DATA STRUCTURES AND ALGORITHMS (SELF PACED)

TOPICS COVERED TILL NOW:

1. **ANALYSIS OF ALGORITHMS**
2. **ASYMPTOTIC ANALYSIS**
3. **ARRAYS(With full concept and basic to intermediate level Qn's)**
4. **Searching(With full concept and basic to intermediate level Qn's)**
5. **Sorting(With full concept and basic to intermediate level Qn's)**

SPECIAL THANKS

SANDEEP JAIN

CEO & Founder of GeeksforGeeks

PREPARED BY

VIVEK VARDHAN REDDY R

Data Structures & Algorithms

Analysis of Algorithms:

Example problem: Sum of "n" natural numbers

i) I/p : n = 3

o/p : 6 // 1+2+3

ii) I/p : n = 5

o/p : 15 // 1+2+3+4+5

Programmes:

i)
```
int fun1(int n)
{
  return n*(n+1)/2;
}
```

ii)
```
int fun2(int n)
{
  int sum=0;
for(int i=1; i<=n; i++)
{
   sum=sum+i;
}
return sum;
}
```

iii)
```
int fun3(int n)
{
  int sum = 0;
for(int i=1; i≤n; i++)
{
  for(int j=1; j<=i; j++)
   {
     sum++;
   }
}
return sum;
}
```

* Time taken to execute the Programme may depends on many factors

i) Machine efficiency (The computer is Running slow or fast)

ii) Programming language

Here "c" & "c++" are compiled lang.s "Java & "python" are interpreted languages. so, "c & c++ likely to take less time to execute

Asymptotic Analysis:

function (or) Prog. in terms of input size

* The idea is to measure order of growth of time taken by
* Does not depends upon machine, programming lang. etc
* No Need to implement, we can analyze algorithms

ORDER of GROWTH:

A function $f(n)$ is said to be growing faster than $g(n)$ if

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

OR

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

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

$f(n)$ and $g(n)$ Represent expression of Time taken

$n \geq 0$

$f(n), g(n) \geq 0$

let, $f(n) = n^2 + n + 6$

$g(n) = 2n + 5$

$$\lim_{n\to\infty} \frac{2n+5}{n^2+n+6}$$

$$\Rightarrow \lim_{n\to\infty} \frac{\frac{2}{n} + 5/n^2}{1 + 1/n + 6/n^2}$$

$$\Rightarrow \lim_{n\to\infty} \frac{0+0}{1+0+0}$$

$$\Rightarrow 0$$

∴ order of growth of $f(n)$ is more faster than $g(n)$.

Note: whenever we are trying to find out time complexity of Recursive fun. we first we have to write Recursive Relation for that function and then solve Recursive Relation

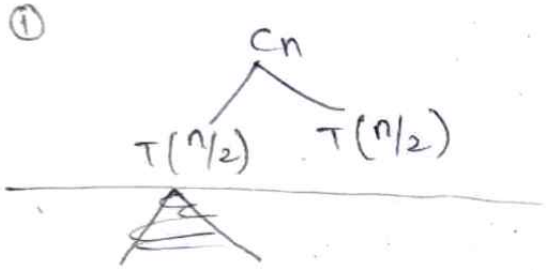
Recursion Tree method for Solving Recurrances:

* we consider the Recursion tree and compute total work done.
* we write non-recursive part as a root of tree and write the Recursive part as children
* we keep expanding untill we see the pattern

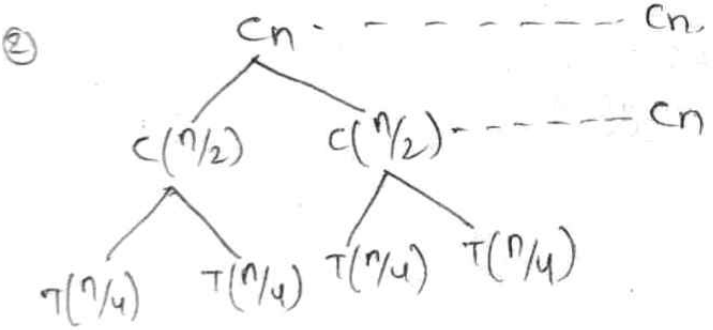
ex:- $T(n) = 2T(n/2) + Cn$

Rec. part non. Rec. part

①



②



Finding no. of digits in a number:

1) Iterative soln:

```
int countdigit(long n)
{
  int count=0
  while(n!=0)
  {
    n=n/10;
    count++;
  }
  return count;
}
```

Recursive soln:

```
int countdigit(long n)
{
  if(n==0)
  {
  return 0;
  }
return 1+countdigit(n/10);
}
```

ARITHETIC AND GEOMETRIC PROGRESSIONS

ex: 2, 4, 6, 8, 10, ----

$a = 2$

$d = 2$ (difference)

then, $a, a+d, a+2d, \ldots a+(n-1)d$.

$$\text{avg} = \frac{\text{sum}}{n}$$

$$\text{sum} = \text{avg} \times n$$

$$= \left[\frac{\text{first} + \text{last term}}{2}\right] \times n$$

$$\Rightarrow \frac{n}{2}(a + a+(n-1)d)$$

$$= \frac{n}{2}(2a+(n-1)d).$$

G.P:

ex: 2, 4, 8, 16, ----

Ratio of any two numbers, $r = 2$

$$r, ar, ar^2, \ldots ar^{n-1}$$

$$\Rightarrow \frac{a(1-r^n)}{1-r}$$

Prime number: is a number only divisible by 1 and itself.

LCM:

ex: lcm of 4 and 6?

multiples of 4 = 4, 8, 12, 16, 20, 24, -

multiples of 6 = 6, 12, 18, 24, 30, --

common multiples = 12, 24

least common multiples = 12

H.C.F:

ex: Hcf of 12 and 16

ex: factors of 12: 1, 2, 3, 4, 6, 12

factors of 16: 1, 2, 4, 8, 16, ---

common factor = 1, 2, 4

Highest common factor = 4.

factorial of a Number:

```
1) int fact(int n)
{
  int res=1;
for(int i=2; i<=n; i++)
{
  res=res*i;
}
return res;
}// Iterative Implementation.
```

for n=5

i=2; res=2

i=3; res=6

i=4; res=24

i=5; res=120

```
2) int fact(int n)
{
  if(n==0)
  {
   return 1;
  }
return n*fact(n-1);
}// Recursive Implementation
```

fact(5) 5*24=120.

↳ 5*fact(4)

4x6=24

↳ 4*fact(3)

3x2=6

↳ 3*fact(2)

2x1=2

↳ 2*fact(1)

1

↳ 1*fact(0);

Trailing zeros in a factorial:

I/p: n=5 ⇒ 1x2x3x4x5 = 120

o/p: 1

I/p: n=10 = 1x2x3x4x5x6x7x8x9x10 = 3628800

o/p=2

```
int countzero (int n)
{
   int fact = 1
for (i=2; i<=n; i++)
{
   fact = fact * i;
 }
int res = 0
while (fact % 10 == 0)
{  res++;
  fact = fact / 10;
  }
return res;
}
```

```
* int countzero (int n)
{
   int res = 0;
for (i=5; i ≤ n; i = i * 5)
 {
    res = res + n / i;
  }
return res;
}
```

Greatest common Divisor (GCD).

If "a" and "b" are two numbers given we have to find the largest number that divides "a" and "b".

ex: I/p: a=4 and b=6

o/p: 2

I/p: a=100 and b=200

o/p: 100

I/p: a=7 and b=13

o/p: 1.

Euclidean Algorithm:

Basic Idea:

let "b" be smaller than "a".

$gcd(a,b) = gcd(a-b, b)$

Why?:

let "g" be GCD of "a" and "b" then

$a = gx$ and $b = gy$ and Gi(x,y)

$(a-b) = g(x-y)$

Implementation of Euclidean Algorithm:

```
int GCD(int a, int b)
{
   while (a&&b != 0)
   {
      if (a>b)
      {
        a = a-b;
      }
      else
      {
        b = b-a;
      }
   }
   return a;
}
```

Suppose:

a = 12, b = 15

a = 12, b = 3

a = 9, b = 3

a = 6, b = 3

a = 3, b = 3

*Insted of using Repeated substration, we can use modulaor division.

```
* Int gcd (int a, int b)
  {
    if (b==0)
    {
     return a;
    }
    else
    return gcd(b, a%b);
  }
```

L.C.M of two Numbers:

The smallest number which is divisible by both the numbers.

```
int lcm(int a, int b)
{
  int res = max(a,b);
  while(True)
  {
    if (res%a==0 && res%b==0)
    {
      return res;
    }
    else
    {
      res++;
    }
```

Suppose, a=4 & b=6

res = 6
res = 7
res = 8
res = 9
res = 10
res = 11
res = 12

efficient soln:

```
int gcd(int a, int b)
{ if (b==0)
  { return a;
  }
  return gcd(b, a%b);
}
int lcm(int a, b)
{ return (a*b)/gcd(a,b);
}
```

* It works on formula, a*b = gcd(a*b) * lcm(a,b)

If a=4, b=6

gcd = 2

lcm = 4x6/2

= 12

Checking of prime Number:

```
bool isPrime(int n)
{
  if(n==1)
  {
    return false;
  }
 for(int i=2;i<n;i++)
 {  if(n%i==0)
    {
      false ;
    }
  }
 return true
```

efficient soln:

```
bool isprime(int n)
{
  if(n==1)
  {
    return false;
  }
for(i=2;i*i≤n;i++)
 {  if(n%i==0)
    {
      True
    }
  }
 return false;
```

most efficient soln:

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

Prime factors:

I/p: n = 12

o/p: 2 2 3

I/p: n = 13

o/p: 13

I/p: n = 315

o/p: 3 3 5 7

Naive solution:

```
void primefactor(int n)
{
   for(int i=2; i<n; i++)
   {
      if( isPrime(i))
      {
         int x = i;
         while(n % x == 0)
         {
            print(i);
            x = x * i;
         }
      }
   }
}
```

```
*void Primefactors(int n)
{
   if(n<=1)
   {
   return 0;
   }
for(i=2; i*i<=n; i++)
{
   while(n%i==0)
   {
      print(i);
      n=n/i;
   }
}
}
```

efficient solution:

① Divisors always appear in pairs

30: (1, 30), (2, 15), (3, 10), (5, 6)

