return S.top();
}
```

Example postfix :- ABC/-AK/L-*

Prefix :- *-A/BC-/AKL";

Postfix to Prefix

- Read postfix expression from left to right.
- → if symbol is an operand, push it onto the stack
- → if symbol is an operator, pop two elements from stack, create a string by concatenating two operands and operator.

String = operator + operand2 + operand1

Push string back to stack.

→ Repeat until end of postfix expression.

Check for balanced Parenthesis in an expression

Example :- Input :- "[()]{}{[()()]()}"

Output :- Balanced

input :- "[(])"

Output :- Not Balanced

Algorithm :-

→ Declare a character stack S.

→ Now traverse expression string

1. if current character is starting bracket ('{', '(', '[') then push it to stack.
2. if current character is closing bracket (')', '}', ']') the pop from stack and if popped character is matching starting character then fine else else not balanced.

→ After complete traversal, if stack is not empty means not balanced else balanced.

```
bool areBalanced ( string expr)
{  stack < char > s;
   Char x;
   for ( int i=0 ; i< expr.length() ; i++)
   { if ( exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
        { s.push ( exp[i]);
          Continue;
        }
   else if ( s.empty ())
        return false;
```

```
Switch ( exp[i] )
{ case ')':
        x = S.top();
        S.pop();
        if ( x != '(' )
            return false;
        break;
  Case '}':
        x = S.top();
        S.pop();
        if ( x != '{' )
            return false;
        break;
  Case ']':
        x = S.top();
        S.pop();
        if ( x != '[' )
            return false;
        break;
  }
} return ( S.empty() );
}
```

Length of longest valid substring

input:- ((()

Output:- 2 ()

input :- ()(())

output:- () (2)

input)()())

Output:- 4 :-()()

1) Create an empty stack and push -1 to it. first element of stack is used to provide a base for next valid string.

2) initialize result as 0.

3) If character is '(' push index i to stack

4) Else [if character is ')']

a) Pop an item from stack

b) if stack is not empty .

result = max (result, i - s.top());

c) if stack is empty

push current index for base for next valid substring.

5. Return Result.

```
int findmaxLen (string str)
{ int n = str.length();
  stack <int> st;
  st.push(-1);
  for (int i=0 ; i<n ; i++)
    { if (str[i] == '(')
        st.push(i);
```

```
else
{ st.pop();
  if(!st.empty())
      result = max(result, i-st.top());
  Else
    st.push(i);
}
}
return result;
```

Priority Queue

It is an extension of Queue with following properties:-

i) Every item has a priority associated with it.

ii) An element with high priority is dequeued before element with low priority.

iii) If two elements have same priority, they are served according to their oder in the queue.

Example:- Queue with priority according highest ascii value.

initial Queue = { }

Operation	Return value	Queue content
insert (C)		C
insert (O)		CO
insert (D)		COD
remove max	O	CD
insert (I)		CDI
insert (N)		CDIN
remove max	N	CDI
insert (S)		CDIS

Typical priority queue support following operations:-

i) insert (item, priority) :- insert given item with priority

ii) getHighestPriority ( ) :- Returns highest priority item.

iii) deleteHighestPriority ( ) :- Removes " " ".

Deque

Deque or Double ended Queue is generalised version of Queue data structures that allows insertion and deletion at both ends.

Operation on deque :-

i) insertFront ( ) :- Add items at front of deque.

ii) insertLast ( ) :- " " " last " "

iii) deleteFront ( ) :- delete " " front " "

iv) deleteLast ( ) :- " " " last " ".

v) getFront ( ) :- Gets front item from deque.

vi) getRear ( ) :- Gets rear (last) " " " ".

vii) isEmpty ( ) :- Returns true if it is empty.

viii) isFull ( ) :- Checks whether deque full or not.

Reverse a queue using stack

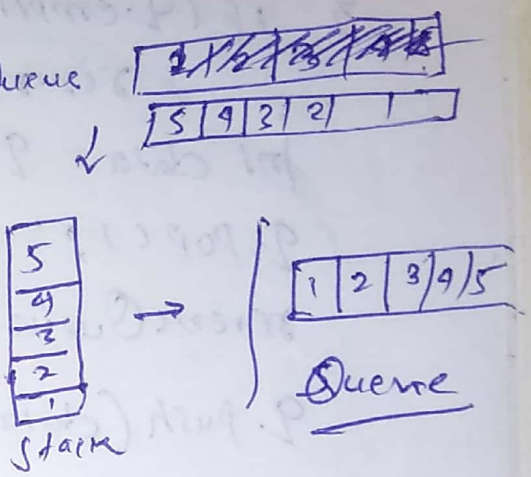
input :- [1, 2, 3, 4, 5]
Queue

Output Queue :- [5, 4, 3, 2, 1]

Approach :- i) Pop element from queue and insert into stack (first element become last)

ii) Pop element from stack and insert into queue.

```
void reverseQueue ( Queue<int> &Queue )
{ Stack <int> Stack;
  while ( Queue.size() != 0)
  { Stack.Push ( Queue.front() );
    Queue.pop();
  }
  while ( Stack.size() != 0)
  { Queue.Push ( Stack.top() );
    Stack.pop();
  }
}
```



Reversing of Queue using Recursion

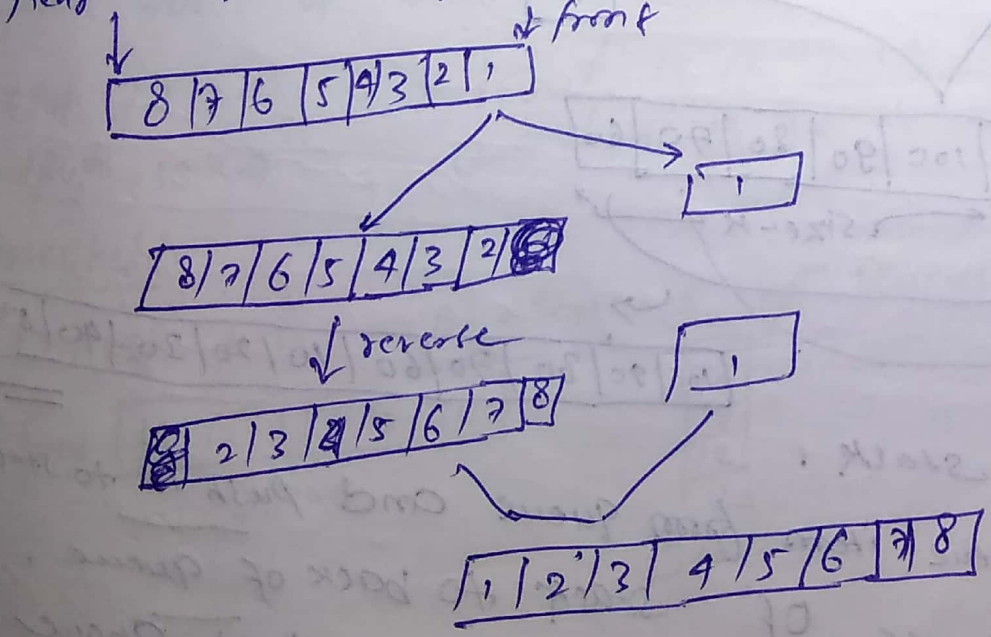
input : Q = | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Output Q = | 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Algorithm :- i) pop element from queue if queue has element else return empty queue

ii) Call reverseQueue function for remaing queue.

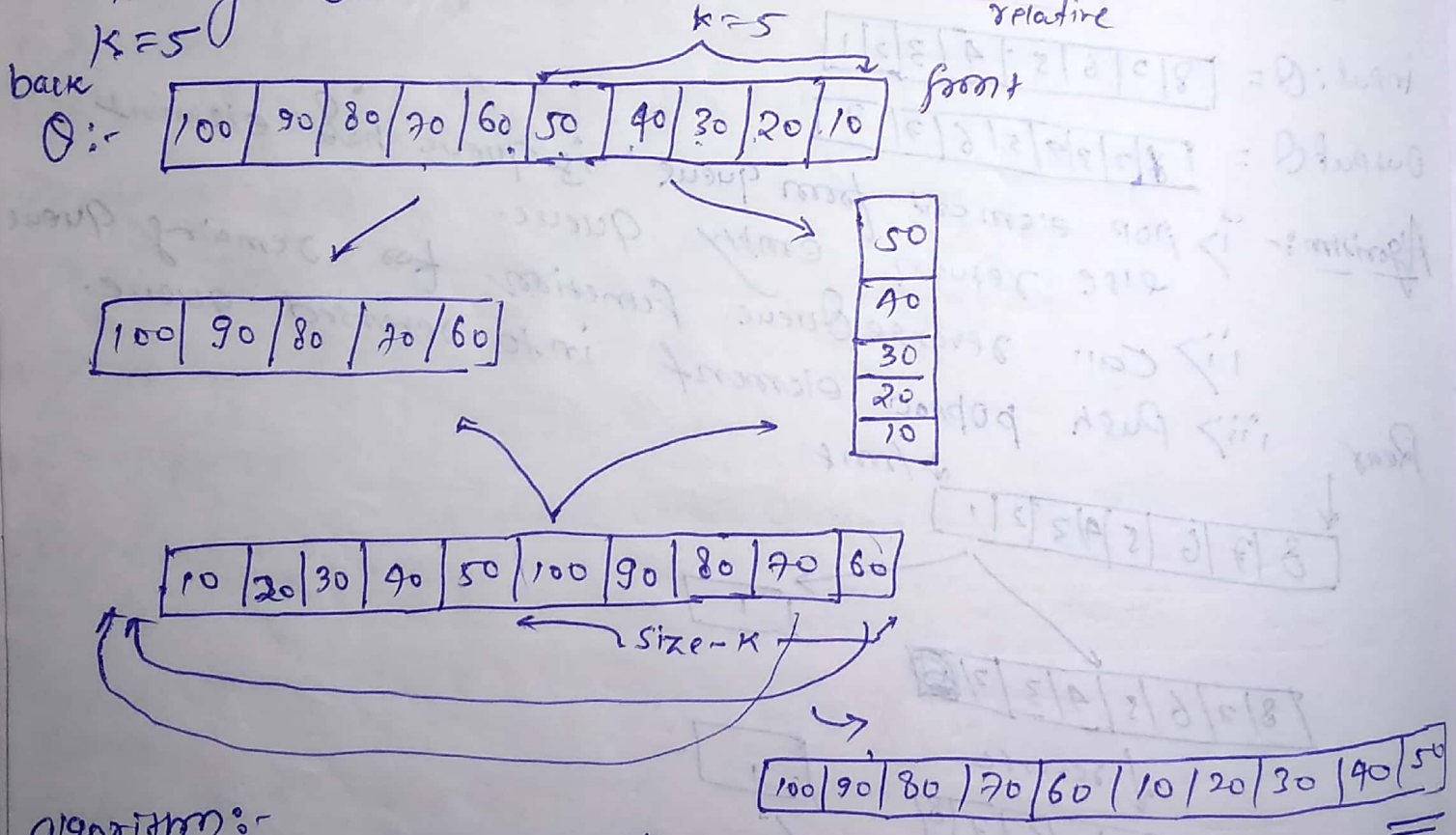
iii) Push popped element into reversed queue



```
void reverseQueue (queue<int> &q)
{ if (q.empty())
      return;
  int data = q.front();
  q.pop();
  reverseQueue (q);
  q.push(data);
}
```

Reversing first K elements of Queue

We have to reverse first K elements of queue.
- leaving other element in same order.



algorithm:-
i) Create empty stack.
ii) One by dequeue item from queue and push it to stack.
iii) Enqueue contents of stack to back of queue.
iv) Dequeue (size-k) elements from front and Enque them one by one to same queue.

```
void reverseQueueFirstKElements ( int K, Queue<int> Queue)
{ if (Queue.empty() == true || K > Queue.size() || K <= 0)
        return;
  Stack <int> S;
  for (int i=0; i<K; i++)
  { S.push (Queue.front());
    Queue.pop();
  } while (!stack.empty())
  { Queue.push (Stack.top());
    Stack.pop();
  }
for (int i=0; i< Queue.size() - K; i++)
  { Queue.push (Queue.front());
    Queue.pop();
  }
}
```

Sorting a Queue without extra space.

With extra space we do simply by dequeue element to array then sort array and enqueue.

```
int minIndex ( queue<int> &q, int sortedIndex )
{ int min_index = -1;
  int min_val = INT_MAX;
  int n = q.size();
  for(int i=0; i<n; i++)
  { int curr = q.front();
    q.pop();
    if ( curr <= min_val && i <= sortedIndex)
    { min_index = i;
      min_val = curr;
    }
    q.push(curr);
  }
  return min_index;
}
Void insertMinToRear (queue<int> &q, int min_index)
{ int min_val;
  int n = q.size();
  for (int i=0; i<n; i++)
  { int curr = q.front();
    q.pop();
    int n = q.size();
    if ( i != min_index)
      q.push(curr);
    else
      { min_val = curr; }
      break
  }
```

```
    q.push(min_val);
  }
void SortQueue( queue<int> &q)
{ for( int i=1 ; i<=q.size();i++)
   { int min_index=MinIndex( q, q.sizer()-i);
     insertMinToRear( q, min_index);
   }
}
```

Interesting method to generate Binary numbers from 1 to n using Queue

input :- n=5

output:- 1, 10, 11, 100, 101

```
Void generatePrintBinary (int n)
{ queue < string > q;
  q.push ("1");
  while (n--)
{ string s1 = q.Front();
    q.pop();
  string s2 = s1;
  q.push ( s1. append("0"));
  q.push ( s2. append("1"));
  }
}
```

Circular tour

There is a circle. There are N petrol pumps on that circle. At this two data sets are given:—
i) Amount of petrol that every petrol pump has.
ii) Distances from petrol pump to next petrol pumps.
Find a starting point where truck can start to get through complete circle without exhausting petrol in between.
Assume:- for 1 litre petrol, truck can go 1 unit of distance.

Input:- 4 → N

4,6	6,5	7,3	4,5

petrol → distances

Output :- 1 (0 indexing)

An efficient approach is use Queue to store current tour. we first enqueue first petrol pump to queue, we keep enqueueing petrol pumps till we either complete tour, or current amount of petrol becomes negative. if Curr_petrol become negative we keep - dequeuing petrol pumps until Curr_petrol become non negative.

Instead we use two pointer start and end that represents rear and front of queue.

```
int PrintTour( PetrolPump arr[], int n)
{  int start=0, end=1;
   int curr_petrol = arr[start].petrol - arr[start].dis
```

```
while (end != start || curr_petrol < 0)
{
    while (curr_petrol < 0)
    {
        curr_petrol -= arr[start].petrol - arr[start].d
        start = (start + 1) % n;
        start = start + 1;
        if (start == n)
            return -1;
    }
    curr_petrol += arr[end].petrol - arr[end].distance;
    end = (end+1) % n;
}
return start;
}
```

Sliding window maximum (maximum of all subarays of size k)

input arr[] :- {1,2,3,1,4,5,2,3,6} k=3

Output :- 3 3 4 5 5 5 6

maximum of 1,2,3 is 3
" " 2,3,1 is 3
" " 3,1,4 is 4
" " 1,4,5 is 5
" " 4,5,2 is 5
" " 5,2,3 is 5
" " 2,3,6 is 6.

1. Create deque to store k elements.
2. Run a loop and insert first k elements in deque. While inserting if element at back of queue is smaller than current element remove all those elements and then insert the elements.
3. Now run a loop from k to end of arr.

4. Print front element of array.
5. Remove element from front of queue if they are out of current window.
6) Insert next element in deque while inserting if element at back of queue is smaller than current element remove all those elements and then insert element.
7. Print maximum element of last window.

