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 $T(n) <= c\, n \log n$

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

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

$<= nc \log n$

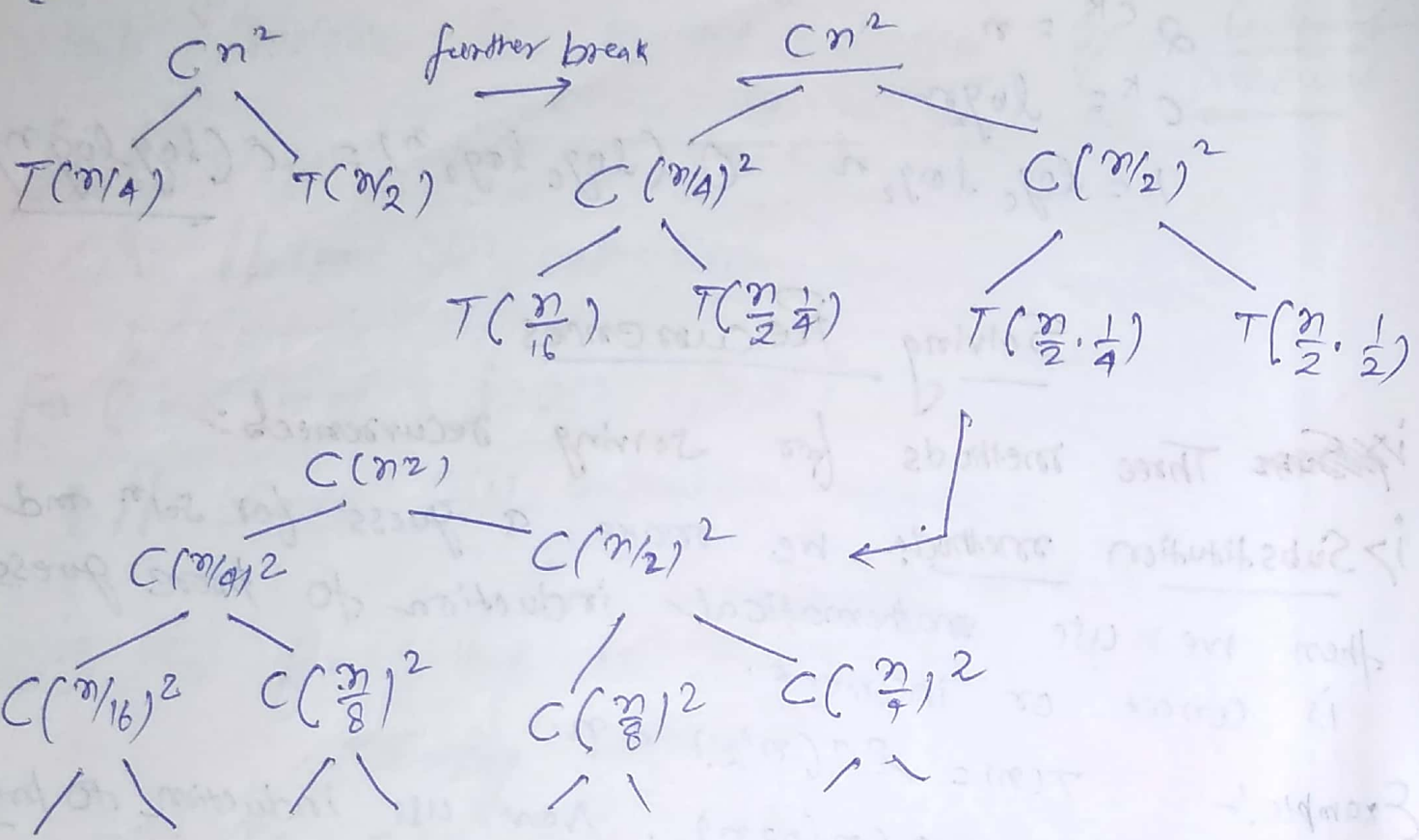
Hence, $T <= nc \log n$

$T = O(n \log n)$

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