② A number "n" can be written as multiplications of powerd prime factors ⇒ $12 = 2^2 * 3$

$450 = 2^1 * 3^2 * 5^2$

More-efficient solution:-

```
ex: void Primefactors(int n)
    {
      if (n<=1)
      {
       return 0;
       }
      while (n%2==0)
     {
       print(2);
       n=n/2;
      }
    while(n%3==0)
     {  Print(3)
       n=n/3;
      }
for(i=5; i*i<=n; i=i+6)
 {  while(n%i==0)
    {
      Print(i);
      n=n/i;
     }
   while (n%(i+2)==0)
    {
      print(i+2);
      n=n/i+2;
     }
 }
if(n>3)
 print(n);
}
```

All Divisors of a Number:-

I/p: n=15

o/p: 1 3 5 15

I/p: n=100

o/p: 1 2 4 5 10 20 50 100

I/p: n=7

o/p: 1 7

Naive solution:-

```
void printDivisor(int n)
{
  for(int i=1; i<=n; i++)
  {
    if (n%.i==0)
    {
      Print(i);
    }
  }
```

efficient solution:-

```
Void printDivisor(int n)
{ int i;
for(i=1; i*i<=n; ;i++)
 { if (n%.i==0)
   {
     Print(i);
   }
 }
for( ; i>=1; i--)
{
  if(n%.i==0)
  {
    Print(n/i);
  }
}
```

computing Power

$n \geq 0$

I/p: x=2, n=3

o/p: 8 // 2^3

I/p: x=3, n=4

o/p: 81 // 3^4

```
int Power(int x, int n)
{
  int res = 1;
 for(int i=0; i<n; i++)
  {
    res = res * x;
  }
 return res;
}
```

Bitwise operators:-

AND - & → If two Inputs are "1" it produces output as "1". In all other cases it produces output as zero.

OR - | → If any of Inputs is "1" it produces out put as "1". all other cases it produces zero.

XOR - ^ → If two Inputs are different it produces "1". other cases it produces zero.

Bitwise AND:

x = 3
y = 6

Binary Representation:

x = 0000-----0011
y = 0000-----0110

0000-----0010 → 2

Bitwise OR:

x = 3
y = 6

x = 00-----0011
y = 00-----0110

00-----0111 → 7

Bitwise XOR:

x = 3
y = 6

x = 0000-----0011
y = 0000-----0110

00-------0101 → 5 → output

left-shift operator:

x = 3

x: 0000------011

x<<1: 000-----0110 → o/p: 6
↳ Zero should be added at end

x<<2: 0000----0110 0 → o/p: 12

* It always works on the formula: $x * 2^y$

Right-shift operator:

x = 33

x: 0000------0100001

x>>1:

00000-----010000 → o/p: 16

Zero should be added at starting

Note: It always works on formula : $\left\lfloor \frac{x}{2^y} \right\rfloor$

Bitwise Not: (~)

X = 1

x = 0000-----01

~~~X = 1000-----~~~

~X = 1111----- 10

*In this all zero's becomes "1" and all one's becomes zero's.

check if K-th bith is set or Not;

I/p: n = 5, K = 1 ⇒ 0000-----0101

O/p: Yes

I/p: n = 8, K = 2 ⇒ 0000-----1000

O/p: No

I/p: n = 0, K = 3

O/p: No.

K ≤ No. of bits in bincory Representation.

method 1: (left shift)

```
void Kthset (int n, int K)
{
  if (n & (1<<(K-1)) != 0)
  {
    print("Yes");
  }
  else
  {
    print("No");
  }
```

I/p: n = 5, K = 3

n = 000-----0101

1<<(k-1) = 000----0100

4 = 000-----0100
~~~

method 2: (Right Shift)

```
void Kthset(int n, int k)
{
  if(((n>>(k-1))&1)==1)
   {
     Print("Yes");
   else
   {
     Print("No");
   }
}
```

Count the Set bits of a Number:

Time complexity: Θ(No. of bits)

```
int countset(int n)
{
  int res=0;
  while(n>0)
  {
    if(n%2!==0) <=> if((n&1)==1) <=> res=res+(n&1);
    {
      reset++;
    }
    n=n/2; <-------> n=n>>1
  }
}
```

```
{ res++
}
```

* Brian kerningam's Algorithm:

```
int countset(int n) // n=40
{ int res=0;
  while(n>0)
  {
    n=(n&(n-1));
    res++;
  }
  return res;
}
```

0000----101000	n=40
0000----100111	n-1=39
000----100000	n=32
0000---100000	n=32
0000---011111	n-1=31
0000----000000	n=0

Time complexity: Θ(set bit count).

Lookup-Table method:

Power of two:

I/p: 4

o/p: Yes

I/p: 6

o/p: No

method 1:

```
bool ispow2(int n)
{
   if(n==0)
   {
    return false;
   }
while(n!=1)
{
   if(n%2!=0)
   {
    return false;
    n=n/2;
   }
}
return true;
```

method 2:

```
bool isPow2(int n)
{
   if(n==0)
   {
    return false;
   }
if((n&(n-1))==0)
{
   print("True");
}
else
{
   false;
}
```

ex: 1:

n = 4 : 000---0100

n-1=3: 000---0011

:000---0000 = 0

ex: 2:

n=6: 000---1010

n-1=5; 000---1001

000---1000 ≠ 0

Find the odd occuring Number:

I/p: arr[] = {4, 3, 4, 4, 4, 5, 5}

o/p: 3

I/p: arr[] = {8, 7, 7, 8, 8}

o/p: 8

method 1:

```
for(int i=0; i<n; i++)
{
  int count=0;
 for (int j=0; j<n; j++)
  { if(arr[i]==arr[i])
    {
      count++;
    }
  }
  if (count%2!=0)
  { Print (arr[i]);
  }
```

Find two odd appearing numbers:

I/p: arr[] = {3, 4, 3, 4, 5, 4, 4, 6, 7, 7}

o/p: 5 6

I/p: arr[] = {20, 15, 20, 16}

o/p: 15, 16

method 1:

```
Void oddappearing(int arr[], int n)
{int count=0
  for(int i=0; i<n; i++)
  {
    for(int j=0; j<n; j++)
    {
      if (arr[i]==arr[j])
      {
        count++
      }
    }
  }
  if (count%2!=0)
  {
    Print (arr[i]);
  }
 }
}
```

method 2:

```
void oddappearing(int arr[], int n)
{
 int XOR=0;
 int res1=0;
 int res2=0;
for(int i=0;i<n;i++)
{ XOR = XOR ^ arr[i];
  }
 int sn = XOR & ~(XOR-1);
for(int i=0; i<n; i++)
{ if ((arr[i] & sn)!=0)
   {
     res1 = res1 ^ arr[i];
   }
  else
  { res2 = res2 ^ arr[i];
  }
Print(res1, res2);
```

working:

XOR = {3^4^3....7^7}

= 8^32

8 : 00---1000

32 : 00-100000

XOR : 00---101000

: 40

XOR-1 : 00---0100111

~(XOR-1) : 11---1011000

&

00--01000

Powerset using Bitwise Operator:

I/p: s="ab"

O/p: "", "a", "b", "ab"

counter(Decimal)	counter(Binary)	subset
0	000	""
1	001	"a"
2	010	"b"
3	011	"ab"

value vary from 0 to 2^n-1

method 1:

```
Void PrintPowerset(string str)
{ int n = str.length(); //n=3
 int Powsize = pow(2,n); //8
 for( int counter=0; counter<Powsize; counter++)
 {
  for(int j=0; j< n; j++)
  {
   if(counter & (1<<j) !=0)
   {
    Print(str[j])
   }
  }
  Print("\n");
 }
```

I/p = "abc"

counter	subset
000	""
001	"a"
010	"b"
011	"ab"
100	"c"
101	"ac"
110	"bc"
111	"abc"

Print n to 1 using Recursion:

I/p: n = 5
o/p: 5 4 3 2 1

I/p: 2
o/p: 2 1

$n \geq 1$

```
ex: void PrintNto1(int n)
    {
      if(n==0)
      { return 0;
      }
     cout<<n<<" ";
     PrintNto1(n-1);
    }
int main()
{ int n=4;
 cout<< PrintNto1(n);
 return 0;
}
```

Print 1 to n using Recursion:

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

I/p: 5
o/p: 1 2 3 4 5

```
void Print1toN(int n)
{
   if(n==0)
   { return;
   }
  Print1toN(n-1)
 cout<<n<<" ";
}
int main()
 {
    int n=4;
cout<<Print1toN(n);
 return 0;
  }
```

Tail Recursion: A Recursive fun. is said to be tail Recursive when the parent fun. has nothing to do when child fun. has finished.

```
ex: void fun (int n)
    {
      start:
        if(n==0)
        {return;
        }
        print(n);
        fun(n-1); // n=n-1, f(n-1) is Replaced by n=n-1 by modern compilers
        goto start; // This is add Internally by compiler
```

Sum of Natural numbers using Recursion:

I/p: n = 2 // 1+2	I/p: n = 4 // 1+2+3+4
o/p: 3	o/p: 10

```
ex: int getsum(int n)
    {
     if(n==0)
     {
      return 0;
     }
    return n+getsum(n-1);
    }
   int main()
   {
    int n=4;
    cout<< getsum(n);
    return 0;
    }
```

```
10
getsum(4); //output:10
  6 -> 4+getsum(3);
    3 -> 3+getsum(2);
      1 -> 2+getsum(1);
        -> 1+getsum(0);
                  0
```

Palindrome check using Recursion:

I/p: abbcbba

o/p: Yes.