```
Void printKmax (int arr[], int n, int k)
{ deque <int> Qi(k);
  for(int i=0; i<k; i++)
    { while((!Qi.empty()) && (arr[i] >= arr[Qi.back()])
        { Qi.pop_back(); }
      Qi.push_back(i);
    }
for(int i=k; i<n; i++)
  { cout << arr[Qi.front()] << " ";
  while((!Qi.empty()) && (Qi.front() <= i-k))
      Qi.pop_front();
  while((!Qi.empty()) && (arr[i] >= arr[Qi.back()])
      Qi.pop_back();
  Qi.push_back(i);
  }
cout << arr[Qi.front()];
}
```

Minimum number of brackets reversals needed to make make an expression balanced

Given expression with only '{' and '}'.

input:- exp = "}{" Output = 2.

input :- }}{{{{}} Output :- 1

O(n) time approach using Stack:-

first remove all balanced part of expression.

Example :- "}{{}}{{{{" to "}{{{{" by removing highlighted part. let m be total number of closing bracket ('}') and n be no. of opening brackets ('{'). we need $[m/2] + [n/2]$ reversals.

Example :- }}}}{{ required $[4/2] + [2/2] = 3$

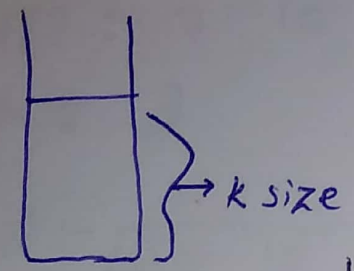
(ceil) (ceil)

```
Void CountMinReversals (String expr)
{ int len = expr.length();
  if (len % 2)
      return -1;
  Stack <char> s;
  for (int i=0; i<l; i++)
  { if (expr[i]=='}' && !s.empty())
    { if (s.top() == '{')
          s.pop();
      else
          s.push(expr[i]);
    }
```

```
else    // means  expr[i] == '}' or s.empty() == true
    s.push(expr[i]);

}
int n=0, m=0;
while(!s.empty())
{ if(s.top()=='}')
    n++;
  else
    m++;
  s.pop();
}
if(a%2==0)
  { if(b%2==0)
      cout << a/2 + b/2;
    else cout << a/2 + b/2 + 1;
  }
else
  { if(b%2==0)
      cout << a/2 + b/2 + 1;
    else
      cout << a/2 + b/2 + 2;
  }
```

Heap and its identification

i) k
ii) Smallest / largest
→ Heap



Heap

Heap

- Min Heap (k + largest) → at top min element present
- max Heap (k + smallest) → at top max element present

time Complexity reduce from n log n to n log k.

Problem :- find k^{th} smallest Element

arr[] :

7	10	4	3	20	15

K = 3

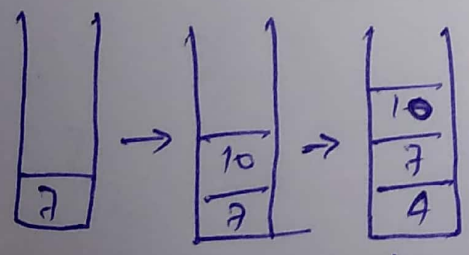
output :- 7.

3 4 7 | 10 15 20

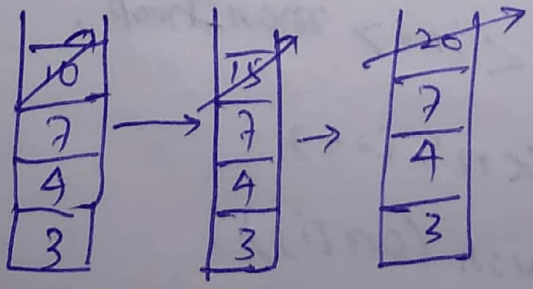
k

Sorting not used for this

n log k.



max Heap



Size greater than K.

return Heap top

ans.

Code :-

Max Heap

```
priority_queue <int> max_Heap;
```

Min Heap

```
priority_queue < int, vector <int>, greater<int> >
                                        min_Heap;
```

for pair :-

```
priority_queue < pair<int,int>, vector < pair<int,int> >,
                greater < pair< int, int > > > min_heap;
```

or

```
type define  pair< int, int>  p;
priority_queue < p , vector <p> , greater<p> > min_heap.
```

Kth Smallest Element

arr: 7 10 4 3 20 15

k = 3 o/p :- 7 , max_Heap.

```
int Solve (int arr[], int n, int k)
{ priority_queue <int> max_heap;
  for (int i=0; i<n; i++)
  { max_heap.push (arr[i]);
    if (max_heap.size() > k)
        max_heap.pop();
  } return max_heap.top();
}
```

Return K^{th} largest element in array

arr[] : 7 10 4 3 20 15

K : 3

20 15 10 7 4 3

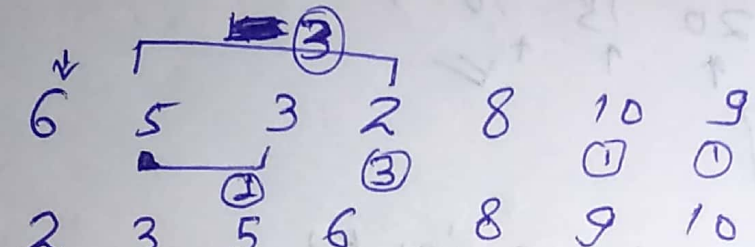
O/P :- 20, 15, 10

```
void Solve(arr[], int n, int k)
{ priority_queue < int, vector<int>, greater <int> >
                                        min_heap;
  for(int i=0; i<n; i++)
  { min_heap.push(arr[i]);
    if(min_heap.size() > k)
        min_heap.pop();
  }
while(min_heap.size())
  { cout << min_heap.top() << " ";
    min_heap.pop();
  }
}
```

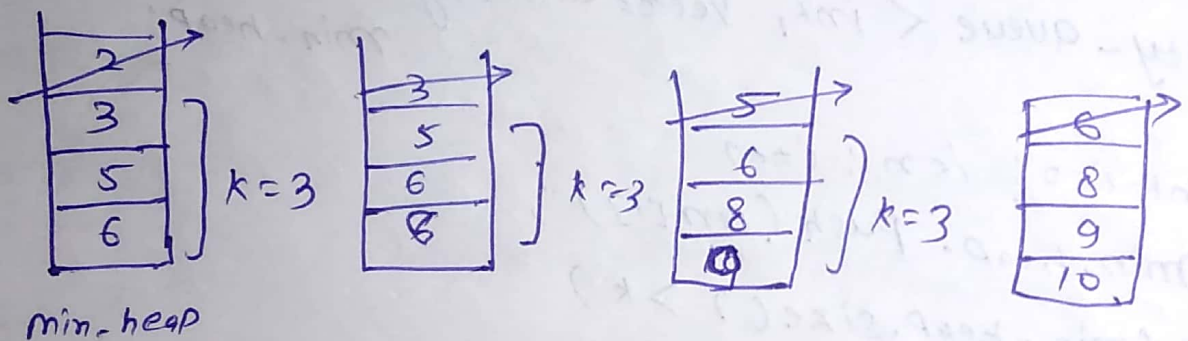
Sort a K-Sorted Array / Sort Nearly Sorted Array

arr[] :- 6 5 3 2 8 10 9

K = 3



K = 3



2 3 5 6 8 9 10

```
Void solve { int arr[], int n, int k)
{ Priority_queue <int, vector <int>, greater <int>>
                                        min_heap;
  for(int i=0 ; i<n ; i++)
  { min_heap.push(arr[i]);
    if (min_heap.size() >K)
        { Cout << min_heap.top() << " ";
          min_heap.pop();
        }
  }
While (min_heap.size())
{ Cout << min_heap.top();
  min_heap.pop();
} }
```

K Closest Number

input :- arr[] : 5 6 7 8 9

K = 3, x = 7

Output :-

5	6	7	8	9
7	7	7	7	7
2	1	0	1	2

↳ key

```
priority_queue < pair < int, int > > max_heap.
{ for (int i=0; i<n; i++)
  { max_heap.push ( { abs(x-arr[i]), arr[i] } );
    if ( max_heap.size() > K )
        max_heap.pop();
  }
while (max_heap.size())
  { cout << max_heap.top().second << ' ';
    max_heap.pop();
  }
}
```

Top K frequent Numbers

arr[]: 1 1 1 3 2 2 4

K = 2

O/P:- 1, 2

1 → 3
2 → 2
3 → 1
4 → 1

Largest
Greatest
top

↓

min-heap

Smallest
lowest
Closest

↓

max-heap

```
void Solve (int arr[], int n, int k)
{   unordered_map <int,int> mp;
    for (int i=0; i<n; i++)
        mp[arr[i]]++;
    priority_queue < pair<int,int>, vector<int,int>,
                     greater<int, int> > min_heap;
    for (auto i=mp.begin(); i!=mp.end(); i++)
    {   min_heap.push ({ i->second, i->first});
        if (min_heap.size() > k)
            min_heap.pop();
    }
    while (min_heap.size())
    {   cout << min_heap.top().second << ' ';
        min_heap.pop();
    }
}
```

Frequency Sort

arr[] :- 1 1 1 3 2 2 4

O/P :- 1 1 1 2 2 3 4

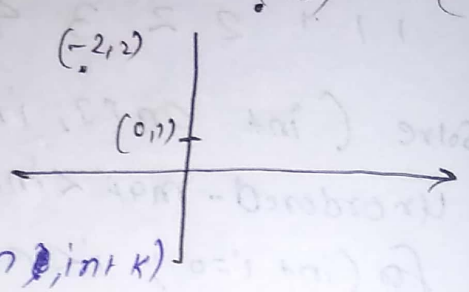
```
Void Solve (int arr[], int n)
{   unordered_map<int,int> mp;
    for(int i=0; i<n; i++)
        mp[arr[i]]++;
    priority_queue < pair<int,int>> max_heap;
    for(auto i = mp.begin(); i!= mp.end(); i++)
    { max_heap.push({i->second, i->first});
    }
    While (max_heap.size())
    { int n = max_heap.top().first;
      int m = max_heap.top().second;
      while(n--)
      { cout<<m<<" ";
      }
    }
cout<<endl;
}
```

K-Closest Points to Origin

arr[] = | x | y |
|---|---|
| 1 | 3 |
| -2 | 2 |
| 5 | 8 |
| 0 | 1 |

(10) (8) (89) (1)

k = 2



O/P :- (0,1), (-2,2)

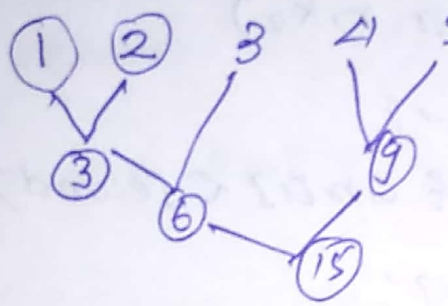
```
Void solve ( int arr[][2], int n, int k)
  { priority_queue < pair < int, pair < int, int > > > max_heap;
  for(int i=0 ; i< n ; i++)
    { max_heap.push ( { arr[i][0] * arr[i][0] + arr[i][1] * arr[i][1],
                        { arr[i][0], arr[i][1] } } );
    if (max_heap.size() > k)
        { max_heap.pop();
        }
    }
  while (max_heap.size())
        <<"("<<
  { Cout << max_heap.top().second.first << ' '
        << max_heap.top().second.second << ")" << ' ';
    max_heap.pop();
  }
}
```

Connect Ropes to minimise the Cost

arr[]: 1 2 3 4 5

Cost is if we connect 3 and 4 then cost = 7.

if we connect 4,5 then cost = 9.



Cost = 3+6+9+15

= 33 ans.

heap;

[i][j];

```
int solve(int arr[3, int n)
{ Priority_queue<int, vector<int>, greater<int>>
                                    min_heap;
int sum=0;
for(int i=0; i<n; i++)
    min_heap.push(arr[i]);
while(min_heap.size()>=2)
{ int first = min_heap.top();
min_heap.pop();
int second = min_heap.top();
  min_heap.pop();
Sum += (first+second);
min_heap.push(first+second);

}
return sum;
}
```

Sum of elements

arr[] : 1 3 12 5 15 11

$k_1^{th} = 3$

$k_2^{th} = 6$ ← Smallest

Find sum between k_1^{th} and k_2^{th}.

```
int sum=0;
int first = solve (arr, n, k1)
int second = solve (arr, n, k2)
for (int i=0; i<n; i++)
    if ( arr[i] > first && arr[i] < second)
        sum += arr[i];
return sum;
```

Minimum difference element in sorted ar

arr[] :- 1 3 8 10 12 15

key :- 12

11 9 4 2 0 3

key :- 11

10 8 3 1 1 4

```
int binarysearch (int arr[], int n, int k)
{
    int low=0;
    int high = n-1; int mid;
    while (low<=high)
    {
        mid = (low+high)/2;
        if (arr[mid]==k)
            return k;
```

```
else if (arr[mid] > k)
        high = mid - 1;
else    low = mid + 1;
}
if (abs(k - arr[low]) > abs(k - arr[high]))
        return arr[high];
else    return arr[low];
}
```

Concept use if element not meet the low and high point to neighbour of key.

Ex:- 1 3 8 10 12 15

key = 11

(11)

step after while loop

1 3 8 10 12 15

↑ high, ↑ low

return (arr[low] > 12);

longest increasing subsequence

3 4 -1 0 6 2 3 = 4

2 5 1 8 3 = 3

```
int t[n+1]; int j; t[0]=0;
for(int i=0; i<n; i++)
    t[i+1]=1;
for( int i=1; i<n; i++)
{   j=0;
    for( ; j<i; j++)
    {  if( arr[i] >= arr[j] )
           t[i+1] = max( t[i+1], t[j+1]+1);
    }
}
return max_element( t+1, t+n+1);
}
```

```
arr[i] = key;
}
```

Heap Sort O(n log n)

Heap Sort, merge sort, Selection sort independent of data present in array.

Heap, merge → Average = Best = worst = O(n log n)

Selection sort → Average = Best = worst = O(n²)

↑ Pick minm element take to prefix.

Stable

Bubble sort, insertion sort → Average / worst = O(n²)
→ Best O(n)

Quick sort → Average / best = O(n log n)
worst = O(n²)

```
Void heapify (int arr[], int n, int i)
{   int largest = i;
    int l = 2*i+1; int r = 2*i+2;
    if (l<n && arr[l] > arr[largest])
            largest = l;
    if (r<n && arr[r] > arr[largest])
            largest = r;
    if (largest != i)
    {   swap(arr[i], arr[largest]);
        heapify(arr[], n, largest);
    }
}
```

Page No.
Date :

```
Void heapsort ( int arr[], int n)
{
  for( int i = (n-1-1)/2 ; i>=0 ; i--)
        heapify ( arr, n, i);

  for( int i=n-1 ; i>0;i--)
    {
      Swap( arr[0], arr[i]);
      heapify ( arr, i, 0);
    }
}

int main ()
{  heapsort (arr, n);
}
```

O(n)

All time complexity
O(nlogn)

06-Apr-20

STL (Standard Template library)

→ For using stack, add following line

```
#include <stack>
```

```
vector < vector < int >> wrong definition
vector < vector < int > > correct definition
                    ↑
                one blank
```

VECTOR :-

It is simplest STL container. vector is just an array with extended functionality. vector is actually array with additional features.

```
vector <int> V(10);
for (int i=0; i<10; i++)
{ V[i] = (i+1)*(i+1);
}
for (int i=9; i>0; i--)
{ V[i] -= V[i-1];
}
```

Actually, when you type:

```
vector <int> V;
```

Empty vector is created. Be careful with constructions like this

```
vector <int> V[10];
```

(not used; use V(10))

most important frequently used feature of vector is that it can report its size.

```
int elements_count = V.size();
```

C++ standard template Library (STL)

It is a library of container classes, algorithms, and iterators.

STL has four components :-

(i) Algorithms (ii) Containers (iii) Function (iv) iterators

Algorithms

Header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of containers.

Sort in C++ standard template library (STL)

There is a buildin function in C++ STL by name Sort().

Prototype for sort is :-

sort (startaddress, endaddress) → It uses Quicksort, Heapsort, insertionsort.

where, startaddress :- address of first element of array.

endaddress :- address of next contiguous memory location of last element of array.

So it actually sort in range of [startaddress, endaddress).