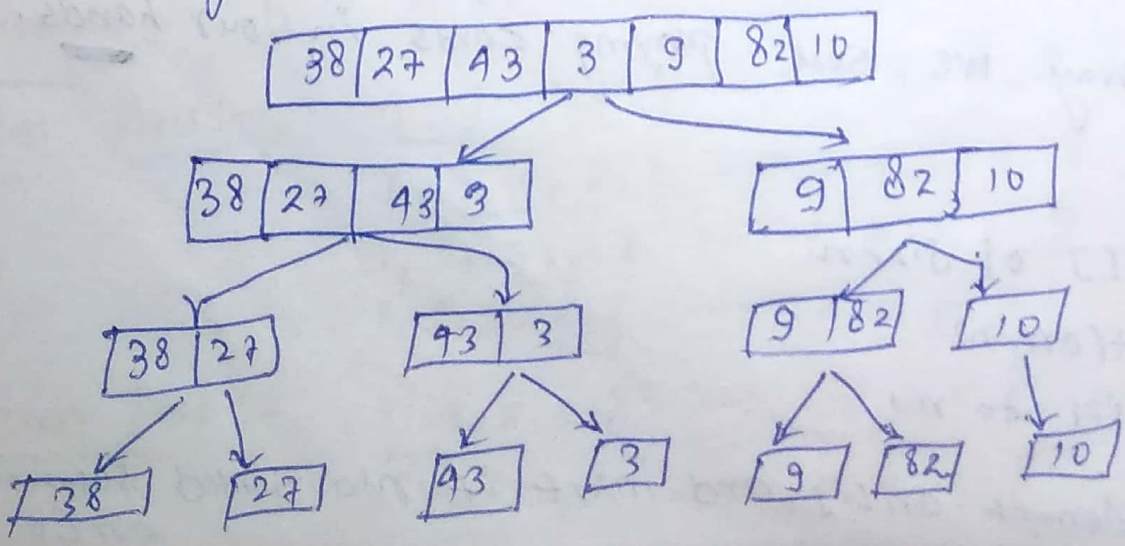
1, 2, 3, 4, 5, 10, 12, 6 → 1, 2, 3, 4, 5, 6, 10, 12

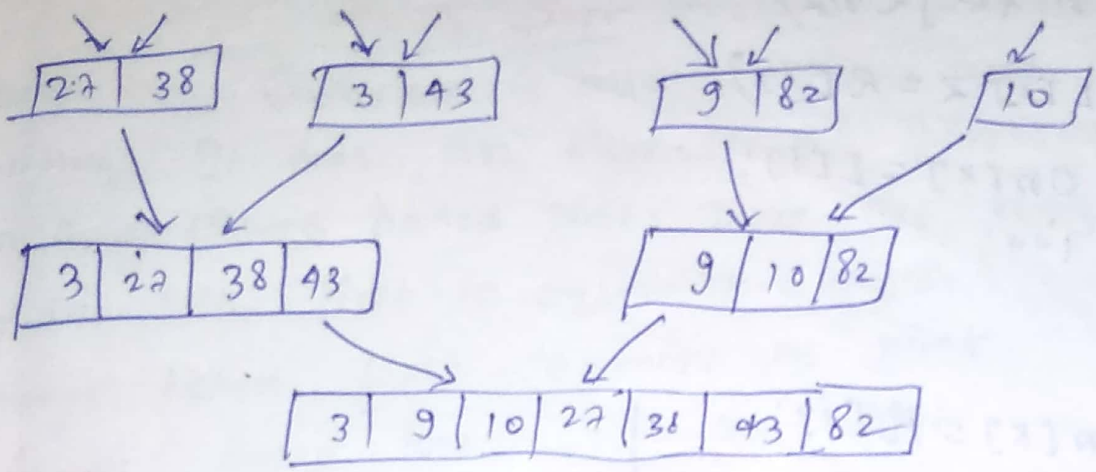
```
void insertionSort (int arr[], int n)
{  int i, key, j;
   for(i=1; i<n; i++)
    { key = arr[i];
      j = i-1;
      while ( j>=0 && arr[j] > key)
      { arr[j+1] = arr[j];
        j = j-1;
      }
      arr[j+1] = key;
    }
}
```