I/p: abba

o/p: Yes.

```
ex: bool isPalindrome(string &str, int start, int end)
    {
     if(start >= end)
     {
      return true;
     }
    return (str[start] == str[end] &&
      ispalindrome(str, start+1, end-1)
    }
```

Sum of Digits using Recursion:

I/p: n=253	I/p: n=9987
o/p: 8	o/p: 33

```
ex: int getsum (int n)
 { if (n==0)
   {
    return 0;
    }
  else
   {
    return getsum(n/10)+n%10;
    }
  }
```

Rope cutting Problem:

I/p: n=5, a=2, b=5, c=1

o/p: 5

we make 5 peices of length "1" each

I/p: n=23, a=12, b=9, c=11

o/p: 2

we can make 2 peices of length 12 & 11.

```
ex: int maxPieces (int n, int a, int b, int c)
    {
      if(n==0)
      {return 0;
       }
      if(n<0)
      {
       return -1;
       }
int res = max (maxPieces(n-a,a,b,c), maxPieces(n-b,a,b,c),
               maxPieces(n-c,a,b,c));
  if (res==-1)
   {
    return -1;
   }
                  return res+1;
```

Generate Subsets:-

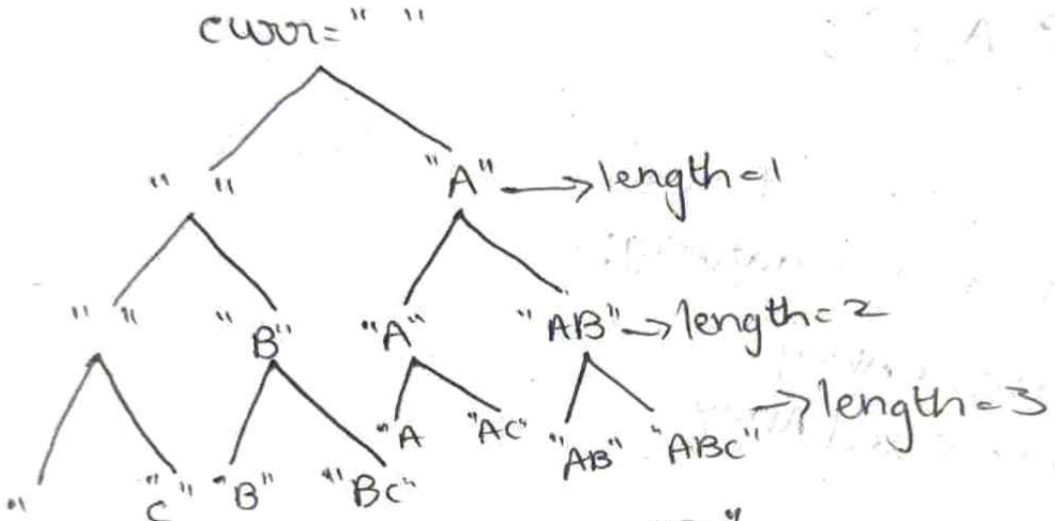
Ip: s = "AB"

dp: "", "A", "B", "AB".

Ip: s = "ABC"

O/p: "", "A", "B", "C", "AB", "BC", "AC", "ABC".

Note: for a string of length "n", there are going to be 2^n subsets



Recursion Tree of "ABC".

```
ex: Void subsets (string s, string curr="", int i=0)
    {
      if (i==s.length())
      {
        cout<<curr;
      }
     subsets(s, curr, i+1);
     subsets (s, curr+s[i], i+1);
    }
```

Tower of Hanoi:

```
ex: void TOH(int n, char A, char B, char C)
    { if(n==1)
      {
      cout<<"move 1 from"<<A<<"to"<<C<<endl;
      return;
      }
    TOH(n-1, A, C, B);
    cout<<"move"<<n<<"from"<<A<<"to"<<C<<endl;
    TOH(n-1, B, A, C);
    }
```

Array: Array is a Data structure which allowed to store multiple elements of same data-type

Types: 1) fixed sized Array
2) Dynamic sized Array

c/c++

* int arr[100] } stack Allocated
* int arr[n] }
* int *arr = new int a[n] } Heap Allocated
* int arr[] = {10, 15, 30, 40}

Dynamic sized Arrays:

Arrays that grew up Automatically according to the user inputs.

C++ : vector
Java : Arraylist
Python : list

Vectors in C++:

Advantages:
* Dynamic size
* Rich library function
* easy to know size
* No need to pass size
* can be returned from a function
* By default Initialized with default values
* we can copy a vector to other.

operations on Array:-

Search (unsorted Array):

I/p: arr[] = {20, 5, 7, 25}
x = 5
o/p: 1

I/p: arr[] = {19, 21, 23, 24}
x = 5
o/p: -1

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

Insert:-

I/p: arr[5] = {5, 10, 20, -, -}
x = 7
pos = 2
o/p: arr[] = {5, 7, 10, 20, -}

I/p: arr[] = {5, 7, 10, 20, -}
x = 3
pos = 2
o/p: arr[] = {5, 3, 7, 10, 20}

ex:
```
int Insert(int arr[], int n, int x,
           int cap, int pos)
{
  if (n==cap)
  {
    return n;
  }
  int index = pos-1;
  for(int i=n-1; i>=index; i--)
  {
    arr[i+1] = arr[i];
    arr[index] = x;
  }
  return (n+1);
}
```

Deletion:

I/p: arr[] = {3, 8, 12, 5, 6}
x = 12

o/p: arr[] = {3, 8, 5, 6, -}

I/p: arr[] = {3, 8, 12, 5, 6}
x = 6

o/p: arr[] = {3, 8, 12, 5, -}

ex: int delete(int arr[], int n, int x)
{
int i;
for(i=0; i<n; i++)
{
if(arr[i] = x)
{
break;
}
}
if(i == n)
return n;
for(int j=i; j<n-1; i++)
{
arr[j] = arr[j+1]
}
return(n-1);
}

Largest element in the Array:

I/p: arr[] = {10, 5, 20, 7}
o/p: 2 // Index of 20

I/p: arr[] = {40, 8, 50, 60}
o/p: 3 // Index of 60

ex: int getlargest(int arr[], n)
{
int res=0;
for(int i=1; i<n; i++)
{
if(arr[i] > arr[res])
{
res = i;
}
}
return res;
}

Second largest Element:

Ip: arr[] = {10, 5, 8, 20}

op: 0 // Index of 10

Ip: arr[] = {20, 10, 20, 8, 12}

op: 4 // Index of 12

```
int Secondlargest (int arr[], int n)
  {
    int res=-1;
   int largest=0;
  for(int i=1; i<n; i++)
   {
     if(arr[i] > arr[largest])
      {
        res = largest;
        largest=i;
      }
     else if (arr[i]!= arr[largest])
      {
        if (res==-1 || arr[i]>arr[res])
         {
           res =i;
         }
      }
    return res;
  }
```

Check if an array is sorted:

Ip: arr[] = {8, 12, 15}

op: Yes.

Ip: arr[] = {8, 10, 10, 12}

op: Yes.

ex:
```
bool isSorted (int arr[], int n)
{
  for (int i=1; i<n; i++)
  {
    if (arr[i]<arr[i-1])
    {
      return false;
    }
  }
return true;
}
```

Reverse an Array:

I/p: arr[] = {10, 5, 7, 30}

o/p: 30, 7, 5, 10

I/p: arr[] = {30, 20, 5}

o/p: 5, 20, 30

ex:
```
void Reverse(int arr[], int n)
{ int low=0;
  int high=n-1;
while (low<high)
{ int temp = arr[low];
  arr[low] = arr[high];
  arr[high] = temp;
  low++;
  high--;
}
}
```

Remove Duplicates from a Sorted Array:

ex: arr[] = {10, 20, 20, 30, 30, 30}
o/p: arr[] = {10, 20, 30, ----}

```
*int remDup(int arr[], int n)
 {
  int temp[n];
  temp[0] = arr[0];
  int res = 1;
 for (int i=1; i<n; i++)
  {  if (temp[res-1] != arr[i])
      { temp[res] = arr[i];
        res++;
      }
  }
  for(int i=0; i<res; i++)
  {
    arr[i] = temp[i];
  }
 return res;
 }
```

```
*int remDup (int arr[], int n)
 {  int res = 1;
 for (int i=1; i<n; i++)
  { if (arr[i] != arr[res-1])
     {  arr[res] = arr[i];
        res++;
     }
  }
 return res;
 }
```

Move all zero's to end:

I/p: arr[] = {8, 5, 0, 10, 0, 20}
o/p: arr[] = {8, 5, 10, 20, 0, 0}

I/p: arr[] = {0, 0, 0, 10, 0}
o/p: arr[] = {10, 0, 0, 0, 0}

I/p: arr[] = {10, 20}
o/p: arr[] = {10, 20}

Naive Solution:

```
ex: void moveToend(int arr(), int n)
  {
    for(i=0;i<n;i++)
    {
      if(arr[i]==0)
      {
        for(int j=i+1;j<n;j++)
        {
          if(arr[j]!=0)
          {
            swap(arr[i],arr[j]);
          }
        }
      }
  }
```

Efficient solution:

```
void moveZeros(int arr[], int n)
{ int count=0;
  for(int i=0;i<n;i++)
    {
      if(arr[i]!=0)
      {
        swap(arr[i], arr[count]);
        count++;
      }
    }
```

Left Rotate an Array by one:

I/p: arr() = {1, 2, 3, 4, 5}

O/p: arr() = {2, 3, 4, 5, 1}.

ex: void leftRotate(int arr[], int n)

```
{
  int temp = arr[0];
for (int i=1; i<n; i++)
{
  arr[i-1] = arr[i];
  }
arr[n-1] = temp;
}
```

Left Rotate an Array by 0 places:

I/p: arr[] = {1, 3, 3, 4, 5}.
d = 2
O/p: arr[] = {3, 4, 5, 1, 2}

I/p: arr[] = {10, 5, 30, 15}
d = 3
O/p: arr[] = {15, 10, 5, 30}

method 1:

```
void leftRotate(int arr[], int n, int d)
{ int temp[d];
  for(int i=0; i<d; i++)
  {
    temp[i] = arr[i];
  }
  for(int i=d; i<n; i++)
  {
    arr[i-d] = arr[i];
  }
  for(int i=0; i<d; i++)
  {
    arr[n-d+i] = temp[i];
  }
}
```

T.C: $\Theta(d+n-d+d)$
$\Rightarrow \Theta(n+d)$
$\Rightarrow \Theta(n)$.

A.S: $\Theta(d)$.

Method 2:

```
void leftRotate(int arr[], int n, int d)
{ reverse(arr, 0, d-1);
  reverse(arr, d, n-1);
  reverse(arr, 0, n-1);
}
void reverse(int arr[], int low, int High)
{ while(low<high)
   { swap(arr[low], arr[high]);
     low++;
     high--;
   }
}
```

T.C: $\Theta(n)$

A.S: $\Theta(1)$.

Leader in an Array:

leader: There is no greater element on the Right side of an element. That element is called leader

I/p: arr[] = {7, 10, 4, 3, 6, 5, 2}

O/p: 10, 6, 5, 2

I/p: arr[] = {10, 20, 30}

O/p: 30.