```
#include <iostream>
#include <algorithm>
using namespace std;
Void show (int a[])
{ for (int i=0 ; i<10; i++)
    cout << a[i] << " ";
}
int main ()
{ int a[10] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
  cout << "\n array before sorting is :";
  show (a) ;
  sort (a, a+10);
  cout << "\n\n array after sorting is :";
  show (a);
  return 0;
}
```

Output:-

1 5 8 9 6 7 3 4 2 0

0 1 2 3 4 5 6 7 8 9

Binary search in C++ Standard template library (STL)

it requires array to be sorted before search applied.
It works by dividing array in half until the elements found.

The prototype of binary search is :-

binary_search (Start address, end address, value to find)

- Start address: address of first element of array.
- end address: address of next contiguous memory of last element of array.
- value to find: target value which has to be found.
- Return: boolean value element present or not.

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{ int a[] = {1,5,8,9,6,7,3,4,2,0};
  int asize = sizeof(a) / sizeof(a[0]);
  if (binary_search(a, a+asize, 2))
        cout << "\n Elements found in array";
  Else
     cout << "\n Elements not found in array";
  if (binary_search(a, a+asize, 10))
        cout << "\n Elements found in array";
  Else  cout << "\n Elements not found in array";
  return 0;
}
```

Output:-
Elements found in array
Elements not found in array

C++ magicians STL Algorithm

Non-manipulating algorithms :-

i) sort (first_iterator, last_iterator) :- to sort given vector
ii) reverse (first_iterator, last_iterator) :- reverse given vector
iii) *max_element (first_iterator, last_iterator) :- to find maxm element of
iv) *min_element (first_iterator, last_iterator) :- to find minm element of
v) accumulate (first_iterator, last_iterator, initial value of sum) :- Does the summation of the vector elements.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <numeric> → for accumulate operation
using namespace std;
int main()
{ int arr[] = { 10, 20, 5, 23, 42, 15 };
  int n = sizeof(a) / sizeof(arr[0]);
  vector<int> vect(arr, arr+n);
  for(int i=0; i<n; i++)
    { cout << vect[i] << " "; }
  sort(vect.begin(), vect.end());
  for(int i=0; i<n; i++)
  { cout << vect[i] << " "; }
  reverse(vect.begin(), vect.end());
  for(int i=0; i<n; i++)
    { cout << v[i] << " "; }
  cout << *max_element(vect.begin(), vect.end());
  cout << *min_element(vect.begin(), vect.end());
  cout << accumulate(vect.begin(), vect.end(), 0);
  return 0;
}
```

vi) count(first_iterator, last_iterator, x) → to count occurances of x in vectors.

vii) find(first_iterator, last_iterator, x) → Point to address of x.

vect.begin() → points to first element.

vect.end() → points to last element.

Return address of first element if element not present in container which is not present in container

first occurance of

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main()
```

```
{ int arr[] = { 10, 20, 5, 23, 42, 20, 15 };
  int n = sizeof(arr) / sizeof(arr[0]);
  vector<int> vect(arr, arr+n);
  cout << count(vect.begin(), vect.end(), 20);
find(vect.begin(), vect.end(), 5) != vect.end() {
        cout << "\n Element found" : cout << "\n Element not found";
  return 0;
} Output:- Occurance of 20 in vector : 2
                Element found.
```

viii) binary_search (first iterator, last iterator, x) :- Test whether x exists in sorted vector or not.

ix) lower_bound (first_iterator, last_iterator, x) :-

Manipulating algorithm :-

distance (first_iterator, desired_position) :- return distance of desired position from the first iterator. This function is very useful while finding the index.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{ int arr[] = { 5, 10, 15, 20, 20, 23, 42, 45 };
  int n = sizeof(arr) / sizeof(a[0]);
  vector<int> vect(arr, arr+n);
cout << distance(vect.begin(), max_element(vect.begin, vect.end()));
  return 0;
}
```

Output :- 7

Array algorithms in c++ STL (all_of, any_of, none_of, copy_n and iota)

i) all_of()

if checks for a given property on every elements and returns true when each elements in range satisfies property, specified else returns false.

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{ int ar[6] = {1,2,3,4,5,-6};
//checking if all elements are positive
all_of(ar, ar+6, [](int x){ return x>0;}) ? cout<<"all positive" :
                                            cout<<"all not positive";

return 0;
}
```

ii) any_of()

This function checks for given range if there is even one elements satisfying a given property mentioned in function. Returns true if at least one element satisfies property else returns false.

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{ int ar[6] = {1,2,3,4,5,-6};
// Checking if any element is negative
any_of(ar, ar+6, [](int x){ return x<0;}) ? cout<<"exist a negative element"
                                          : cout<<"all elements are non negative";

return 0;
}
```

iii) none_of ()

this function returns true if none of elements satisfies the given condition else return false.

iv) Copy_n ()

It copies one array elements to new array. This function takes 3 arguments, source array name, size of array and target array name.

```
#include <iostream>
#include <algorithm>
using namespace std;
int main ()
{ int ar[6] = {1, 2, 3, 4, 5, 6};
  int ar1[6];
  Copy_n (ar, 6, ar1);
}
```

v) iota ()

this function used to assigns continuous values to array.

It accepts 3 arguments, the first iterator, last iterator, starting numbe

```
#include <iostream>
#include <algorithm>
using namespace std;
int main ()
{ int ar[6] = {0};
  iota (ar, ar+6, 20);
  for (int i=0; i<6; i++)
  cout << ar[i] << " ";
  return 0;
}
```

Output:- 20, 21 22 23 24 25

Circular linked list connected to form

Partition in C++ STL

Partition refers to act of dividing elements of containers depending upon a given condition.

Partition operations:-

1. Partition (beg, end, condition) → used to partition element on basis of given condition
2. is_Partitioned (beg, end, condition) :- function returns boolean true if container is partitioned else return false.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main ()
{ vector <int> vect = {2,1,5,6,8,7};
  // checking if vector is partitioned using is_partitioned ()
  is_partitioned (vect.begin (), vect.end (), [] (int x) { return x%2==0; })
      ? cout << "vector is partioned" : cout << "Not partitioned ";
  cout << endl;
  // partitioning vector using partition ()
  partition (vect.begin (), vect.end (), [] (int x) { return x%2==0; });
  is_partitioned (vect.begin (), vect.end (), [] (int x) { return x%2==0; }) {
      cout << "Now vector is partitioned" : cout << "Not partitioned" ;
  cout << endl;
  for (int i=0 ; i< vect.size () ; i++)
      cout << vect[i] << "  " ;
  return ;
}
```

Output:- Not partitioned
Now vector is partitioned
2 8 6 5 1 7

3. Stable_partition(beg, end, condition) :- partition on basis of Condition such that relative order of elements is preserved.

4. partition_point (beg, end, condition) :- function returns an iterator pointing to the partition point of container. i.e first element in partioned range [beg, end] for which condition is not true. Container should already be partitioned for this function to work.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
{ vector<int> vect ={2,1,5,6,8,7};
  Stable_partition ( vect.begin(), vect.end(), [](int x){return x%2==0;});
  for(int i=0; i< vect.size(); i++)
    { cout << vect[i]; }
    return 0;
}
```

Valarray Class in C++

Public member functions in valarray class :-

1. apply() :- This function applies the manipulation given in its arguments to all valarray elements at once and returns a new valarray with manipulated values.

2. sum() :- function returns summation of elements of valarrays at once.

```
#include <iostream>
#include <valarray> // for valarray function
using namespace std;
int main()
{ valarray<int> var = {10,2,20,1,30};
  valarray<int> var1;
```

Circular linked list

Connected to form

// using apply() to increment all elements by 5

```
varr1 = varr.apply ( [](int x){return x=x+5;});
```

// Displaying new element value

```
for(int i=0; i<varr1.size(); i++)
    cout << varr1[i];
```

// Displaying sum of both old and new valarray

```
cout << varr.sum() << endl;
cout << varr1.sum() << endl;
return 0;
}
```

Output :- 15, 7, 25, 6, 35
63,
88

```
});
```

3. min() :- function returns smallest elements if valarray.
4. max() :- " " largest " " ".

```
#include <iostream>
#include <valarray>
using namespace std;
int main()
{ valarray <int> varr = {10,2,20,1,30};
    cout << varr.max() << "  " << varr.min();
    return 0;
}
```

Output :- 30. 1

5. shift() :- function returns new valarray after shifting element by number mentioned in its argument.
if number is positive, left-shift is applied.
if number " negative, right-shift " " .

6. cshift() :- function return new valarray after circularly shifting (rotating) elements by number mentioned in Argument. if number is positive, left circular shift is applied
if number is negative, right circular " " ".

```
#include <iostream>
#include <valarray>
Using namespace std;
int main ()
{
    valarray<int> varr = {10, 2, 20, 1, 30};
    valarray<int> varr1
    varr1 = varr.shift(2); // left shift by 2.
    for (int i=0; i< varr1.size(); i++)
        cout << varr1[i] << "  ";
    cout << endl;
    varr1 = varr.cshift(-3); // Circulary shift elelements to right
    for (int i=0; i< varr1.size(); i++)
        cout << varr1[i] << "  ";
    return 0;
}
```

Output :- 20 1 30 0 0
20 1 30 10 2

7. swap() :- function swaps one valarray with other.

```
#include <iostream>
#include <valarray>
Using namespace std;
int main ()
{ valarray <int> varr1 = {1, 2, 3, 4};
  valarray <int> varr2 = {2, 4, 6, 8};
  varr1.swap (varr2);
  for (int i=0; i< 4; i++)
    cout << varr1[i] << "  ";
  cout << endl;
  for (int i=0; i<4; i++)
    cout << varr2[i] << "  ";
```

Output; 2 4 6 8
1 2 3 4

Vector

vect.size() ⟶ give size of array

vect.empty() ⟶ Return True if array are empty else false.

vect.push_back(i) :- adds an element to end of vector. increasing its size by one.

vect.resize(i) ⟶ resize vector size. and put Zero to ~~empty~~ newly created location.
ex:- 25

vect.clear() ⟶ use to clear a vector. makes vector to contain 0 elements. it does not make elements Zeros it complete erases container.
rotate

∴ initialization of vector
we can create vector from another vector :-

```
Vector < int > V1;
// ....
vector < int > V2 = V1;
vector < int > V3(V1);
```

} initialization of V_2 and V_3 in this example are exactly same.

∴ Creation of vector of specific size, use following constructor :-

```
vector < int > Data (1000);
```

In this data will contain 1000 zeroes after creation.

∴ Two dimensional array via vector is to create vector of vectors.

```
vector < vector < int > > matrix;
```

Example :-
```
int N, M;
// ....
vector < vector < int > > matrix (N, vector < int > (M, -1));
```

Here we create matrix of size N*M and fill it with -1.

∴ To avoid of lot time uses and memory uses

```
Void Some-function (vector < int > v)
{
  ....
}
```

} -> never do it Unless necessary required.

instead, use following construction :-

```
Void Some-function (Const vector < int > &V)
{
  ---
}
```

→ Correct method to pass vector to a function.

if we have to change contents of vector in function just Omit "const" modifier.

```
Void modify_vector (vector < int > &r)
{
  ---
}
```

11-apr-2020

Pairs

Pair is just a pair of elements;

```
template < typename T1, typename T2 > struct pair
{ T1 first;
  T2 second;
};
```

In general pair < int, int > is a pair of integer values.

Pair < string, pair < int, int > > is pair of String and two int.

Usage :- pair < string, pair < int, int > > P;

```
String s = P.first ; // extract string
int x = P.second.first; // extract first int
int y = P.second.second ; // extract second int
```

Iterators

typeof(a+b) X = (a+b);

This will create the variable X of type matching type of (a+b) expression.

vect.insert(1,42) :- insert value 42 at 1 iterator.

All elements from second (index1) to last will be shifted right one element to leave a place for a new element.

For insert many elements, apply following technique:-

```
Vector <int> V;
Vector <int> V2;
V.insert (3, all(v2)); // Shift all elements from second to last
                       // to appropriate number of elements
                       // then copy of contents of V2 into V.
```

V.erase (iterator) :- Single element of vector is deleted.

V.erase (begin iterator, end iterator); interval, specified by two iterators, is erased from vector.

STRINGS

```
String S = "hello";
String S1 = S.substr(0,3), // "hel"
       S2 = S.substr(1,3), // "ell"
       S3 = S.substr(0, S.length()-1), // "hell"
       S4 = S.substr(1); // "ello";
```

Set

Set need a container with following features:-

→ add an element, but do not allow dubels [duplicates ?]

→ Remove elements

→ get count of elements (distinct elements)

→ check wheather elements are present in set

STL provides special container for set — set.

```
Set <int> S;
for(int i=1; i<=100; i++)
  { S.insert(i); // insert 100 elements [1.....100]
  }
S.insert(42); // does nothing, 42 already exist in set
for(int = 0; i<=100; i+=2)
  { S.erase(i); // erase even values
  }
int n = int(S.size()); // n will be 50
```

Note:- Push_back() member not used with set. It make sense since order of element in set does not matter, Push_back() is not applicable here.

Since set is not linear container, it's impossible to take element in set by index. Therefore, only way to traverse element of set is iterators.

```
// Calculate Sum of element in set using iterator
Set<int> s;
//....
int r=0;
for(set<int>::const_iterator it=s.begin(); it!=s.end(); it++)
{ r+=*it;
}
```

Traversing macros use :-

```
Set<pair<string, pair<int, vector<int>>>> ss;
int total = 0;
for(ss, it)
{ total += it->second.first;
}
```

it->second.first ↓ equivalent to '(*it).second.first'

Basic Number Theory-1

1. Modular Arithmetic

Property:-

1. $(a+b)\%c = (a\%c + b\%c)\%c$
2. $(a*b)\%c = ((a\%c)*(b\%c))\%c$
3. $(a-b)\%c = (a\%c - b\%c + c)\%c$
4. $(a/b)\%c = (a\%c * ((b^{-1})\%c))\%c$

2. Modular exponentiation :-

Calculate x^n :-

1.
```
int recursivepower(int x, int n)
{ if(n==0)
      return 1;
  return (* x * recursivepower(x, n-1));
}
```

2.
```
int iterativepower(int x, int n)
{ int result=1;
  while(n>0)
  { result = x * result;
    n--;
  }
  return result;
}
```

Efficient method :-

3.
```
int binaryExponentiation(int x, int n)
{ if(n==0)
      return 1;
  else if(n%2==0)
      return (binaryexponentiation(x*x, n/2));
  else if(n%2)
      return (x*binary exponentiation(x*x, n/2));
}
```

4. int binaryexponentiation (int x, int n)

```
{ if (n == 0)
      return 1;
else if
while (n>0)
  { if (n%.2 == 1)
        result = result * X;

    X = X * X
    n = n / 2;
  }
  return result;
}
```

Greatest Common Divisor (GCD)

using Euclid's algorithm (GCD(A,B) = GCD(B, A%.B))

```
int GCD (int A, int B)
{ if (B == 0)
    return A;
  Else
    return GCD (B, A%.B);
}
```

Extended Euclidean algorithm

Check prime number

if you have two positive number N and D, such that N is divisible by D and D is less than square root of N.

→ N/D must be greater than square root of N.

→ N is also divisible by (N/D).

Ex:- if N = 50 , $\sqrt{50} = 7$

Divisor :- 1, 50 ; 2, 25 ; 5, 10 → total 6.