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 :- $O(n \log n)$

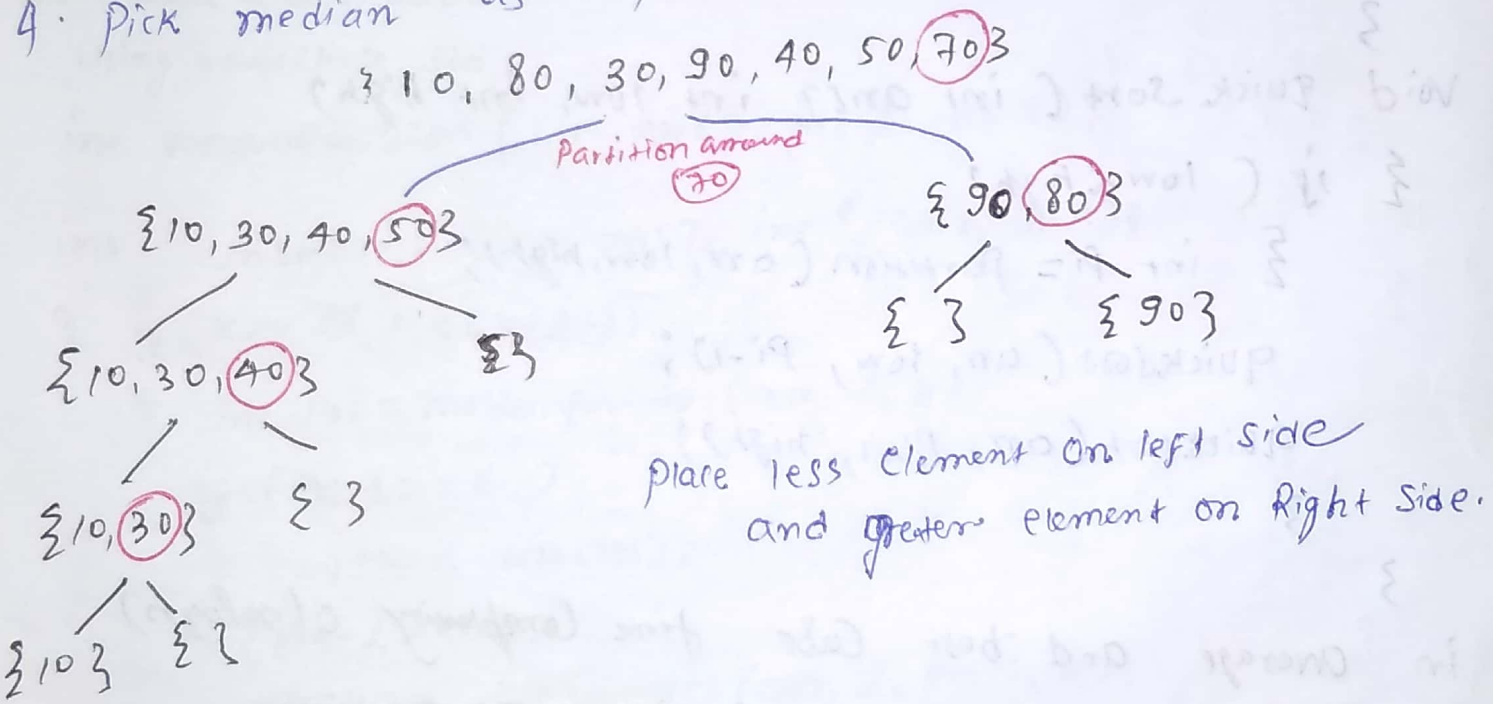
n :- for linear merge $\log n$ – two divide in half.

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

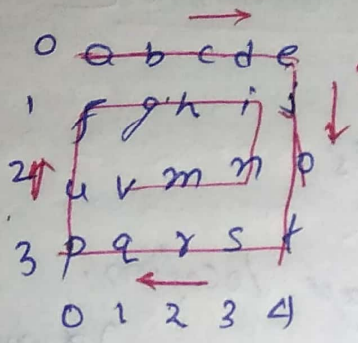
Find maxm profit if at a time only one job can be performed.

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; }