I/p: arr[] = {30, 20, 10}

O/p: 30. 20 10.

method 1: (Naive) $O(n^2)$

```
Void leader(int arr[], int n)
{
  for(int i=0; i<n; i++)
  {
     bool flag=false;
    for (int j=i+1; j<n; j++)
    {
      if(arr[i]<= arr[j])
      { flag= true;
        break;
      }
    }
  if(flag==false
  {
    cout<< arr[i];
  }.
```

method:2 (efficient) $\Theta(n)$

```
void leader (int arr[], int n)
{ int curr_ldr = arr[n-1];
  Print(curr_ldr);
for (i=n-2; i>=0; i--)
 { if (curr_ldr < arr[i])
   {
    curr_ldr = arr[i];
    Print (curr_ldr);
   }
 }
```

output: 2, 5, 6, 10

Maximum Difference Problem with order:

maximum value of arr[j]-arr[i], such that j>i

Ip: arr[]= {2,3,6,10,6,4,8,1}

Op: 8

Ip: arr[]= {7,9,5,6,3,2}

Op: 2

Naive method:

```
int maxDiff (int arr[], int n)
{ int res= arr[i]- arr[0];
 for (int i=0; i<n-1; i++)
  {
    for (int j=i+1; j<n; j++)
     {
      res =max (res, arr[j]-arr[i]);
  }
 return res;
}
```

efficient soln:

```
int maxDiff(int arr[], int n)
{ int res = arr[1] - arr[0];
  int min_val = arr[0];
  for(j=1; j<n; j++)
  {
    res = max(res, arr[j] - min_val);
    min_val = min(min_val, arr[j]);
  }
  return res;
}
```

Frequencies of an Sorted Array:

I/p: arr[] = {10, 10, 10, 25, 30, 30}

o/p: 10 3
25 1
30 2

```
ex: void Printfreq(int arr[], int n)
{ int freq=1; i=1;
  while(i<n)
  {
    while(i<n && arr[i]==arr[i-1])
    { freq++;
      i++;
    }
    Print(arr[i-1]+" "+freq);
    i++;
    freq=1;
  }
  if(n==1 || arr[n-1]!=arr[n-2])
  { Print(arr[n-1]+" "+1);
```

Stock Buy and sell:

Ip: arr[] = {1, 5, 3, 8, 12}

Op: 13.

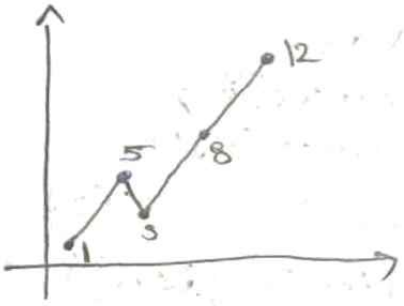
Ip: arr[] = {30, 20, 10}

Op: 0

```
ex: int maxProfit(int Price[], int start, int end)
{
  if (end <= start)
  {
    return 0;
  }
  int profit = 0;
for (int i=start ; i<end ; i++)
{
  for(int j=i+1; j<=end ; j++)
  {
    if (price[j]>price[i])
    {int curr_profit = price[j]-price[j]+maxProfit(price, start, i-1)+
                       maxProfit (price, j+1, end);
      Profit = max(Profit, curr_Profit);
    }
  }
}
return Profit;
}
```

efficient soln:

```
int maxProfit(int Price[], int n)
{ int Profit=0;
for(int i=1; i<n; i++)
{ if (price[i] > price[i-1])
  {
    Profit = Profit + (Price[i]-Price[i-1]);
  }
}
return Profit;
}
```

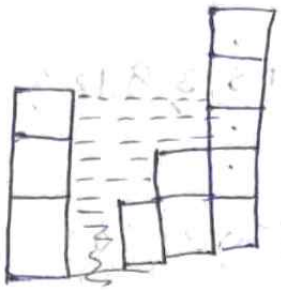


*we have to buy the stock at bottom and sell the stock at peak

Trapping Rain water:

I/p: arr[] = {3, 0, 1, 2, 5}

O/p: 6



Naive:

```
ex: int getwater(int arr[], int n)
  {
   int res=0;
   for(int i=1; i<n-1; i++)
   {
     int lmax = arr[i];
     for (int j=0; j<i; j++)
     {
       lmax = max (lmax, arr[j]);
     int rmax = arr[i];
      }
     for(int j=i+1; j<n; j++)
     {
       rmax = max(rmax, arr[i]);
        res = res+(min(lmax, rmax)-arr[i]);
     }
     return res;
  }
```

→ To find the left maximum

→ To find the Right maximum

efficient:

```
int getwater(int arr[], int n)
 {
  int res=0;
 int lmax[n];
 int rmax[n];
  lmax[0] = arr[0];
 for(int i=1; i<n; i++)
  {
    lmax[i] = max (arr[i], lmax[i-1]);
  }
 rmax [n-1] = arr[n-1];
```

```
for(int i=n-2; i>=0; i--)
 {
  rmax[i] = max (arr[i], rmax[i+1]);
 }
for (int i=0; i<n-1; i++)
 {
  res = res + (min (lmax[i], rmax[i]) - arr[i]);
 }
return res;
}
```

Maximum consecutive 1's in a Binary Array:

Ip: arr[] = {0, 1, 1, 0, 1, 0}

Op: 2

Ip: arr[] = {1, 0, 1, 1, 1, 0, 1, 1}

Op: 4

Naive: O(n²) O(1) A.S

```
int max (int arr[], int n)
{ int res=0;
 for(int i=0; i<n; i++)
 { int curr=0;
  for (int j=i; j<n; j++)
  { if (arr[j]==1)
   {
    curr++;
   }
   else
   { break;
   }
  }
  res = max(res, curr);
 }
return res;
}
```

efficient:

```
*int max(int arr[], int n)
 { int res=0;
for(int i=0; i<n; i++)
 { if (arr[i]=0)
  {
   curr=0;
  }
  else
  {
   curr++;
   res=max(res,curr);
  }
 }
return res;
}.
```

Maximum SubArray:

What is subArray:

Subarray's of {1,2,3} is {1}, {2}, {3}, {1,2}, {2,3}, {1,2,3}

Naive $O(n^2)$:

```
int maxsum (int arr[], int n)
{ int res = arr[0];
for (int i=0; i<n; i++)
{ int curr=0;
 for (int j=i; j<n; j++)
 { curr = curr+arr[i];
   res = max(res, curr);
 }
}
return res;
}
```

Efficient no(n):

```
int maxsum(int arr[], int
{ int res = arr[0];
  int maxEnding = arr[0];
for(int i=1; i<n; i++)
{
maxEnding = max(maxEnding+arr[
          arr[i]);
res = max(res, maxEnding);
}
return res;
}
```

Longest Even odd SubArray:

*maximum length Even-odd subArray:

I/p: arr[] = {10, 12, 14, 7, 8}

o/p: 3

I/p: arr[] = {7, 10, 13, 14}

o/p: 4

Naive: $O(n^2)$:

```
int maxEvenOdd(int arr[], int n)
{ int res=1;
 for(int i=0; i<n; i++)
 { int curr=1;
  for(int j=i+1; j<n; j++)
```

```
{
 if((arr[j]%2==0 && arr[j-1]%2!=0) || (arr[j]%2!=0 &&
  arr[j-1]%2==0)
 {
  curr++;
 }
  else
  {
   break;
  }
 }
  res=max(res,curr);
 }
return res;
}
```

efficient: O(n):

```
int maxEvenOdd (int arr[], int n)
{  int res=1;
   int curr=1;
for(int i=1; i<n; i++)
{if((arr[i]%2==0 && arr[i-1]%2!=0) || (arr[i]%2!=0 &&
 arr[i-1]==0))
{
  curr++;
   res=max(res,curr);
  }
else
{ curr=1;
}
}
return res;
}
```

Maximum Circular SubArray Sum:

i/p: arr[] = {10, 5, -5}

Normal Sub-Arrays { {10}, {5}, {-5}, {10,5}, {5,-5}, {10,5,-5} }

Circular Sub-Arrays { {5,-5,10}, {-5,10}, {-5,10,5} }

I/p: arr[] = {5, -2, 3, 4} = {3, 4, 5}

O/p: 12

Naive: $O(n^2)$:

```
int MaxCircularSum(int arr[], int n)
{ int res = arr[0];
for (int i=0; i<n; i++)
{ int curr_max = arr[i];
 int curr_sum = arr[i];
 for(j=1; j<n; j++)
 { int index = (i+j)%n;
  curr_sum = curr_sum + arr[index];
  curr_max = max (curr_max, curr_sum);
 }
 res = max(res, curr_max);
}
return res;
}
```

efficient solution;

```
int normalMaxSum(int arr[], int n)
{ int res = arr[0];
int maxEnding = arr[0];
  for(int i=1; i<n; i++)
  {
  MaxEnding = max(arr[i], maxEnding+arr[i]);
      res = max(res, maxEnding);
  }
  return res;
  }
int overallMaxSum(int arr[], int n)
{ int max_normal = normalMaxSum(arr, n);
  if (max_normal < 0)
  {
   return max_normal;
  }
  int arr_sum = 0;
  for(int i=0; i<n; i++)
   {
     arr_sum = arr_sum + arr[i];
     arr[i] = -arr[i];
    }
 int max_circular = arr_sum + normalMaxSum(arr, n);
 return max(max_normal, max_circular);
 }
```

Majority Element: An element which appears more than $n/2$ times in the Array.

Ip: arr[] = {8, 3, 4, 8, 8}

Op: 0 or 3 or 4.