```
Void checkprime (int N)
{   int count = 0;
    for(int i=1 ; i*i<=N ; ++i)
    {  if (N % i == 0)
       {  if (i*i == N)
              count ++;
          else
              count += 2;
       }
    }
    if (count == 2)
       cout << "prime number" << endl;
    else cout << "Not a prime number" << endl;
}
```

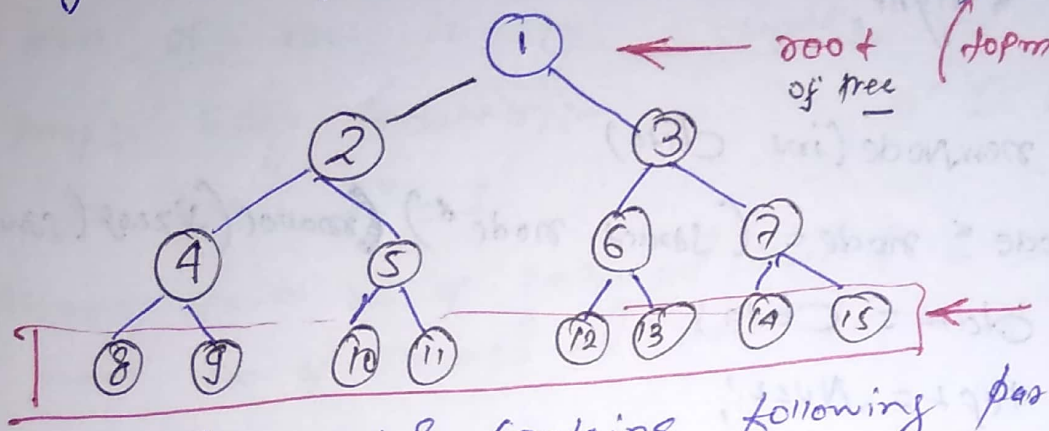
Time complexity $O(\sqrt{N})$

Sieve of Eratosthenes

Find all prime numbers that are less than or equal to given number N, and find wheather a number is prime number.

Binary Tree

A tree whose elements have at most 2 children is called binary tree. Name as left and right child.



Binary tree nodes contains following parts :-

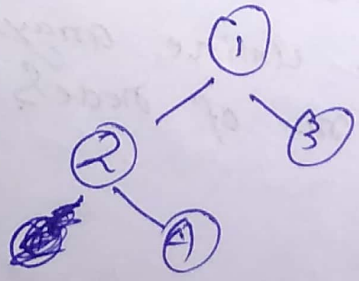
→ Data
→ pointer to left child
→ pointer to right child.

Unlike, array, linked list, stack, queues, Trees are hierarichical data structures rather than linear data structures.

→ Trees provides moderate access/search (quicker than linked list and slower than arrays)
→ provides moderate insertion/deletion (quicker than arrays and slower than linked list)
→ Like linked list unlike arrays it do not have upper limits on no. of nodes (as it linked through pointers).

```
struct node {
  int data;
  node * left;
  node * right;
};
```

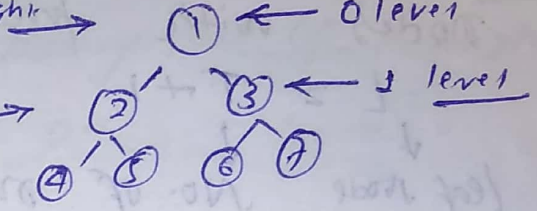
```
Struct node
{ int data;
  Struct node * left;
  struct node * right;
};
Struct node * newNode (int data)
{ Struct node * node = (struct node *) malloc (sizeof (struct node));
  node -> data = data;
  node -> left = NULL;
  node -> right = NULL;
  return (node);
}
int main ()
{ Struct node *root = newNode (1);
  root -> left = newNode (2);
  root -> right = newNode (3);
  root -> left -> right = newNode (4);
  return 0;
}
```



Properties :-

1) maximum number of nodes at level 'l' of binary tree is 2^l.

level of root is 0.



Proof :- Using induction :-

for, $l=0$ $\quad 2^0 = 1$

Assume max^m no. of nodes at level $l = 2^l$

then no. of nodes at level $l+1 = 2^{l+1} = 2 * 2^l$ satisfies since if nodes have max^m 2 children.

2) max^m number of nodes in binary tree of height 'h' is $2^h - 1$.

→ **Proof :-**

max^m no. of nodes $= 2^0 + 2^1 + 2^2 + \cdots + 2^{(h-1)}$

$= \frac{2^0(2^h - 1)}{2-1} = 2^h - 1.$

3) Binary tree with N nodes, min^m possible height or min^m no. of levels is $\log_2(N+1)$

Proof :- $2^h - 1 = N$

$\Rightarrow h = \log_2(N+1)$, $l = \log_2(N+1) - 1$.

level

4) Binary tree with L Leaves has at least levels :-

$2^{l-1} >= L$

$l >= \log_2 L + 1$

↓

min^m number of levels.

(5) In binary tree where every node has 0 or 2 children number of leaf nodes is always one more than nodes with two children.

$L = T + 1$

L → leaf node, T → No. of nodes with two children.

Assume height h :-

total nodes $= 2^h - 1$

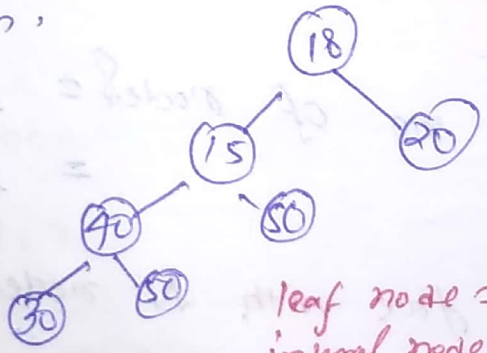
leaf nodes $= 2^{h-1}$

nodes with two child $= 2^{h-1} - 1$

Types of binary tree

i) Full Binary tree :- Binary tree is full if every node has 0 or 2 children.

(Means node full)



In full binary tree :-

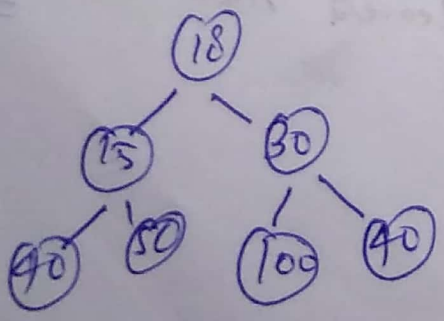
leaf node = internal node + 1

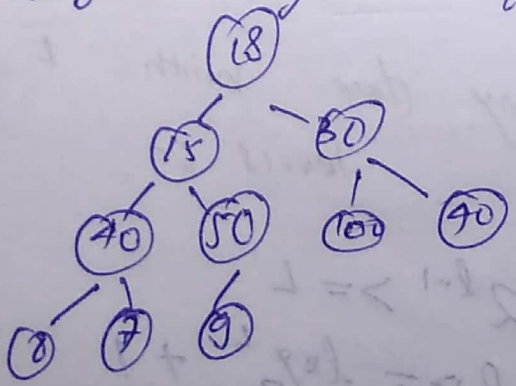
leaf node = 4 = 3 + 1

internal node = 3

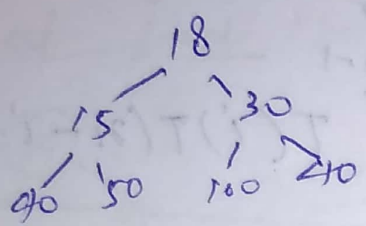
ii) Complete Binary Tree :-

if all levels are completely filled except last level and last level has all keys as left as possible.



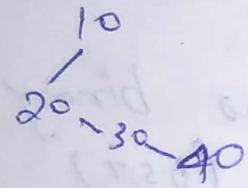


(3) **Perfect Binary tree:-** All internal nodes have two children and all leaf nodes at same level.



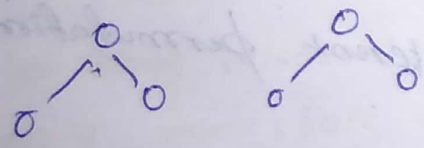
Perfect binary tree of h height have $2^h - 1$ nodes.

(4) **degenerate tree:-** where every internal node has one node child. Such trees are performance wise same as linked list.



∴ Binary tree is labeled if every node is assigned label.

two same unlabeled trees



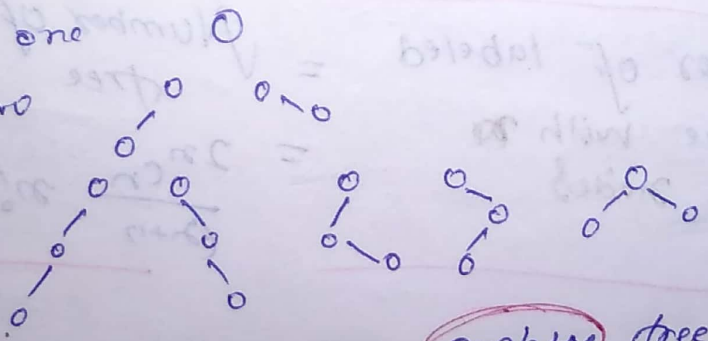
two different labeled trees

Number of different Unlabeled tree with n nodes

for n=1 → only one

for n=2 → two

n=3 → five



T(0) = 0 → There is only one empty tree.

T(1) = 1 → 5

T(2) = 2

T(3) = 5 = T(0) * T(2) + T(2) * T(0) + T(1) * T(1) = 5

T(4) = 14

$T(n)$ = C_n

$T(n)$ → no. of different Unlabeled free.

C_n → Catalan numbers

$$T(n) = \sum_{i=1}^{n} T(i-1)\,T(n-i) = \sum_{i=0}^{n-1} T(i)\,T(n-i-1)$$

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

$T(i-1)$:- no. of nodes in left side

$T(n-i-1)$:- no. of " " right side.

$$C_n = \frac{2n!}{(n+1)!\,n!} = \frac{(2n C_n)}{(n+1)}$$ → Same for binary search tree (BST)

Number of ~~binary~~ labeled binary tree with n nodes

Every unlabeled tree with n nodes can create $n!$ different labeled tree by assigning different permutation of labels to all nodes.

Number of labeled tree with n nodes = Number of unlabeled tree $\times\ n!$

$$= \frac{2n C_n}{(n+1)}\,n! = \frac{2n!}{(n+1)\,n!}$$

Given tree

11

as le

node

new

tree un

is comp

using no

Struct N

int k

Node * l

Node (

{ key =

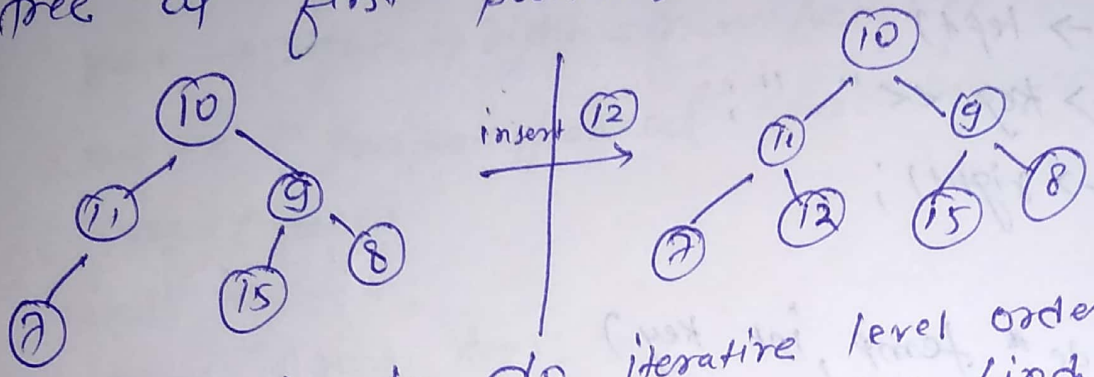
left =

3

3;

Insertion in a Binary Tree in level order

Given binary tree and key, insert key into binary tree at first position available in level order.

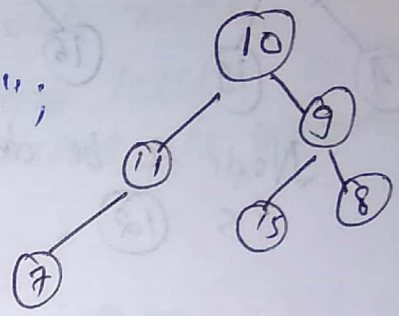


The idea is to do iterative level order traversal of given tree using queue. If we find a node whose left child is empty, we make new key as left child of node. Else if we find a node whose right child is empty, we make new key as right child. We keep traversing tree until we find a node whose left or right is empty.

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
  int key;
  Node *left, *right;
  Node (int x)
  { key = x;
    left = right = NULL;
  }
};
```

```
Void inorder (Node * temp)
{ if (!temp)
      return;
  inorder (temp -> left);
  Cout << temp -> key << " ";
  inorder (temp -> right);
}
Void insert (Node * temp, int key)
{ queue < Node * > q;
  q.push (temp);
  While (! q.empty ())
  { Node * temp = q.front ();
    q.pop();
    if (! temp -> left)
      { temp -> left = new Node(key);
        break;
      }
    Else  q.push (temp -> left);
  if (! temp -> right)
    { temp -> right = new Node (key);
      break;
    }
  else q.push (temp -> right);
}
  int main ()
{ Node * root = new Node (10);
  root -> left = new Node (11);
```

```
root -> left -> left = new Node(7);
root -> right = new Node(9);
root -> right -> left = new Node(15);
root -> right -> right = new Node(8);
Cout << "inorder traversal before insertion";
inorder(root);
int key=12;
insert (root, key);
Cout<<endl;
Cout << "inorder traversal after insertion";
inorder(root);
return 0;
```

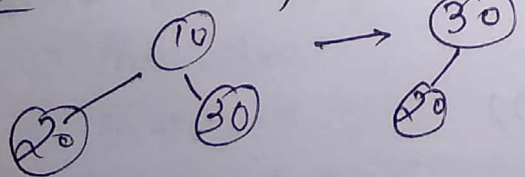


3 Output:- inorder traversal before insertion : 7 11 10 15 9 8
" " after " : 7 11 12 10 15 9 8

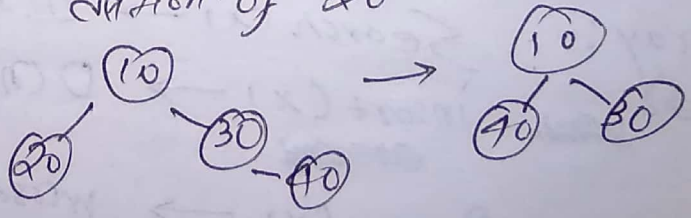
Deletion in Binary Tree.

This deletion is different from BST deletion. Here, we do not have any order among elements, so deleted element replace by last element. (bottommost ~~element~~ node and rightmost node)

Ex:- deletion of 10



deletion of 20

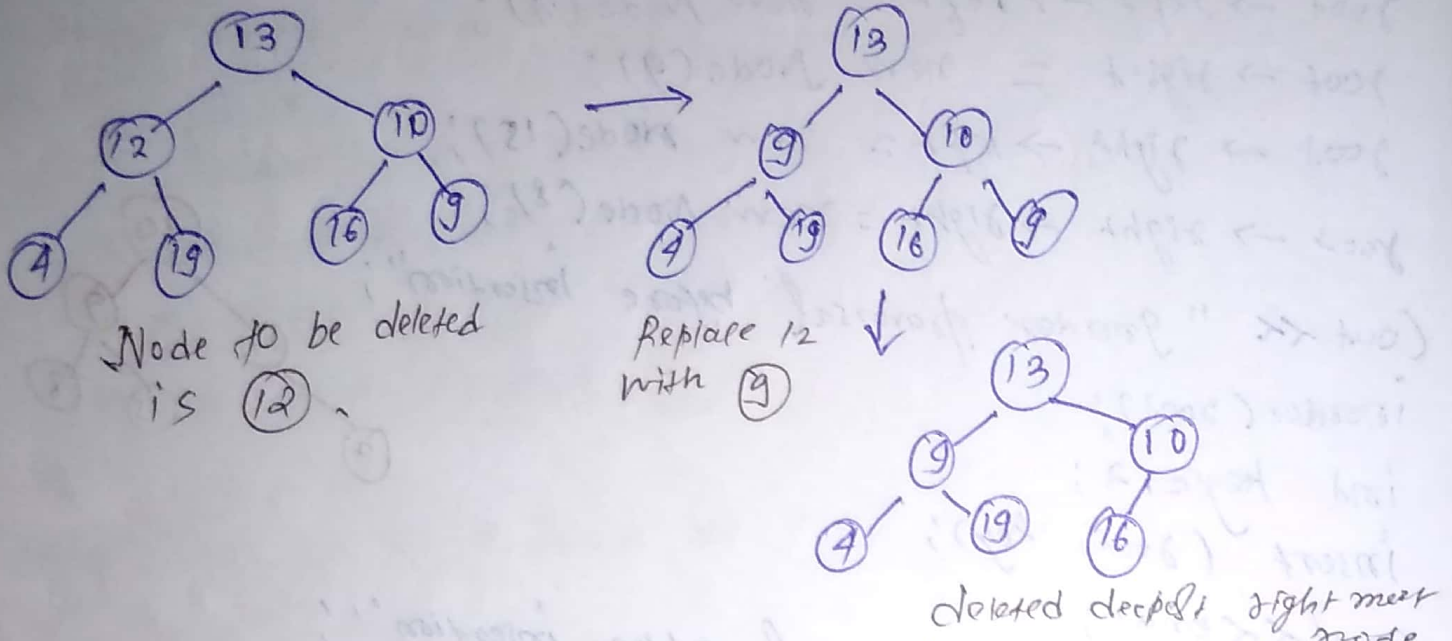


Algorithm:-

i) Starting from root, find deepest and rightmost node in binary tree and node which we want to delete.

ii) Replace deepest rightmost node's data with node to be delete.

iii) Delete deepest rightmost node.