If: arr[7] = {3, 7, 4, 5, 6, 2, 1, 9}

dpi: -1 (no majority)

Naive: O(n²)

```
int findMajority(int arr[], int n)
{
  for(int i=0; i<n; i++)
  { int count=1;
    for(int j=i+1; j<n; j++)
    {
      if (arr[i]=arr[j])
      {
        count++;
      }
    }
    if count (count > n/2)
    {
      return i;
    }
  }
  return -1;
}
```

Efficient: O(n)

*using Moore's voting Algorithm

```
int findMajority (int arr[], int n)
{
  int res=0; count=1;
  for(i=1; i<n; i++)
  {
    if( arr[res]=arr[i])
    {
      count++;
    }
    else
    {
      count--;
    }
    if(count==0)
    {
      int res=i;
      int count=1;
    }
```

```
count=0;
for(int i=0;i<n;i++)
 {
  if(arr[res]==arr[i])
  {
    count++;
  }
  if(count<=n/2)
  {
   res=-1;
  }
 return res;
 }
```

Minimum consecutive flips:

```
void printFlips(bool arr[], int n)
{
  for(int i=1;i<n;i++)
  {
    if(arr[i]!=arr[i-1])
    {
      if(arr[i]!=arr[0])
      {
        cout<<"from"<<i<<"to";
      }
      else
      {
        cout<<(i-1)<<endl;
      }
    }
  }
  if(arr[n-1]!=arr[0])
  {
    cout<<(n-1)<<endl;
  }
```

arr[] = {0, 0, 1, 1, 0, 0, 1, 1, 0, 1}

o/p: from 2 to 3
from 6 to 7.

Window Sliding Technique:

Given an array of integers and a number, K, find the maximum sum of K consecutive elements

I/p: arr[] = {1, 8, 30, -5, 20, 7}, K = 3

O/p: 45

I/p: arr[] = {5, -10, 6, 90, 3}, K = 2

O/p: 96

Naive: $O(n^2)$

```
int max_sum = INT_MIN;
for (int i=0; i+k-1<n; i++)
{
    int sum=0;
  for(int j=0; j<k; j++)
  {
     sum += arr[i+j];
   max_sum = max(sum, max_sum);
  }
}
return max_sum;
```

efficient: $O(n)$

```
int curr_sum=0;
for(int i=0; i<k; i++)
{
  curr_sum += arr[i];
  int max_sum = curr_sum;
}
for(int i=k; i<n; i++)
{
  curr_sum += (arr[i] - arr[i-k]);
  max_sum = max(max_sum, curr_sum);
}
return max_sum;
```

I/p: arr[] = {1, 8, 30, -5, 20, 7}
K = 3
O/p: 45

*Given an unsorted Array of non-negative Integer's. Find if there is a SubArray with given sum?

I/p: arr[] = {1, 4, 20, 3, 10, 5} (indices 0 1 2 3 4 5)
sum = 33

o/p: Yes

Naive:

```
for(int i=0; i<n; i++)
{ int sum=0;
  for(int j=i; j<n; j++)
  {
    sum += arr[j];
  }
  if(sum == given sum)
  {
    return true;
  }
  else
  {
    return false;
  }
  }
}
```

*Return the SubArray with sum equal to the given sum

```
bool Subsum(int arr[], int n, int sum)
{ int curr_sum = arr[0], s=0;
  for (int e=1; e<n; e++)
  {
    while(curr_sum > sum && s<e-1)
    {
      curr_sum -= arr[s];
      if (curr_sum == sum)
      {
        return true;
      }
      if(e<n)
      {
        curr_sum += arr[e];
      }
    }
  }
  return (currsum == Sum); }
```

Prefix sum:

Given a fixed array and multiple quires of following types on the Array, how to efficiently perform these quires

Ip: arr[] = {2, 8, 3, 9, 6, 5, 4}

Queries: getsum(0, 2)
getsum(1, 3)
getsum(2, 6)

Approach:

* first build a pre-fix sum Array from given original Array

int prefix-sum[] = {2, 10, 13, 22, 28, 33, 37};

// code to build pre-fix sum Array

```
=> int prefix sum[n];
   prefix-sum[0] = arr[0];
  for(int i=1; i<n; i++)
  { prefix-sum[i] = prefix-sum[i-1] + arr[i];
   }
```

```
* Int getsum(int prefix-sum[], int l, int r)
  { if(l != 0)
    {
  return prefix_sum[r] - prefix-sum[l-1];
    }
    else
    {
    return prefix_sum[r];
     }
    }
```

2) Given an array of integers, find if it has an equilibrium point?

equilibrium Point: the point in Array in which sum of left elements and right elements are equal

I/p: arr[] = {3, 4, 8, -9, 20, 6}
sum = 6 sum = 6
equilibrium point

o/p: Yes.

I/p: arr[] = {4, 2, -2}

o/p: Yes

Naive (O(n²)):

```
for(int i=0;i<n;i++)
{ int l-sum=0, r-sum=0;
 for(int j=0; j<i; j++)
  {
   l-sum += arr[j];
  }
 for(int k=i+1; k<n; k++)
  {
   r-sum += arr[k];
  }
 if(l-sum == r-sum)
  {
  return True;
  }
}
return false;
```

efficient (O(n)):

```
bool iseqPoint(int arr[], int n)
{ int sum=0;
 for(int i=0; i<n; i++)
 {
  sum += arr[i];
 }
 int l-sum=0;
for(i=0; i<n; i++)
{ if(l-sum == sum - arr[i])
  {return true;
  }
  l-sum += arr[i];
  sum -= arr[i];
}
return false;
```

Searching

Binary search:

I/p: arr[] = {10, 20, 30, 40, 50, 60}, X = 20

o/p: 1

I/p: arr[] = {5, 15, 25}, x = 25

o/p: 2

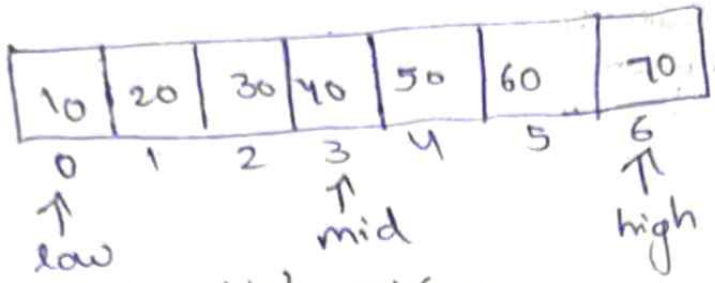
I/p: arr[] = {10, 10}

o/p: 0 (or) 1.

I/p: arr[] = {10, 15}, X = 20

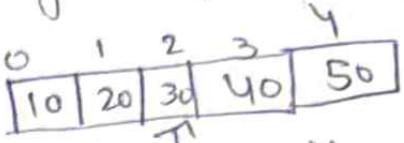
o/p: -1

*If Array is sorted in increasing order

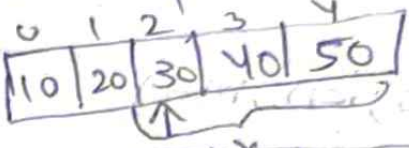


$mid = \frac{low + high}{2} = \frac{0+6}{2} = 3$

case 1: (arr[mid] == x), x=30

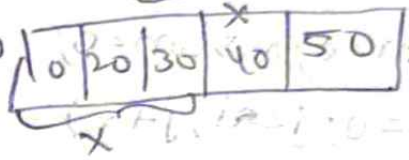


case 2: (arr[mid] > x), x=10



⇒ high = mid-1;

case 3: (arr[mid] < x), x=50



⇒ low = mid+1;

ex:

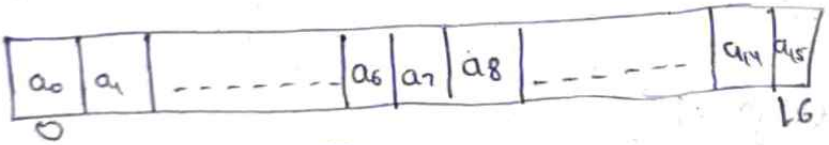
```
int binarySearch(int arr[], int n, int x)
{ int low=0;
  int high=n-1;
  while(low<=high)
  {int mid= low+high/2;
   if(arr[mid]==x)
   {
    return mid;
   }
  else if(arr[mid]>x
   {
     high=mid-1;
   }
  else
    {
     low=mid+1;
    }
  }
}
```

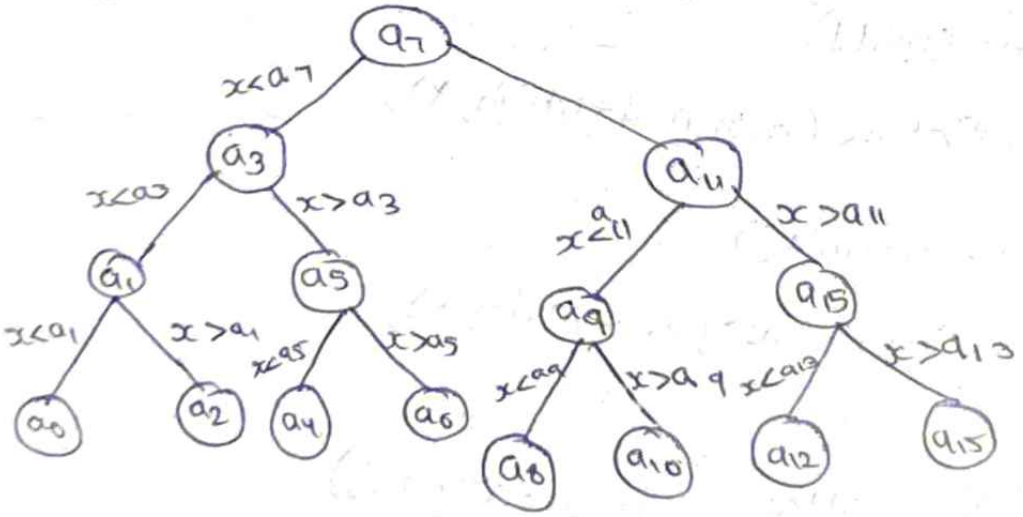
Recursive Binary Search:

```
int bsearch(int arr[], int low, int high, int x)
{
  if (low>high)
  {
   return -1;
   }
  else
   {int mid= low+high/2;
   if(arr[mid]==x)
   {
    return mid;
    }
   else if(arr[mid]>x)
   {
    return bsearch(arr, low, mid-1, x);
   else
   {
   return bsearch(arr, mid+1, high, x);
    }
   }
}
```

Complete Analysis of Binary search:

All possible execution paths for Successful Searches





Time complexity of Binary Search: $O(\log n)$

* for unsuccessful search,

T.C is O(logn)

Index of first occurance;

arr[] = {5, 10, 10, 15, 15}, X = 10

o/p: 1

Naive: O(n)

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

Time: O(n)
Aux. space: O(1)