Depth of x :-
No. of edges in path from root to x

Height of x :-
No. of edges in longest path from x to leaf.

Height of tree :- Height of root is equal to Height of tree.

Binary Search Tree

Quick Search, deletion, insertion.

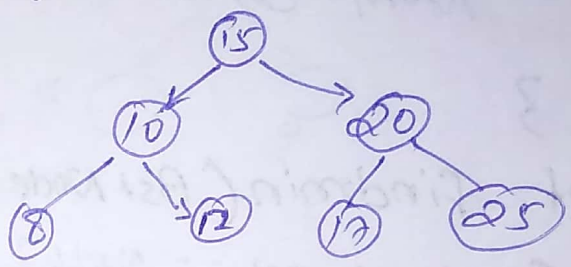
Array :- Search (x) → O(log n)
Unsorted insert (x) → O(n)
Remove (x) → worst O(n)

linked list → Search (x) → O(n)
unsorted insert (x) → O(1)
remove (x) → O(n)

Binary search tree (balanced) → search O(logn)
insert O(logn)
delete O(logn)

↳ A binary tree in which for each node, value of all nodes in left subtree are lesser or equal and value of all nodes in right subtree is greater.

binary search O(logn)



binary search tree

```
Struct BstNode {
    int data;
    BstNode * left;
    BstNode * right;
};  int main() {
BstNode* rootptr;
  rootptr = NULL;
  insert (root, 15);
```

BstNode* Get

Find min and max element in a BST

```
Struct BstNode {
    int data;
    BstNode* left;
    BstNode* right;
};
```

```
int Findmin( BstNode* root)
{ BstNode * Current = root;
  if (root == NULL)
      return -1;
  while (current -> left != NULL)
     { Current = Current -> left; }
  return Current -> data;
}
```

Iterative

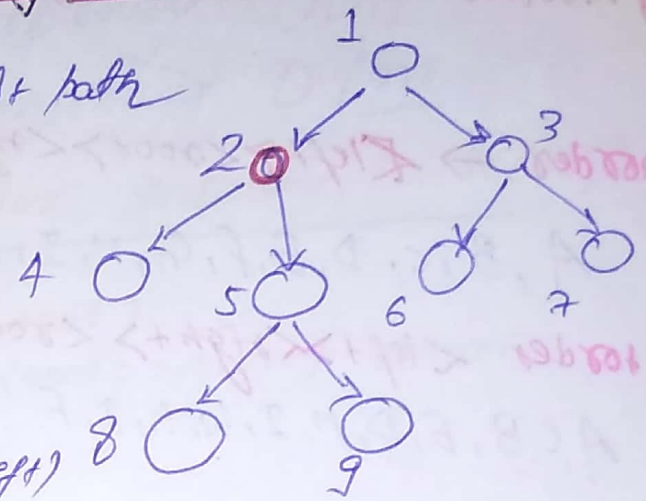
```
int Findmin( BstNode *root)
{ if ( root == NULL)
   { return -1; }
  Else if ( root -> left == NULL)
        return root -> data;
  Else return Findmin(root -> left);
}
```

Recursive

```
int Findmax ( BstNode * root)
{ BstNode * Current = root;
  while ( current -> right != NULL)
        { current = current -> right ; }
  return current -> data;
}
```

Find height of Binary tree

→ no. of edges in longest path from root to leaf.



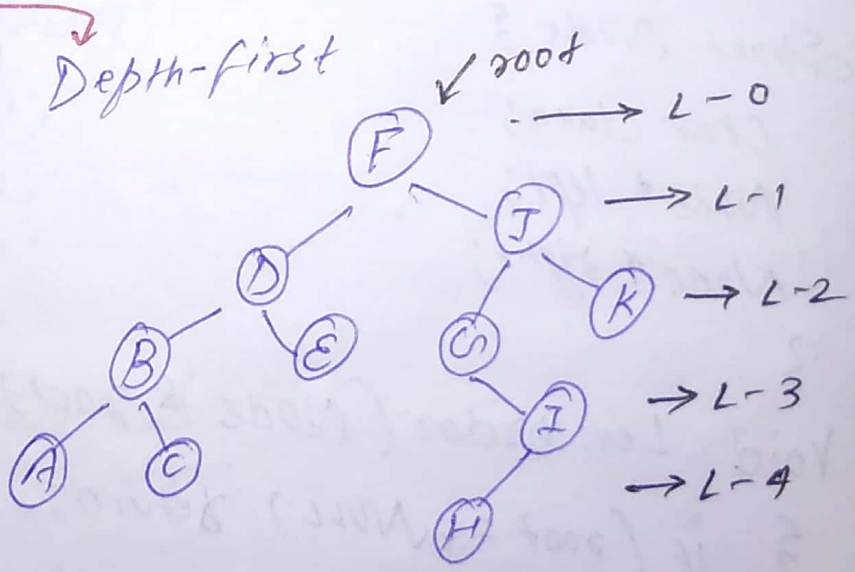

```
Find Height (root)
{  if (root == NULL)
       return -1;
   left height = Find Height (root -> left)
   Right height = Find Height (root -> right)
   return max (left height, right height) + 1;
```

3 Binary tree traversal

→ Breadth-first (Level order)

F, D, J, B, E, S, K, A, C, I, H

→ Depth-first



Depth-first

i) <root> <left> <right> → preorder

ii) <left> <root> <right> → inorder

iii) <left> <right> <root> → postorder

Preorder → <root><left><right>

Inorder → <left><root><right>

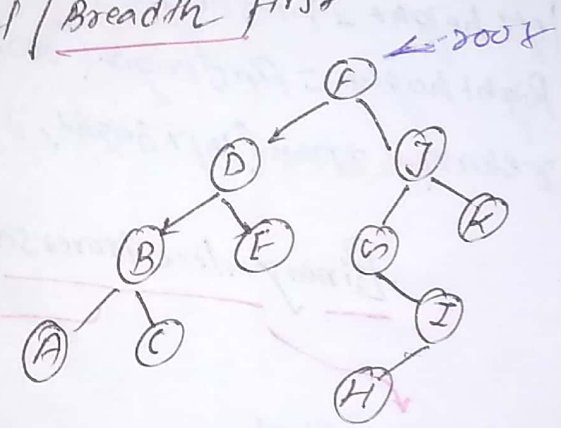
A, B, C, D, E, F, G, H, I, J, K

Postorder <left><right><root>

A, C, B, E, D, H, I, G, K, J, F

level-order traversal / Breadth first

F D J B E G K A C I H




```
#include<iostream>
Using namespace Std;
Struct Node {
   char data;
   Node * left;
   Node * right;
}
Void LevelOrder(Node * root)
{ if (root == NULL) return;
   queue<Node*> Q;
   Q.push(root);
   while (!Q.empty())
   { Node * current = Q.front();
   Cout << current -> data << " ";
   if (current -> left != NULL) Q.push(current -> left);
   if (current -> right != NULL) Q.push(current -> right);
```

```
}
Void
{ Co
  Pr
  Pr
}
Void
{ if
  gr
  co
  gr
}
```

Q.pop();

}

}

Time-complexity O(n)

Space-Complexity O(n)

Depth-first (preorder, inorder, postorder)

```
Void preorder(struct Node *root)
{ Cout << root->data;
  preorder(root->left);
  preorder(root->right);
}
```

if (root==NULL) return;

Extra space O(h) → height of tree for stack function call.

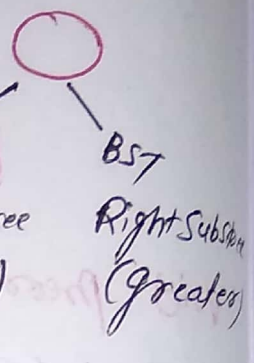
```
Void inorder(struct Node *root)
{ if (root==NULL) return;
  inorder(root->left);
  cout << root->data;
  inorder(root->right);
}
Void postorder(struct Node *root)
{ if (root==NULL) return;
  inorder(root->left);
  inorder(root->right);
  Cout<< root->data;
}
```

Time complexity O(n)

Space " O(h)

Check if a given binary tree is BST

BST :- binary tree in which for each node value of all node in leftsubtree should be less or equal and value of all node in rightsubtree should be greater.




```
bool IsBinarySearchTree(Node *root)
{ if(root==NULL) return True;
  if(IssubtreeLesser(root->left, root->data) &&
     Issubtree Greater(root->right, root->data) &&
     IsBinarySearchTree(root->left) && IsBinarySearchTree
     (root->right))
          return true;
  else    return false;
}
bool IssubtreeLesser(Node *root, int value)
{   if(root==NULL) return true;
  if(root->data<=value && IssubtreeLesser(root->left, value)
     && IssubtreeLesser(root->right, value))
          return true;
  else    return false;
}
bool Issubtreegreater(Node *root, int value)
{ if(root==NULL return true;
  if(root->data>value && issubtreegreater(root->left, value)
     && Issubtreegreater(root->right, value))
          return true;
  else    return false; }
```

method-2

```
· IsBstUtil (Node *root, int minValue, int maxValue)
{ if (root == NULL) return true;
  if (root -> data > minvalue && root -> data < maxvalue
     && IsBSTUtil(root->left, minvalue, root -> data)
  && IsBSTUtil (root -> right, root->data, maxvalue)
      return true;
  else return false;
}
bool IsBinarysearchTree (Node* root)
{ return IsBSTUtil (root, INT_MIN, INT_MAX);
}
```

Delete a node from BST

```
Struct Node* Delete (struct Node *root, int data)
{   if (root == NULL) return root;
    else if (data < root -> data)
            root->left = Delete (root -> left, data);
    else if (data > root -> data)
            root -> right = Delete(root -> right, data);
    else
    { if (root -> left == NULL && root -> right == NULL)
      { delete root;
        root = NULL;
        return root;
      }
```

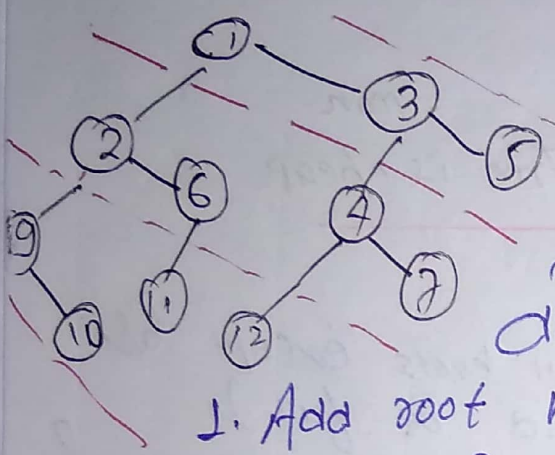
```
else if (root->left == NULL)
  { struct Node *temp = root;
    root = root->right;
    delete temp;
    return root;
  }
else if (root -> right == NULL)
    { struct Node *temp = root;
      root = root -> left;
      delete temp;
      return root;
    }
Else {
    struct Node *temp = find min (root -> right);
    root -> data = temp -> data;
    root -> right = delete (root -> right, root -> data);
    return root;
    }

} Node * findmin (Node * root)
  { if (root -> left == NULL)
      return root;

    else
      return find min (root -> left);

  }
```

Inorder Successor in BST

```
Struct Node * GetSuccessor (Struct Node * root, int data)
{ Struct Node * current = Find (root, data);
  if (current == NULL) return NULL;
// case I : Node has right subtree
 if (current -> right != NULL)
    { Struct Node * temp = current -> right;
    while (temp -> left != NULL) temp = temp -> left;
     return temp;
    }
Else
{ Struct Node * sucessor = NULL;
  Struct Node * ancestor = root;
  while (ancestor != current)
    { if (current -> data < ancesstor -> data)
        { successor = ancestor;
          ancestor = ancestor -> left;
        }
      else
        ancestor = ancestor -> right;
    }
  return sucessor;
}
}
```

```
root -> right = ConvertExpression(str, i);
return root;
}
Else return root;
}
```

Diagonal Sum of Binary Tree



Algorithm:-

idea is to keep track of vertical distance from top diagonal passing through root. We increment vertical distance we go down to next diagonal.

1. Add root with vertical distance as 0 to queue
2. Process the sum of all right child and right of right child and so on.
3. Add left child current node into queue for later preprocessing. Vertical distance of left child is vertical distance of current node plus 1.
4. keep doing 2nd, 3rd and 4th step till queue is empty.

```
Vector<int> Diagonal_sum(Node* root)
{ Vector<int> v; map<int, int> mp;
  int x=0;
  Solve(root, mp, x);
  for(auto it = mp.begin(); it != mp.end(); it++)
      v.push_back(it->second);
  return v;
}
```

```
void Solve(Node* root, map<int,int>& mp, int x)
{
    if(root==NULL)
        return;
    mp[x]+= root->data;
    Solve(root->left, mp, x+1);
    Solve(root->right, mp, x);
}
```

Check if given binary Tree is max heap

For max heap two condition satisfy:-

i) It should be Complete tree - (i.e all levels except last should be full).

ii) Every node's value should be greater than or equal to its child node.

```
int CountNodes(struct Node* root)
{
    if(root==NULL)
        return 0;
    return 1+CountNodes(root->left)+CountNodes(root->right);
}
bool isCompleteUtil(Node* root, int index, int number_nodes)
{
    if(root==NULL)
        return true;
    if(index >= number_nodes)
        return false;
```

```
    return (isCompleteUtil(root->left, 2*index+1, number_nodes)
        && isCompleteUtil(root->right, 2*index+2, number_nodes));
}
bool isHeapUtil(Node* root)
{
    if (root->left == NULL && root->right == NULL)
        return true;
    if (root->right == NULL)
    {
        return (root->data >= root->left->data);
    }
    else {
        if (root->key >= root->left->key &&
            root->key >= root->right->key)
            return ((isHeapUtil(root->left)) &&
                    (isHeapUtil(root->right)));
        else
            return false;
    }
}
bool isHeap(struct Node* root)
{
    int count = countNodes(root);
    if (isCompleteUtil(root, 0, count) &&
        isHeapUtil(root))
        return true;
    return false;
}
```

3

DP On Trees

→ General Syntax

→ How DP can be applied to trees

→ Diameters of binary trees

→ Maximum path Sum from any node to any node

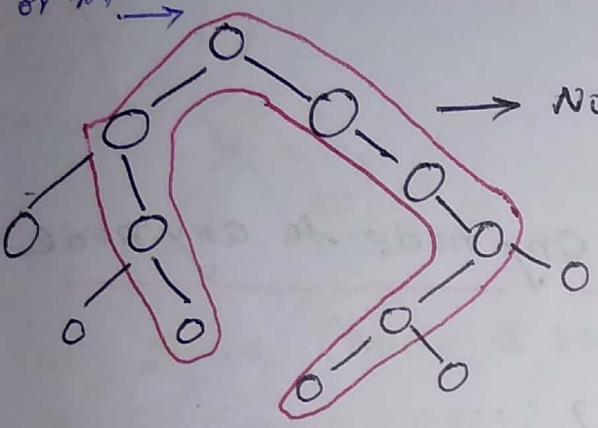
→ maximum path Sum from leaf to leaf

→ diameter of N-array trees.

Diameter of tree :- longest distance / path b/w two leaf.

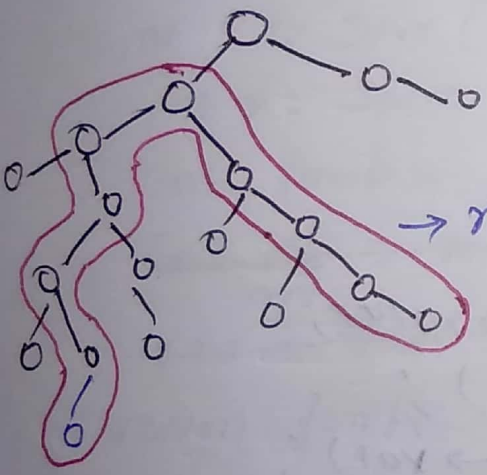
Roots included or not →

Diameter of binary trees



→ No. of Nodes in path = 9

O/P = 9



→ no. of nodes in path = 10.

O/P = 10

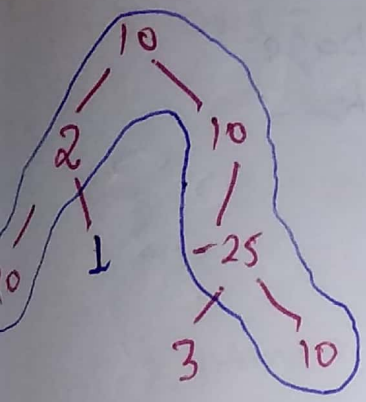
```
int Solve (Node * root, int *res)
{   if (root == NULL)
        return 0;
    int l = solve (root -> left, &res);
    int r = Solve (root -> right, &res);

int temp = max (l, r) + 1;
int ans = ~~max(temp~~, 1 + l + r);
res = max(res, ans);
return temp;
}
```

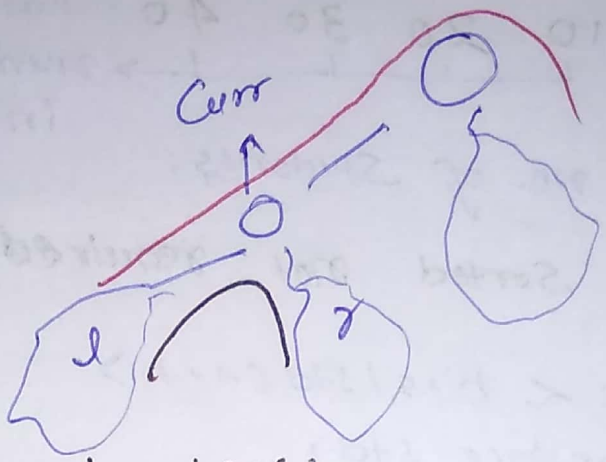
```
int main ()
{
  int res = INT_MIN;
  Solve (root, &res);
  Cout<< res <<endl;
}
```

Maximum path sum from any node to any node

```
int Solve (Node * root, int *res)
{ if (root == NULL)
        return 0;
  int l = Solve (root->left, &res);
  int r = Solve (root->right, &res);
  int temp = max( max(l, r) + root->value,
                  root->value );
  int ans = max(temp, l+r+root->val);
  res = max(res, ans);
  return temp;
}
    int main ()
  { int res = INT_MIN;
    Solve (root, &res);
    Cout<< res <<endl;
  }
```

maximum path sum from leafs to leafs



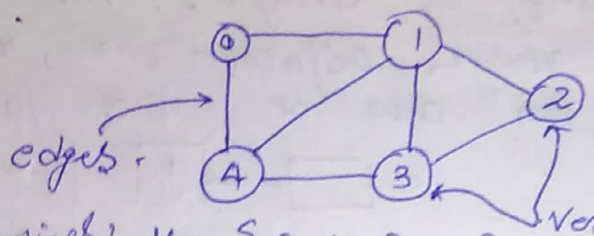



```
int Solve (Node * root, int * res)
{ if (root == NULL)
      return 0;
  int l = Solve (root -> left, &res);
  int r = Solve (root -> right, &res);
  int temp = max( l, r) + root -> value;
  int ans = max( temp, l+r+root -> value);
  res = max(res, ans);
  return temp;
}
int main()
{ int res = INT_MIN;
  Solve (root, &res);
  Cout << res << endl;
}
```

```
if (root -> left && root -> right)
    res = max(res, l+r+root -> value);
```

Graph Data Structure & Algorithms

A graph is non-linear data structure consisting of nodes (vertices) and edges (line).



Set of Nodes (vertices) $V = \{0, 1, 2, 3, 4\}$

Set of edges (lines) $E = \{01, 12, 13, 14, 23, 34, 04\}$

It is used to solve real-life problem.

Graph is data structure that consists of two components:-

A) Finite set of vertices know as nodes.

B) Finite set of ordered pair of form (u,v) called as edge. pair (u,v) indicates that is an edge from vertex u to vertex v. Edges may contain weight / value / cost.

∴ following two are most commonly used representation of graph:-

1) Adjacency matrix
2) Adjacency list

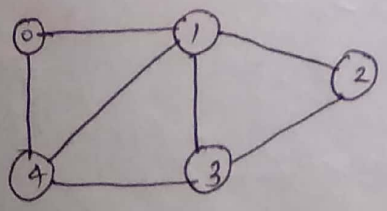
Adjacency matrix

It is 2D array of size V x V where V is number of vertices (nodes) in graph. let array be adj[][], the slot adj[i][j]=1 indicated that there is an edge from vertex i to vertex j.

Adjacency matrix for undirected graph is always symmetric.

Adjacency matrix is also used to represent weighted graphs. if adj[i][j] = w then there is an edge from vertex i to vertex j with weight w.

Undirected graph with 5 vertices

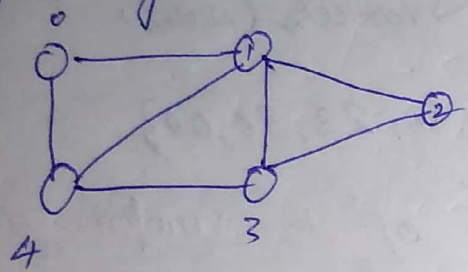


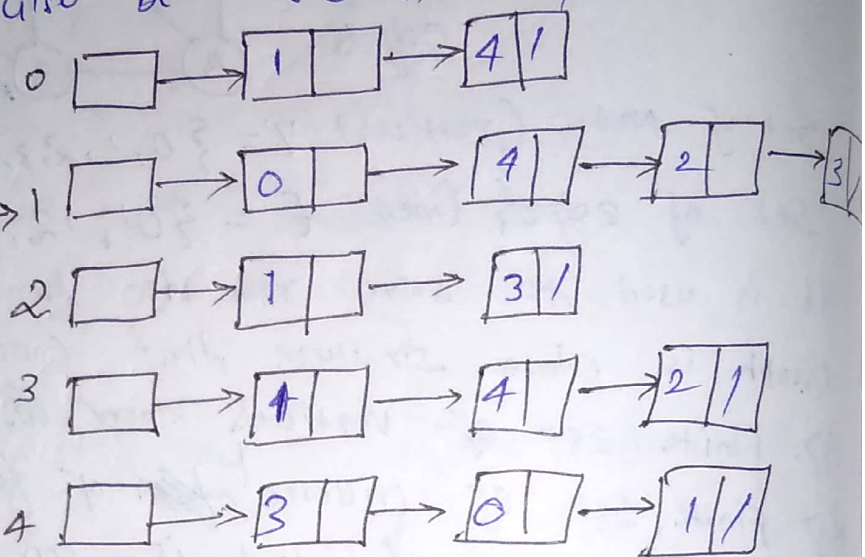
adjacency matrix

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

Adjacency list

An array of list used. Size of array is equal to number of vertices. let array be array[]. An entry array[i] represents list of vertices adjacent to i^{th} vertices. This representation can also be used to represent a weighted graph.



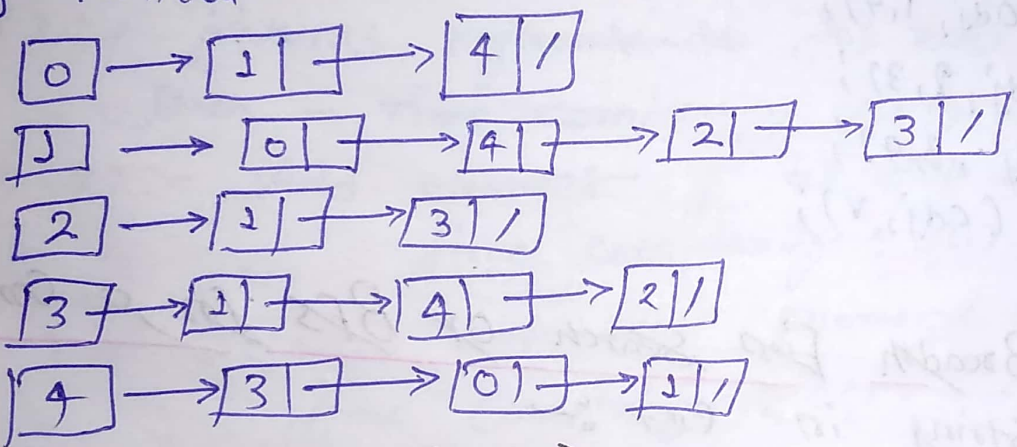


Note that in below implementation, we use dynamic arrays (vector in C++/ ArrayList in Java) to represent adjacency list.

vertex take $O(V^2)$ time.

Adjacency List:-

Array of list is used. Size of array equal to number of vertices.




```
#include <bits/stdc++.h>
using namespace std;
void addEdge (vector<int> adj[], int u, int v)
{ adj[u].push_back(v);
  adj[v].push_back(u);
}
```

```
void printGraph(vector<int> adj[], int V)
{
    for (int v=0; v<V; v++)
    {
        for (int i=0; i<adj[v].size(); i++)
            { cout << "->" << adj[v][i];
            }
        cout << endl;
    }
}
int main()
{
    int V=5;
    vector<int> adj[V];
    addEdge(adj, 0,1);
    addEdge(adj, 0,4);
    //  (adj, 1,2);
    //  (adj, 1,3);
    //  (adj, 1,4);
    //  (adj, 2,3);
    //  (adj, 3,4);
    printGraph(adj, V);
    return 0;
}
```

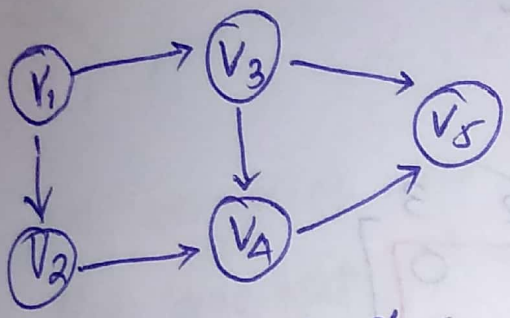
Output:-

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

Breadth First

Introduction to Graph

- undirected graph Eg:- Social network (friend)
- directed " Eg:- World wide web (page 1 has link to page 2).



Undirected graph

Outdegree (V_3) = 2

↓

No. of edge outgoing.

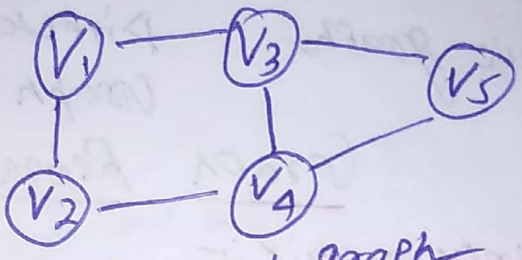
indegree (V_3) = 1.

Sum of indegree = |E|

Sum of Outdegree = |E|

maxm no. of edges = $^{V}C_2$

$= |V| * (|V|-1)$



Directed graph

degree (V_3) = 3

↓

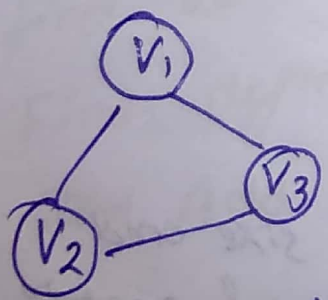
No. of edge connected to this vertices.

Sum of degree = 2 |E|

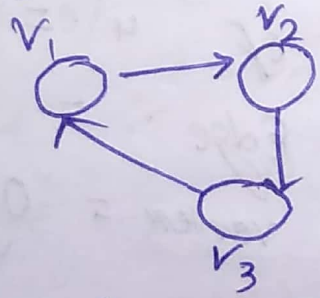
maxm no. of edges = $^{V}C_2$

$= \frac{|V|(|V|-1)}{2}$

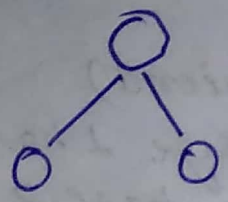
Cyclic :- There exists a walk that begins and ends at same vertex.



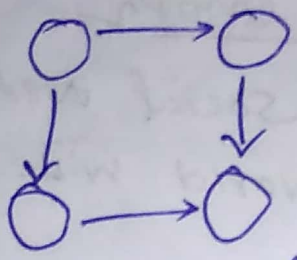
Cyclic undirected



directed cyclic graph



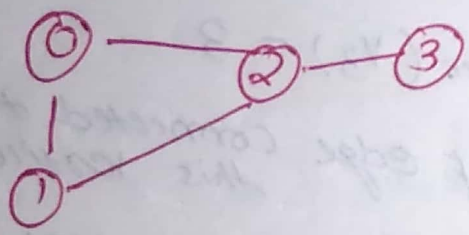
Undirected Acyclic graph



Directed Acyclic Graph (DAG)

Graph Representation

Adjacency matrix :-



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

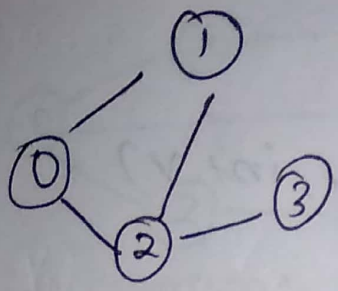
Size of matrix = $|V| \times |V|$
↳ No. of vertices

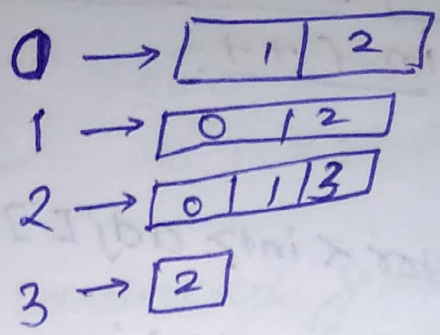
For undirected graph it is symmetric matrix.

matrix[i][j] = 1 → if there is an edge from i to j.
0 → otherwise

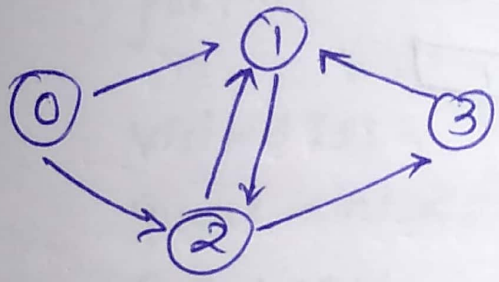
Operations :- Check if u and v are adjacent = O(1)
Find all vertices adjacent to u = O(V)
Find degree of u = O(V)
Add/remove an Edge = O(1)
Add/remove a vertex = $O(V^2)$
↳ Reallocate size and Copy previous matrix value.