Approach 2: Recursive Binary search.

```
int firstocc(int arr[], int low, int high, int x)
{  if(low>high)
   {
    return -1;
    }
   int mid = low+high/2;
   if(x>arr[mid])
   {
    return firstocc(arr, mid+1;high, X);
    }
   else if(x<arr[mid])
    {
    return firstocc(arr, low, mid-1, X);
    }
   else
    {
     if(mid==0||arr(mid-1]!=arr[mid])
     {return mid;
      }
     else
   } { return firstocc(arr, low, mid-1, x);
```

Approach 3: Iterative Binary search.

```
int firstocc(int arr[], int n, int x)
{
  int low=0, high=n-1;
  while(low<=high)
  {
    int mid=low+high/2;
    if(arr[mid]>x)
    {
      high=mid-1;
    }
    else if(arr[mid]<x)
    {
      low=mid+1;
    }
    else
    {
      if(mid==0||arr[mid-1]!=arr[mid])
      {
        return mid;
      }
      else
      {
        high=mid-1;
      }
    }
  }
  return -1;
}
```

Index of last occurance:

Ip: arr[]={10, 15, 20, 20, 40, 40}, x=20

Op: 3

Ip: arr[]={5, 8, 8, 10, 10}

Op: 4

Recursive Binary Search:

arr[] = {5, 10, 10, 10, 10, 20, 20}
0 1 2 3 4 5 6

```
int lastocc(int arr[], int low, int high, int n)
{
  if (low > high)
  {
   return -1;
  }
  if(arr[mid] > x)
  {
   return lastocarr(arr, low, mid-1, n);
  }
  else if (arr[mid] < x)
  {
   return lastocc(arr, mid+1, high, n);
  }
  else
  {
   if (mid == n-1 || arr[mid] != arr[mid+1])
   {
    return mid;
   }
   else
   {
    return lastocc (arr, mid+1, high, n);
   }
  }
}
```

Iterative Binary Search:

```
int lastocc(int arr[], int n, int x)
{ int low = 0, high = n-1;
 while(low <= high)
 {
  int mid = low+high/2;
  if( arr[mid] < x)
  {
   low = mid+1;
  }
  else if(arr[mid] > x)
  {
   high = mid-1;
  }
  else
  {
  if (mid != n-1 || arr[mid] != arr[mid
  {
   return mid;
  }
  else
  {
   low = mid+1;
  }
 }
return -1;
}
```

Count occurance in Sorted Array:

I/p: arr[] = {10, 20, 20, 20, 30, 30}, x = 20

O/p: 3

I/p: arr[] = {10, 10, 10, 10, 10, 5}, x = 10

O/p: 5

I/p: arr[] = {5, 15, 20, 20, 20}, x = 9

O/p: 0

```
=> int countOcc(int arr[], int n, int x)
   {
     int first = firstOcc(arr, n, x);
     if(first == -1)
     {
       return 0;
     }
     else
     {
     return (lastOcc(arr, n, x) - first + 1);
     }
```

Count 1's in a Sorted Binary Array:

I/p: arr[] = {0, 0, 0, 1, 1, 1, 1}

O/p: 4

I/p: arr[] = {1, 1, 1, 1, 1}

O/p: 5

```
=> int countOnes(int arr[], int n)
{I})  {
      int low = 0, high = n-1;
      while(low <= high)
      {int mid = low + high/2;
       if(arr[mid] == 0)
       {
         low = mid + 1;
       }
       else
       {if(mid == 0 || arr[mid-1] != arr[mid])
        {return (n - mid);
        else {
         high = mid - 1; } return 0; }
```

Square Root:

I/p: x=4	I/p: x=14	I/p: x=25
o/p: 2	o/p: 3	o/p: 5

Naive:

```
int squareRoot(int x)
{
  int i=1;
  while(i*i<=x)
  {
    i++;
  }
 return(i-1);
}
```

T.C : $\Theta(\sqrt{x})$

Efficient solution: using Binary search

```
int sqrtFloor(int x)
{ int low=1, high=x, ans=-1;
 while(low<=high)
 {
   int mid=(low+high)/2;
   int msq=mid*mid;
   if(msq==x)
   {
     return mid;
   }
   else if(msq>x)
   {
     high=mid-1;
   }
   else
   {
     low=mid+1;
     ans=mid;
   }
 }
 return ans;
}
```

Finding element in a infinite sized sorted Array:

arr[] = {1, 10, 200, ----}, x = 15

```
int search(int x, int arr[])
{ int i=0;
  while(true)                    T.C : O(Position)
   {
     if(arr[i]==x)
     {
      return i;
      }
     if(arr[i]>x)
     {
      return -1;
     }
   }
}
```

efficient:

```
int search (int arr[], int x)
{ if(arr[0]=x)
  {
   return 0;
   }
  int i=1;
 while(arr[i]<x)
 {
   i=i*2;
  }
 if (arr[i]==x)
 {
   return i;
   }
  return binarysearch(arr, x, i/2+1, i-1);
```

Searching the element in a Sorted & Rotated Array

arr[] = {10, 20, 40, 60, 5, 8}

x = 5

```
int search (int arr[], int n, int x)
{
  int low=0, high=n-1;
  while(low<=high)
  {
    int mid = low+high/2;        -> normal Binary Search
    if(arr[mid]==x)
    {
      return mid;
    }
    if (arr[low]<arr[mid])
    {
      if (x>=arr[low] && x<arr[mid])
      {
        high = mid-1;
      }
      else
      {
        low = mid+1;
      }
    }                            -> left half sorted
    else
    {
      if(x>arr[mid] && x<=arr[high])
      {
        low = mid+1;
      }
      else
      {
        high=mid-1;
      }
    }                            -> right half sorted
  }
  return -1;
}
```

Find a Peak element:

*Not smaller than neighbours (both right and left)

I/p: arr[] = {5, 10, 20, 15, 7}

o/p: 20

I/p: arr[] = {10, 20, 15, 5, 23, 90, 60}

o/p: 20 (or) 90

I/p: arr[] = {80, 70, 90}

o/p: 80 (or) 90

Naive:

```
int getPeak(int arr[], int n)
{ if(n==1)
  {
   return arr[0];
  }
 if(arr[0]>=arr[1])
 {
  return arr[0];
 }
 if(arr[n-1]>=arr[n-2])
 {
  return arr[n-1];
 }
 for(int i=1;i<n-1;i++)
  {
   if(arr[i]>=arr[i-1]&&arr[i]<=arr[i+1])
   {
    return arr[i];
   }
  }
}
```

efficient:

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

```
if(mid>0 && arr[mid-1]>=arr[mid])
{ high=mid-1;
  }
else
  {
   low=mid+1;
  }
return -1;
}
```

Two-Pointer Approach:

1) Given an unsorted array and a number 'x', we need to find if there is a pair in the Array with sum equals to 'x'.

Eg: I/p: arr[] = {3,5,9,2,8,10,11}, x=17

O/p: Yes

there is a pair of (9,8) which sum is 17

Naive:

```
bool TwoPointer(int arr[], int x, int n)
{ for(int i=0; i<n; i++)
  {
    for(j=i+1; j<n; j++)
    {
    if(arr[i]+arr[j]==x)
     { return true;
     }
    }
  }
return false;
}
```

2) Given a sorted Array and a sum, find if there is a pair with given sum?

Ans: Here were using Two-pointer Approach.

```
bool isPair(int arr[], int n, int x)
{ int left=0, right=n-1;
  while(left<right)
  {
    if(arr[left]+arr[right]==x)
    {return True;
    }
    else if(arr[left]+arr[right]>x)
    { right--;
    }
    else
    { left++;
    }
  }
 return false;
}
```

3) Given a sorted Array and a sum, find if there is a triplet with given sum?

Ans: arr[] = {2,3,4, 8, 9, 20, 40}, x=32

o/p: Yes

```
for(int i=0; i<n; i++)
{
  for(int j=i+1; j<n; j++)
  {
    for(int k=j+1; k<n; k++)
    {
    if(arr[i]+arr[j]+arr[k]==x)
    {
      return true;
    }
  }
}
return false;
```

Two pointer Approach Popular qn's:

1) count pairs with given sum
2) count triplets with given sum
3) find if there is a triplet a, b, c such that $a^2+b^2=c^2$

Median of two sorted Arrays:

I/p: arr1[] = {10, 20, 30, 40, 50}
arr2[] = {5, 15, 25, 35, 45}

O/p: 27.5 // {5, 10, 15, 20, 25, 30, 35, 40, 45, 50} // median of Index 4,5 gives 27.5

I/p: arr1[] = {1, 2, 3, 4, 5, 6}
arr2[] = {10, 20, 30, 40, 50}

O/p: 6.0 // {1, 2, 3, 4, 5, 6, 10, 20, 30, 40, 50}

Naive: $O((n_1+n_2)*\log(n_1+n_2))$

n_1 = size of array 1
n_2 = size of array 2

1) create an array temp[] of size (n_1+n_2)
2) copy elements of $a_1[]$ and $a_2[]$ into temp[]
3) Sort temp[].
4) If (n_1+n_2) is odd, then return middle of temp.
5) else return avg. of middle two elements

Repeating element:

I/p: arr[] = {0, 2, 1, 3, 2, 2}
O/p: 2

I/p: arr[] = {1, 2, 3, 0, 3, 4, 5}
O/p: 3

I/p: arr[] = {0, 0}
O/p: 0

Super Naive: $O(n^2)$

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

Naive solution: $O(n \log n)$

1) Sort the Array
arr[] = {0, 1, 2, 2, 2, 3}

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

Efficient Approach: O(n) time and O(n) space.

arr[] = {0, 2, 1, 3, 2, 2}

1) create a boolean array of size n-1.

visited[] = {F, F, F, F, F, F}

2) for (int i=0; i<n; i++)
{
if (visited [arr[i]])
{
return arr[i];
}
visited[arr[i]] = true;
}

* int find Repeating (int arr[], int n)
{
int slow = arr[0]+1;
int fast = arr[0]+1;
do {
slow = arr[slow]+1;
fast = arr[arr[fast]+1]+1;
}while (slow != fast)
slow = arr[0]+1;
while(slow != fast)
{
fast = arr[fast]+1;
slow = arr[slow]+1;
}
return slow-1;
}

Allocate minimum no. of Pages:

I/p: arr[] = {10, 20, 30, 40}, k=2 , Here, k is no. of students and 10, 20, 30 are pages in a Book

O/p: 60

I/p: arr[] = {10, 20, 30}, k=1

O/p: 60.

Sorting

Stability in sorting Algorithms:

arr[] = {("Anil", 50), ("Aryan", 80), ("Piyush", 50), ("Ramesh", 80)}

⇒ By using stable sorting Algorithm to sort the array in increasing order of marks, If any two have same marks it will go with Alphabetical order

arr[] = {("Anil", 50), ("Piyush", 50), ("Aryan", 80), ("Ramesh", 80)}

↳ stable

arr[] = {("Piyush", 50), ("Anil", 50), ("Ramesh", 80), ("Aryan", 80)}

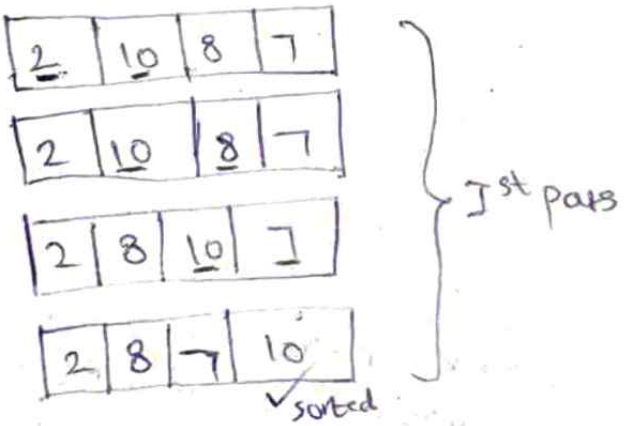
↳ unstable

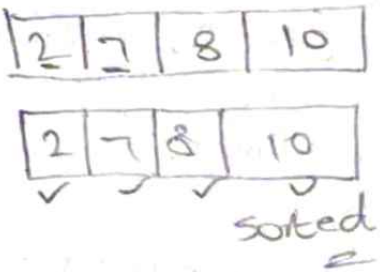
By using unstable sorting Algo. It may be changes or remain same as stable Algo.

* Stable sort Examples: Bubble sort, Insertion sort, merge sort

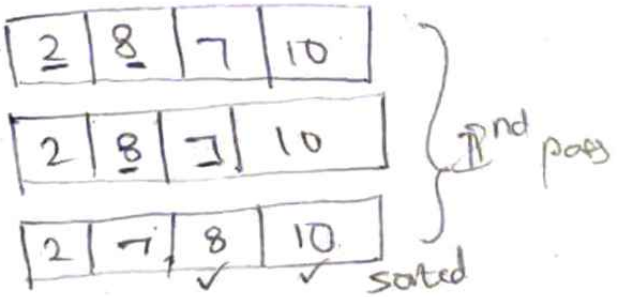
* Unstable sort example: Selection sort, Quick sort, Heap Sort

Bubble Sort:





* So, "n-1" passes are required to sort an array of size "n"



```
ex: void bubblesort (arr, n)
{
 for (int i=0; i<n-1; i++)
  {
   for(int j=0; j<n-i-1; j++)
    { if (arr[j] > arr[j+1])
      {
       swap (arr[i], arr [j+i]);
      }
    }
  }
}
```

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

selection sort:

*Inplace Algorithm (No extra space req. for Array for sorting).

arr[] =

INF	INF	INF	INF	INF	INF
10	5	8	20	2	18

→ find minimum element in given array and copied to 1st position in temp. array

→ Replace that position with Infinity

temp[] =

2	5	8	10	18	20	

*It always takes $O(n^2)$ time in all cases.

Naive:

```
void selection sort(int arr, n)
{
 int temp[n];
 for(int i=0; i<n; i++)
 {
  int min_ind=0;
  for(int j=1; j<n; j++)
   {
    if(arr[j]<arr[min_ind])
     {
      min_ind=j;
     }
   temp[i]=arr[min_ind]
   arr[min_ind]=INF; }
  }
 for(int i=0; i<n; i++)
 { arr[i]=temp[i]; // coping elements in temp array to original array
 }
}
```

In-place Implementation:

*To avoid taking an temp. array we can do more optimisation

```
Void selectSort(int arr[], int n)
{ for(int i=0; i<n-1; i++)
  min_ind = i;
for (int j= i+1; j<n; j++)
  { if (arr[j]<arr[min_ind])
    {
      min_ind = j;
    }
  swap(arr[min_ind], arr[i]);
  }
}
```

arr[] =

10	5	8	20	2	18
0	1	2	3	4	5

i=0, min_ind=4

2	5	8	20	10	18

i=1; min_ind=1

2	5	8	20	10	18

i=2; min_ind=2

2	5	8	20	10	18

i=3; min_ind=4

2	5	8	10	20	18

i=4; min_ind=5

2	5	8	10	18	20

Insertion Sort:

-> $O(n^2)$ -> worst case
-> Inplace and stable
-> Used in practice for small sized Arrays (Tim Sort and Intrasort)
-> $\Omega(n)$ in Best case

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

arr[] = {20, 5, 40, 60, 10, 30}

T.C:

Best case: Already sorted [10, 20, 30, 50] $\Rightarrow \Theta(n)$

Worst case: Reverse sorted

50	30	20	10

$\Rightarrow \Theta(n^2)$

∴ Generally $O(n^2)$

Merge sort:

① Divide and conquer Algorithm (Divide, conquer and merge)
② Stable Algo.
③ $\Theta(n \log n)$ Time and $O(n)$ Aux. space
④ well suited for linked lists. works in $O(1)$ Aux. space
⑤ Used in external sorting.
⑥ In general for Arrays, Quicksort performs better than mergesort

Merge two sorted Arrays:

Ip: arr[] = {10, 15, 20}
arr[] = {5, 6, 6, 15}
o/p: 5, 6, 6, 15, 10, 15, 20

Ip: arr[] = {1, 1, 2}
brr[] = {3}
o/p: 1, 1, 2, 3

Naive:

```
void merge(int a[], int b[], int m, int n)
{ int c[m+n];
for(int i=0; i<m; i++)
  {
    c[i]=a[i];
  }
for(int i=0; i<n; i++)
  {
    c[m+i]=b[i];
  }
sort(c, c+m+n);
for(int i=0; i<m+n; i++)
{
  cout<<arr[i]<<" ";
}
}
```

a[] = {10, 15, 20, 20}, m = 4
b[] = {1, 12}, n = 2

After 1st loop:
c[] = {10, 15, 20, 20, --}

After 2nd loop:
c[] = {10, 15, 20, 20, 1, 12}

After sorting:
c[] = {1, 10, 12, 15, 20, 20};

T.C: $O((m+n) * \log(m+n))$
Aux. space: $O(m+n)$;

Efficient;

```
void merge (int a[], int b[], int m, int n)
{ int i=0;
  int j=0;
 while (i<m && j<n)
 { if (a[i]<=b[j])
   {
    cout<<a[i]<<" ";
     i++;
    }
   else
    {
    cout<<b[j]<<" ";
    j++;
    }
  }
 while(i<m)
 {cout<<a[i]<<" ";
   i++;
  }
 while (j<n)
  {cout<<b[j]<<" ";
  j++;
  }
}
```

Time complexity: O(m+n)

Merge function of Merge sort:

I/p: a[] = {10, 15, 20, 11, 30} * low to mid is sorted
* mid+1 to high is sorted

low = 0
mid = 2
high = 4

o/p: a[] = {10, 11, 15, 20, 30}

I/p: a[] = {5, 8, 12, 14, 7}

low = 0
mid = 3
high = 4

o/p: a[] = {5, 7, 8, 12, 14}

```
void merge (int a[], int low, int mid, int high)
{ int n1 = mid-low+1;
  int n2 = high-mid;
  for(int i=0; i<n1; i++)
  { left[i] = a[low+i];
  }
  for(int i=0; i<n2; i++)
  {
    right[i] = mid+i+1;
  }
 int i=0, j=0, k=0;
 while (i<n1 && j<n2)
 { if (left[i]<=right[i])
   { a[k] = left[i];
     i++, k++;
   }
   else
   { a[k] = right[j];
     k++, j++;
   }
 }
 while(i<n1)
 { a[k] = left[i];
   i++, k++;
 }
 while (j<n2)
 { a[k] = right[j];
   j++, k++;
 }
```

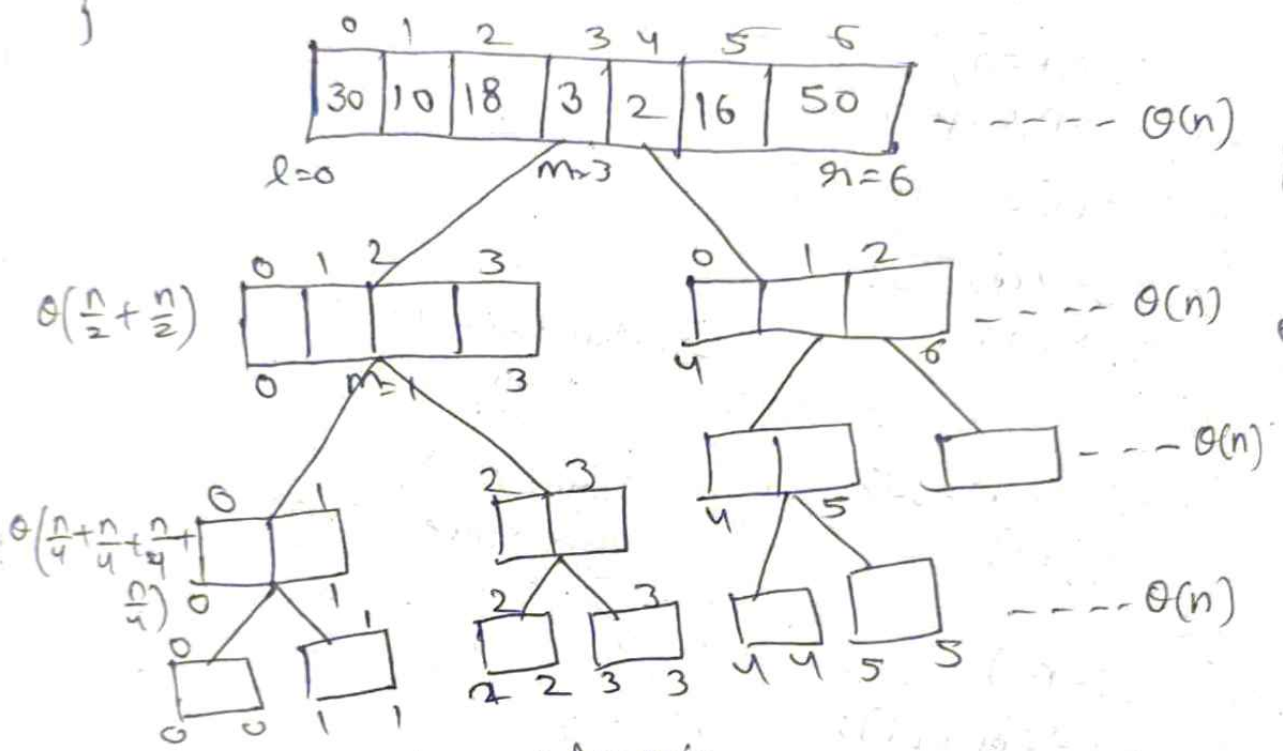
→ setting up the Auxiliary Array's.