Adjacency List :-

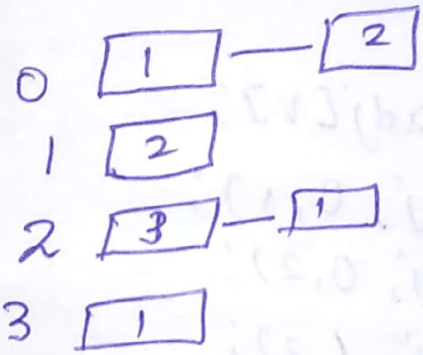




An array of list where list are properly represented as
i) Dynamic sized arrays
ii) Linked Lists.



Directed graph

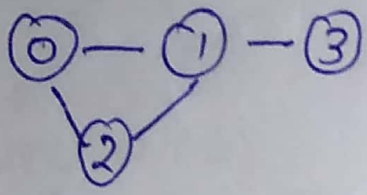


Space :- $O(V+E)$ → for directed
$O(V+2E)$ → for undirected

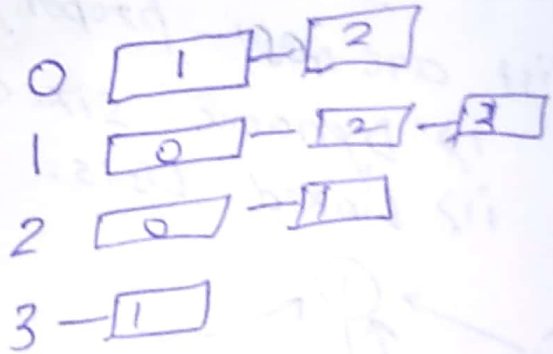
Operations :-

Check if there is an edge from u to v : $O(V)$
Find all adjacent of u :- $O(degree(u))$
Find degree of u :- $O(1)$
~~Find~~ Add an edge :- $O(1)$
Remove an Edge :- $O(V)$

Graph Adjacency list representation in C++



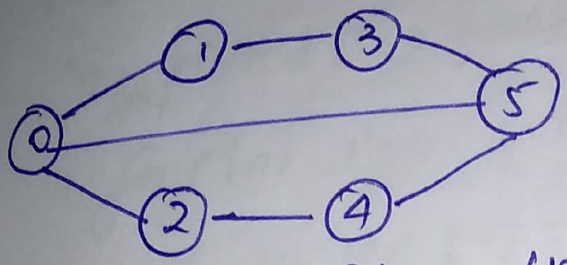

```
Void addEdge ( vector<int> adj[], int u, int v)
  { adj[u].push_back(v);
    adj[v].push_back(u);
  }
int main ()
{ int V = 4;
  vector<int> adj[V];
  addEdge (adj, 0, 1);
  addEdge (adj, 0, 2);
  addEdge (adj, 1, 2);
  addEdge (adj, 1, 3);
  return 0;
}
```



Comparision of adjacency list and matrix

	list (used for sparsh)	matrix (Dense)
memory	O(V+E)	O(V x V)
Check if there is an edge from u to v	O(V)	O(1)
Find all adjacent of u	O(degree(u))	O(V)
Add an Edge	O(1)	O(1)
Remove an Edge	O(V)	O(1)

Breadth First Search (BFS)



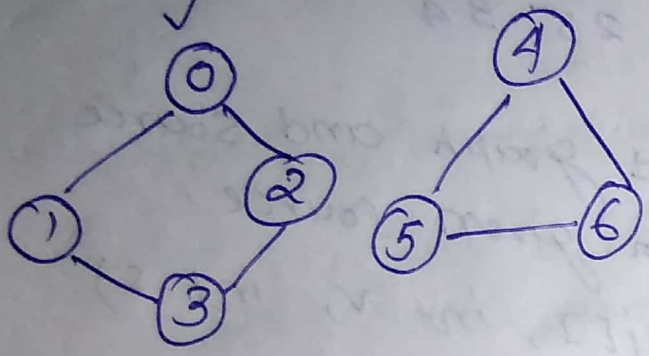
S=0

0 1 2 5 3 4

First version:- Given undirected graph and source vertex 's' print BFS from given source.

```
Void BFS (vector<int> adj[], int v, int s)
{ bool visited[v];
  for(int i=0 ; i<v; i++)
        Visited[i] = false;
  visited[s] = true;
  queue<int> q;
  q.push(s);
  while (q.emptyc)=false)
  { int u=q.fronte);
    q.pop();
    Cout<<u<<' ';
    for( vector<int>::iterator it = adj[u].begine);
                  it!=adj[u].ende) ; it++)
      { if (visited[*it] == false)
        { q.push (*it);
          visited[* it] = true;
        }
      }
  }
}
```

Second Version :- No source Given and graph may be disconnected.

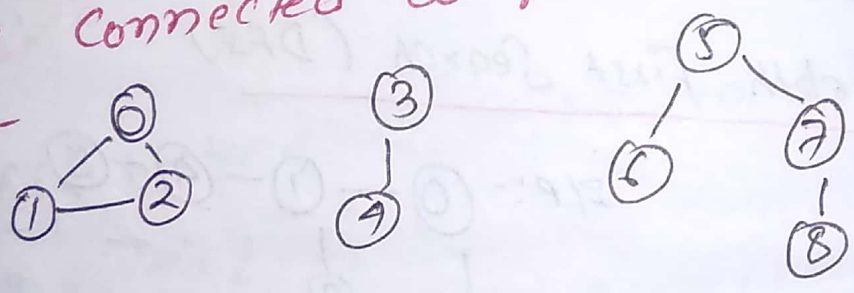



```
Void Bfs ( vector<int> adj[ ], int v, int s, bool
                                        visited[ ])
{ queue<int> q;
  q.push(s);
  visited[s] = true;
  while ( q.empty() == false)
    { int x = q.front();
      q.pop();
      cout << x << ' ';
      for( vector<int>:: iterator it = adj[x].begin();
                        it != adj[x].end(); it++)
        {  if ( visited[*it] == false)
           { q.push(*it);
             visited[*it] = true;
           }
        }
    }
}
```

```
Void BfsDfn{ vector<int> adj[], int v)
{ bool visited[v];
for(int i=0 ; i<v; i++)
    visited[i] = false;
for(int i=0; i<v ; i++)
    if( visited[i] == false)
        Bfs(adj, v, i, visited);

}
```

time complexity O(V+E);

Counting Connected compent in an undirected graph :-



Ans :- 3.

In bfsDfn :- change :-

```
int count=0;
for(int i=0; i<v; i++)
{   if( visited[i] == false)
    { count++;
      Bfs(adj, v, i, visited);
    }
}
Cout<< count<<endl;
```

Application of Bfs

i) Shortest path in ~~undirected~~ Unweighted graph

ii) Crawlers in search Engine.

iii) peer to peer network

iv) social networking search.

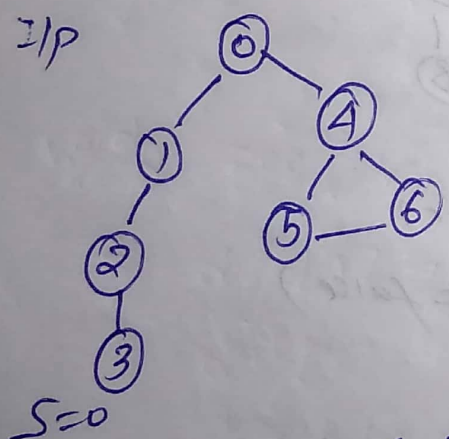
v) In Garbage Collection (cheney's Algorithm)

vi) Cycle detection. (both bfs and DFS)

vii) ford fulkerson Algorithm

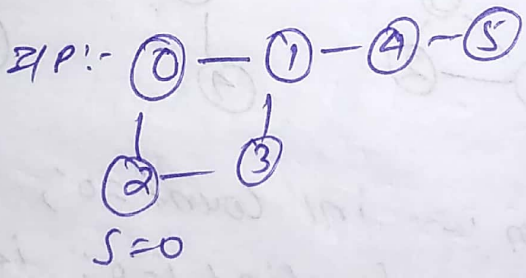
viii) Brodcasting.

Depth First Search (DFS)



S=0

O/p:- 0 1 2 3 4 5 6



S=0

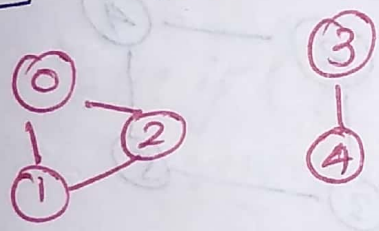
O/p:- 0 1 3 2 4 5

Case:-1 Simple undirected and connected graph

```
void dfsRec( vector<int> adj[], int v, int s, bool visited [])
{ visited [s] = true;
  cout << s << ' ';
  for(int u:adj[s])
```

```
        if (visited[u] == false)
            dfsRec(adj, v, s, visited);
    }
void dfs(vector<int> adj[], int v, int s)
{
    bool visited[v];
    memset(visited, false, sizeof(visited));
    dfsrec(adj, v, s, visited);
}
```

Case-2 disconnected graph




```
Void dfs(vector<int> adj[], int v)
{   bool visited[v];
    memset(visited, false, sizeof(visited));
    for(int i=0; i<v; i++)
        if(visited[i]==false)
            dfsrec(adj, v, i, visited);
}
```

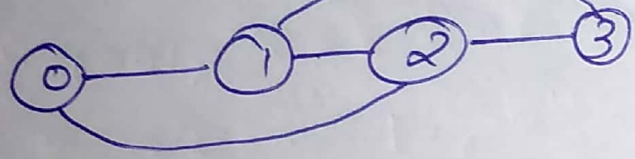
Time Complexity :- O(V+E)

Application of DFS

i) Cycle detection (DFS and BFS)
ii) topological sorting (make file utility)
iii) Strongly connected components
iv) Solving maze and similar puzzle
v) Path Finding → min no. of edge

Sortest path in Unweighted graph (BFS)

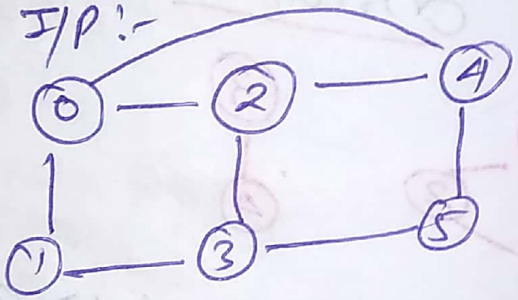
I/P



Source = 0

O/P :- 0 1 1 2

I/P :-

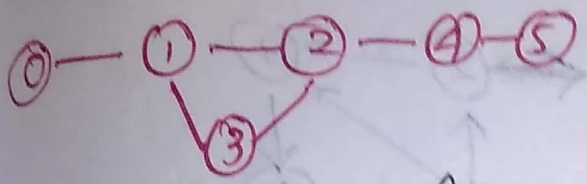


Source = 0

O/P :- 0 1 1 2 1 2

```
void BFS ( vector<int> adj[], int V, int S)
                                    visited[],
{ dist[V] = { int_MAX };
  dist[S] = 0; visited[S] = true;
  q.push(S);
  while ( q.empty() == false )
  { int u = q.front();
    q.pop();
    for ( int x : adj[u] )
      { if ( visited[x] == false )
        { visited[x] = true; q.push(x);
          dist[x] = dist[u] + 1;
} } }
```

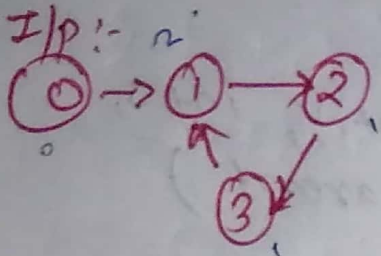
Detect cycle in an Undirected Graph




```
bool DfsRec( vector<int> adj[], int v, int s,
             int visited, int parent )
{ visited[s] = true;
  for (int x : adj[s])
  { if ( visited[x] == false )
        if ( DfsRec(adj, v, u, visited, s) == true)
            return true;
    else if ( parent != x)
        return true;

  }
  return false;
}
bool DFS ( vector<int> adj[], int v )
{bool visited[v] = {false};
  for(int i=0 ; i< v; i++)
      if (visited[i] == false)
          if (DFS( adj, v, i, visited, -1) == true)
              return true;

  return false;
}
```

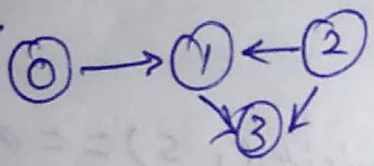
Detect cycle in undirected graph (DFS)

I/P :-



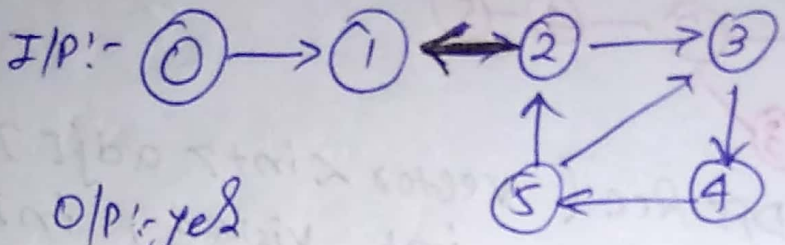
O/P :- yes

I/P :-



O/P :- NO

I/P :-



O/P :- yes

```
bool
DFSRec(adj, s, visited[], recst[])
{   visited[s] = true;
    recst[s] = true;
    for (int x : adj[s])
    {   if ( visited[s] == false && dfsrec(adj, x, visited, recst)
                                    == true)
            return true;
        else if ( recst[x] == true)
            return true;
    }
    recst[s] = false;
    return false;
}
bool dfs (vector<int> adj[], int v)
{   bool visited[s] = {false};
    bool recst[s] = {false};
```

```
for(int i=0; i<v; i++)
    if (visited[i]==false && dfsrec(adj, i, v,
                                    visited, recst)
                                    ==true)
        return true;
return false
}
```

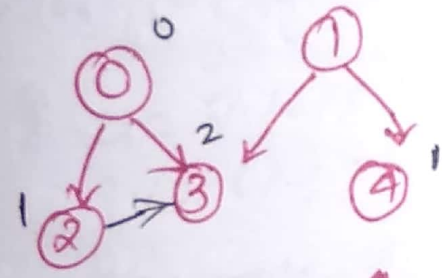
Topological Sorting (Kahn's Algorithm) (only for Non-cyclic)

I/P

Job 1 and 2 dependent job 0 so it must be printed after 0.

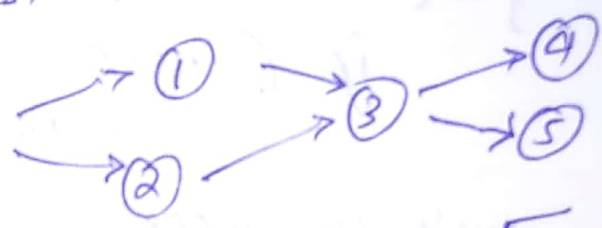
O/P 0 1 2

I/P



O/P :- 0 1 2 3 4

I/P :-



O/P :- 0 1 2 3 4 5

BFS Based Solution :-

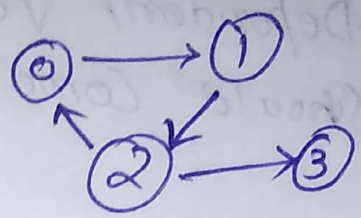
i) Store indegree of every index.
ii) Create a queue q.
iii) add all 0 indegree vertices to q.
④ while (q.empty == false)

```
{ u = q.front();
  q.pop();
  cout << u << ;
  for every v in u :-
      i) reduce indgree of v by 1.
      ii) if indgree of v becomes 0
          add v to q.
}
```

```
Void topological(vector<int> adj[], int V)
{ bool visited[V] = {false};
  int indegree[V];
  for(int i=0; i<V; i++)
     for(int x: adj[i])
        indegree[x]++;
  queue<int> q;
  for(int i=0; i<V; i++)
     if(indegree[i]==0)
        q.push(i);
  while(q.empty()==false)
  { int u = q.front();
    cout<<u<<' ';
    q.pop();
    for(int x: adj[u])
    { indegree[x]--;
      if(indegree[x]==0)
        q.push(x);
    }
  }
}
```

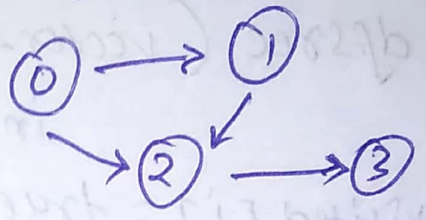
Cycle detection in Directed graph using Kahn's Algorithm

I/P :-



O/P :- yes

I/P



O/P :- NO

1) Store indegree of every vertex.
2) Create a queue q
3) Add all 0 indegree vertices to q
4) Count = 0
5) while (q.empty == false)

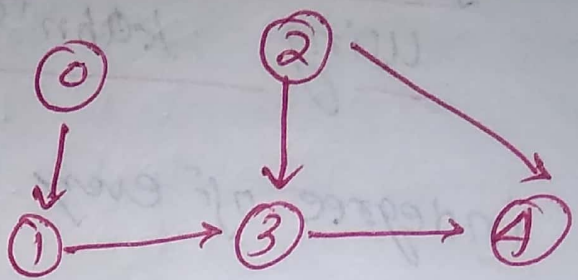
```
{ u = q.top()
  q.pop();
  for every v of u
      i) reduce indegree of v by 1
      ii) if indegree of v becomes 0
          push v to q.

  count++;
}
```

6) return (count != v);

}

Topological Sorting using DFS

Dependent job
Should come later.



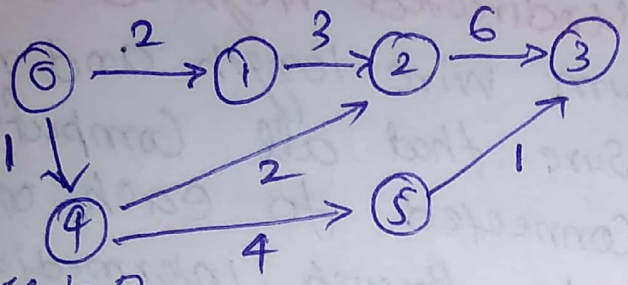
O/P :- 0 2 1 3 4

```
Void dfsrec (vector<int> adj[], int V, bool visited[],
                int i, stack<int> &S)
{ visited[i] = true;
  for(int x: adj[i])
      if (visited[x] == false)
          dfsrec(adj, V, visited, x, S);
  S.push(i);
}
Void dfs (vector<int> adj[], int V)
{ bool visited[V] = {false};
  stack<int> S;
  for(int i=0; i<V; i++)
      if (visited[i] == false)
          dfsrec(adj, V, visited, i, S);
  while (S.empty() == false)
  { cout<<S.top()<<" ";
    S.pop();
  }
}
```

Shortest path in Directed Acyclic Graph

weighted

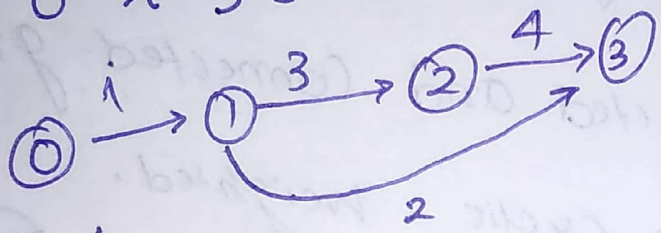
I/P :-



source :- 0

O/P :- 0 2 3 6 1 5

I/P :-



source :- 1

O/P :- INF 0 3 2

Using topological sort time complexity O(V+E)

1) Initilize dist[V] = {inf, INF, -- INF}

ii) dist[s] = 0;

iii) Find a topological sort of graph

iv) for every vertex u in topological sort.

a) for every vertex v in u

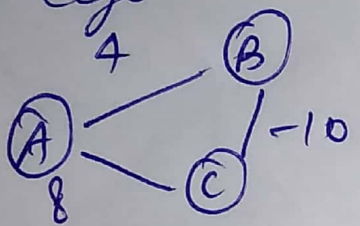
if dist[v] > dist[u] + weight(u,v)

dist[v] = dist[u] + weight(u,v);

Dijkstra's Algorithm for weighted Undirected graph

Shortest path

(1) Does not work for negative weight edge

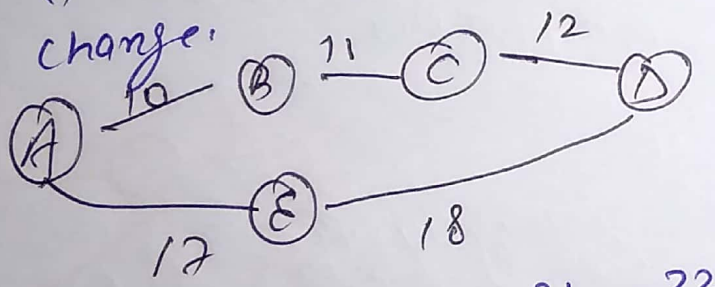


	A	B	C
A	0	4	8
B	0	4	-6
C	0	4	-6
	0	4	8

But ans :-

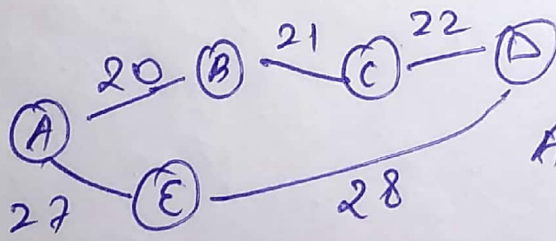
(2) Does shortest path change if add a weight to all edge of original (graph)

Ans :- change:



Shortest path from A to D :- 10+11+12 = 33

Add 10 to all



Shortest path A to D = 55

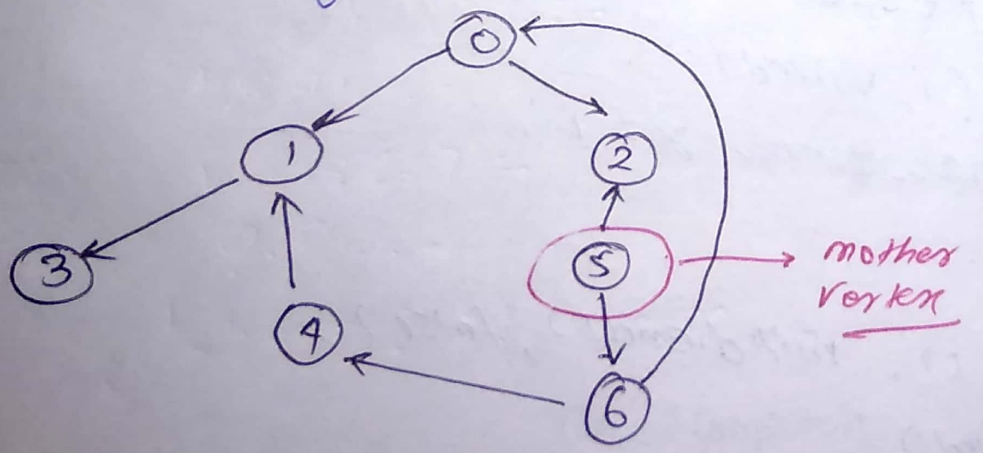
```
Void Graph::shortestPath(int src)
{ priority_queue< pair<int,int>, vector<pair<int,int>>,
                  greater<pair<int,int>>> pq;

vector<bool> visit(V, false);
vector<int> dist(V, INF);
pq.push(make_pair(0,src));
dist[src]=0;
```

```
while (pq.empty())
{ int u = pq.top().second;
    pq.pop();
    visit[u] = true;
  for (pair<int,int> p : adj[u])
      { if (visit[p.first]==false && dist[p.first] >
                                    dist[u]+p.second)
          { dist[p.first] = dist[u] + p.second;
            q.push(make_pair(dist[p.first], p.first));
          }
      }
}
Return dist;
}
```

Find a mother vertex in a Graph

→ A mother vertex in graph $G = (V, E)$ is vertex v such that all other vertices in G can be reached by a path from v.

This is for only directed - connected graph.



Naive approach :- We perform DFS / BFS on each vertices and check all vertices reach or not. Time Complexity $O(V(E+V))$ — for each vertices.

Better :-

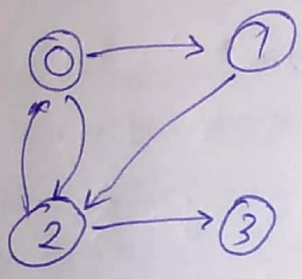
Mother vertices are always vertices of source Component in Connected ; Time :- $O(V+E) + O(V+E)$

```
Void Graph::DFSutil(int v, vector<bool> &visited)
{   visited[v] = true;
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); i++)
      { if(!visited[*i])
            DFSutil(*i, visited);
      }
}
int Graph::findMother()
{   vector<bool> visited(V, false);
                          size  initialisation
    int x=0; // for store last finished vertex (for mother vertex)
    for(int i=0; i<V; i++)
      { if(visited[i]==false)
          { DFSutil(i, visited);
            x = i;
          }
      }
    fill(visited.begin(), visited.end(), false);
    DFSutil(x, visited);
```

```
for (int i=0; i<V; i++)
  { if (visited[i]==false)
      return -1;
  }
return x;
}
```

Transitive Closure of Graph using DFS

The reach-ability matrix is called transitive closure of graph.



transitive closure :-

```
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
```

```
Void Graph::transitiveClosure()
{  bool tc[V][V];
   memset(tc, 0, sizeof(tc));
   for(int i=0; i<V; i++)
     { DFSUtil(i, tc); }
}
Void Graph::DFSUtil(int x, bool tc[][V])
{  tc[x][x] = 1;
   list<int>::iterator it
   for(it = adj[x].begin(); it!=adj[x].end(); it++)
     { if(tc[x][*it]==false)
         DFSUtil(x, tc, *it);
     }
}
```

Time complexity O(V²)

Count Pairs with given Sum

Given an arrays of integers, and a number 'sum', find number of pairs of integer in array whose sum is equal to 'Sum'.

Eg arr[] = {1,5,7,-1}
Sum = 6
Output :- 2 (1,5) and (7,-1)

arr[] = {1,5,7,-1,5}
Sum = 6
Output :- 3 (1,5) (1,5), (7,-1)

Simple solution :-

```
int getPairsCount (int arr[], int n, int sum)
{ int count=0;
  for(int i=0; i<n; i++)
     for(int j= i+1; j<n; i++)
        { if (arr[i] + arr[j] == sum)
             count++;
  return count;
}
```

Time :- O(n²)

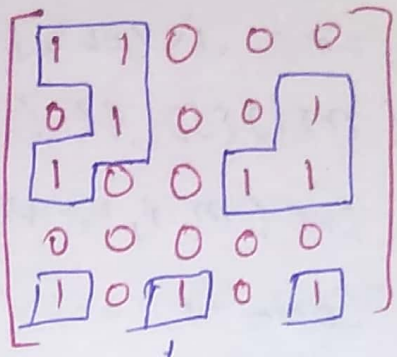
Better Solution :- O(n)

```
int getPairsCount (int arr[], int n, int sum)
{ unordered_map <int, int> mp;
  for(int i=0; i<n; i++)
     { mp[arr[i]]++; }
int count=0;
  for( int i=0 ; i<n; i++)
     { count += mp[sum-arr[i]];
       if (sum == 2 * arr[i])
          count--;
     }
return count/2;
}
```

Find number of islands using DFS

input :-

```
mat[][] = { { 1, 1, 0, 0, 0 },
            { 0, 1, 0, 0, 1 },
            { 1, 0, 0, 1, 1 },
            { 0, 0, 0, 0, 0 },
            { 1, 0, 1, 0, 1 } }
```



Output = 5

No. of Connected sub groups

```
#include <bits/stdc++.h>
using namespace std;
#define Row 5
#define Col 5
int issafe(int m[][Col], int row, int col, bool visit[][Col])
{ return (row >= 0) && (row < Row) && (col >= 0) && (col < Col)
    && m[row][col] && !visit[row][col];
}
void DFS(int m[][Col], int row, int col, bool visited[][Col])
{ visited[row][col] = true;
  static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};
  static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};
  for (int k=0; k<8; k++)
    { if (issafe(m, row+rowNbr[k], col+colNbr[k], visited))
      DFS(m, row+rowNbr[k], col+colNbr[k], visited);
    }
}
int CountIslands(int m[][Col])
{ bool visited[Row][col];
  memset(visited, 0, sizeof(visited));
```

```
int count=0;
for(int i=0; i<Row; i++)
  for(int j=0; j<COL; j++)
    if (m[i][j] && !visited[i][j])
    { DFS(m, i, j, visited);
      count++;
    }
  return count;
}
int main()
{ int m[][COL] = { {1, 1, 0, 0, 0},
                   {0, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1},
                   {0, 0, 0, 0, 0},
                   {1, 0, 1, 0, 1} };
  Cout << countIslands(m);
  return 0;
}
```

Sanake and Ladder Problem

use Breadth first serach to get minm no. of dice through.

```
#include <bits/stdc++.h>
Using namespace std;
#define siz 30
class Cell
  { int v;
    int distance;
  };
int Solve(int move[])
{ queue <Cell> q;
  Cell x, y;
```

```
x.v=0;  x.distance=0;
q.push(x);
while(q.size())
{ y=q.front();
  if(y.v==siz-1)
      break;
  q.pop();
  for(int i=(y.v+1); i<=(y.v+6) && i<siz; i++)
      { cell z;
        z.distance = y.distance+1;
        if(move[i]!=-1)
            z.v = move[i];
        else
            z.v=i;
        q.push(z);
      }
}
return y.distance;
}
int main()
{ int t; cin>>t;
  while(t--)
  { int move[siz]; int n;
    cin>>n;
    memset(move, -1, sizeof(move));
    int x,y;
    for(int i=0; i<n; i++)
      { cin>>x>>y;
        move[x-1]=y-1;
      }
    cout<<solve(move)<<endl;
  }
  return 0;
}
```

Count of Inversion of Array

Two element of array form an inversion if $a[i] > a[j]$ $i < j$.

input :- 2 4 1 3 5

output :- 3. (2,1), (4,1), (4,3)

Simple :- $O(n^2)$

```
int count=0;
for (int i=0; i<n; i++)
    for (int j=i+1; j<n; j++)
        if (arr[i] > arr[j])
            count++;
```

Best approach $O(n \log n)$ using merge sort :-

```
#include<bits/stdc++.h>
using namespace std;
int merge ( int arr[], int l, int mid, int r)
{ int n= mid-l+1; int m=r-mid;
  int x[n], y[m];
  for (int i=0; i<n ; i++)
      x[i] = arr[l+i] ;
  for (int i=0; i<m; i++)
      y[i] = arr[mid+i+1];
  int count=0;
  int i=0, j=0, k=l;
  while (i<n && j<m)
  { if (x[i] > y[j])
      { count +=(n-i);
        arr[k] = y[j];
        k++;  j++;
      }
    else arr[k++] = y[i++];
  }
```

```
    while (i<m)
      { arr[k++] = x[i++];
      }
    while (j<m)
     { arr[k++] = y[j++];
     }
    return count;
}   int solve(int arr[], int l, int r)
    { if (l>=r)
        return 0;
      int mid = (l+r)/2; int count=0;
      int count = solve(arr, l, mid);
      count += solve(arr, mid+1, r); count += merge(arr, l, mid+1, r); retu
      return count;
    }
int main()
{   int n; cin>>n; int arr[n];
    for(int i=0; i<n; i++)
        cin >> arr[i];
    cout << solve(arr, 0, n-1) << endl;
}
```

let

then,

int

{

Given

Small

be cut

power