→ standard merge logic

Time : $\Theta(n)$
Aux. space : $\Theta(n)$

Merge sort Analysis:

```
Void Mergesort (int arr[], int l, int r)
{ if (r > l)
  { int m = l + (r-l)/2; // (l+r)/2
  mergesort (arr, l, m);
  mergesort (arr, m+1, r);
  merge(arr, l, m, r);
  }
}
```



Intersection of Two Sorted Arrays:

I/p: a[] = {3, 5, 10, 10, 10, 15, 15, 20}
b[] = {5, 10, 10, 15, 30}

O/p: 5 10 15

I/p: a[] = {1, 1, 3, 3, 3}
b[] = {1, 1, 1, 1, 3, 5, 7}

O/p: 1 3

Naive: O(mxn)

```
void Intersection(int a[], int b[], int m, int n)
{
 for(int i=0; i<m; i++)
 {
  if(i>0 && a[i]==a[i-1])
  {
   continue;
  }
  for(int j=0; j<n; j++)
  {
   if(a[i]==b[j])
   {
    cout<<a[i]<<" ";
    break;
   }
  }
 }
```

efficient: Using merge function

```
void Intersection(int a[], int b[], int m, int n)
{
 int i=0, j=0;
 while(i<m && j<n)
 {
  if(i>0 && a[i]==a[i-1])
  {
   i++;
   continue;
  }
  if(a[i]<b[j])
  {
   i++;
  }
  else if(a[i]>b[j])
  {
   j++;
  }
  else
  {
   cout<<a[i]<<" ";
   i++;
   j++;
  }
 }
```

Union of two Sorted Arrays:

I/p: a[] = {3, 5, 8}
b[] = {2, 8, 9, 10, 15}

O/p: 2 3 5 8 9 10 15

I/p: a[] = {2, 3, 3, 3, 4, 4}
b[] = {4, 4}

O/p: 2, 3 4

Naive: O((m+n) * log(m+n))

```
Void printUnion(int a[], int b[], int m, int n)
{ int c[m+n];
  for(int i=0; i<m; i++)
  {
    c[i]=a[i];
  }
  for(int i=0; i<n; i++)
  {
    c[m+i]=b[i];
  }
  sort(c, c+m+n);
  for(int i=0; i<m+n; i++)
  { if(i==0||c[i]!=c[i-1])
    {
      cout << c[i]<<" ";
    }
  }
```

Efficient solution: Using Merge function.

```
Void printunion(int a[], int b[], int m, int n)
{
  int i=0, j=0;
  while( i<m && j<n)
  { if (i>0 && a[i]==a[i-1])
    {
      i++;
      continue;
    }
    if( j>0 && b[j] ==>b[j-1])
    { j++;
      continue;
```

```
if(a[i]<b[j])
 {
  cout<<a[i]<<" ";
    i++;
  }
else if(a[i]>b[j])
 {
  cout<<b[j]<<" ";
   j++;
  }
 else
  {
   cout<<a[i]<<" ";
   i++;
    j++;
  }
while(i<m)
 {if(i>0&&a[i]!=a[i-1])
  {
   cout<<a[i]<<" ";
     i++;
  }
while(j<n)
  {if(j>0&&b[j]!=b[j-1])
   {
    cout<<b[j]<<" ";
      j++;
   }
  }
```

Count Inversions in an Array:

Ip: {2,4,1,3,5}
Op: 3 // (4,1)(4,3)(2,1)

* A pair (arr[i], arr[j]) is said to be inversion when i<j and arr[i] > arr[j].

Ip: {10,20,30,40}
Op: 0

* There are "0" inversions in a sorted Array.

Ip: {40,30,20,10}
Op: 6

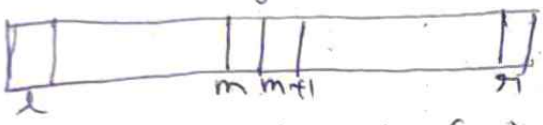
* There are max. no. of inversions in a sorted Array in decreasing order.

* for a sorted array in dec. order then first element has (n-1) & second element has (n-2) inversions and so on
(n-1)+(n-2)+---+1

Naive: $O(n^2)$

```
int countInversions (int arr[], int n)
{ int res=0;
for(int i=0; i<(n-1); i++)
{
  for(int j=i+1; j<n; j++)
  {
    if(arr[i]>arr[j])
    {
      res++;
    }
  }
}
return res;
}
```

Efficient: $O(n \log n)$ time



$m = \frac{l+r}{2}$

Every Inversion (x,y) where x>y has possibilities

a) Both x and y are in left half
b) " " " " " Right half
c) "x" is in left half and "y" is in Right half

```
int countAndMerge (int arr[], int l, int m, int r)
{
int n1= m-l+1, n2=r-m;
int left[n1], right[n2];
for(int i=0; i<n1; i++)
{ left[i]=arr[l+i];
}
for(int i=0; i<n2; i++)
{ right[i]=arr[m+1+i];
}
int res=0, i=0, j=0, k=l;
while (i<n1 && j<n2)
{ if(left[i]<=right[j])
  { arr[k]=left[i];
    i++;
```

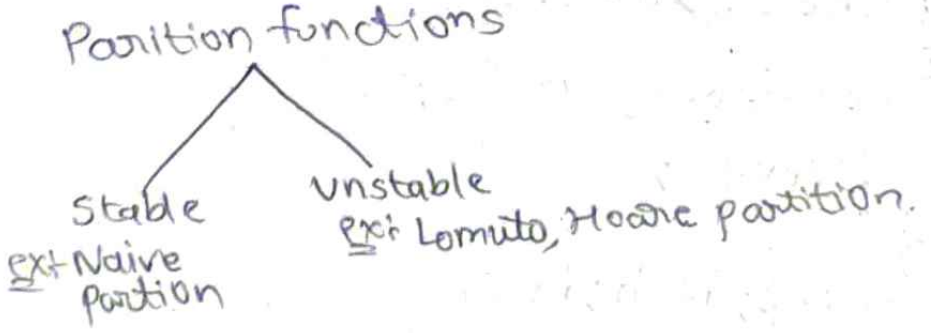
```
else if (arr[k] = right[j])
 {
   j++;
   k++;
   res = res + (n1 - i);
 }
while (i<n1)
 {
   arr[k] = left[i];
   i++;
   k++;
 }
while (j<n2)
 {
   arr[k] = right[j];
   j++;
   k++;
 }
return res;
}
```

Partition of a given Array:

I/p: arr[] = { 3, 8, 6, 12, 10, 7 }
P = 5 // index of last element

O/p: arr[] = { 3, 6, 7, 12, 10 }
(OR)
{ 6, 3, 7, 12, 10 }
Return value = 2 // index of element 7.

Stability: If two elements has the same value, then their original order should be maintained after sorting or any operation on it.



Naive partition:

<Pivot	Pivot	>Pivot

I/p: arr[] = {2, 7, 8, 3, 7}
p = 4

o/p: arr[] = {2, 3, 7, 7, 8}
Return value = 3

```
int partition(int arr[], int l, int h, int p)
{ int temp[h-l+1], index=0; ----> creating temp. Array
  for(int i=l; i<=h; i++)
  {
    if(arr[i]<arr[p])
    {
      temp[index]=arr[i];
      index++;
    }
  }
  for(int i=l; i<=h; i++)
  {
    if(arr[i]==arr[p])
    {
      temp[index]=arr[i];
      index++;
    }
  }
  int res=l+index-1;
  for(int i=l; i<=h; i++)
  {
    if(arr[i]>arr[p])
    {
      temp[index]=arr[i];
      index++;
    }
  }
  for(int i=l; i<=h; i++)
  {
    arr[i]=temp[i-l];
  }
  return res;
}
```

Lomuto partition:

```
int lPartition(int arr[], int l, int h)
{
  int pivot = arr[h]; //Assume pivot is always last element
  int i = l-1;
  for(int j=l; j<=h-1; j++)
  {
    if(arr[j]<pivot)
    {
      i++; //To increase size of smaller elements window
      swap(arr[i],arr[j]);
    }
  }
  swap(arr[i+1],arr[h]);
  return(i+1);
}
```

I/p: arr[] = {10, 80, 30, 90, 40, 50, 70} (50 ↓ pivot)

*How to handle the cases when pivot is not the last element?

I/p: arr[] = {50, 30, 20, 10, 5, 11} (20 ↑ p=2)

```
swap(arr[p], arr[h]); //add this line to move pivot value to last positn
```

O/p: {5, 30, 11, 10, 5, 20}

Hoare's partition:

```
int Partition(int arr[], int l, int h)
{
  int pivot = arr[l];
  int i = l-1; j = h+1;
  while(true)
  {
    do{
      i++;
    }while(arr[i]<pivot)
    do{
      j--;
    }while(arr[j]>pivot)
    if(i>=j)
    {
      return j;
    }
    swap(arr[i],arr[j]);
```

arr[] = {5, 3, 8, 4, 2, 7, 1, 10} (5 ↑ Pivot.)

Quick sort:

1) Divide and conquer algo.
2) worst case Time : $O(n^2)$
3) Despite $O(n^2)$ in worst case, it is considered as faster because of the following reasons
   1) In place
   2) cache friendly
   3) Average case is $O(n \log n)$
   4) Tail Recursive
4) Partition is key function (Naive, Lomuto, Hoare)

* If you don't want stability then Quick sort is more useful but if you want stability go for merge sort.

Quick sort using Lomuto Partition:

```
void qsort(int arr[], int l, int h)
{ if(l<h)
  {
    int p = partition(arr, l, h); // To put pivot element at it's correct position
    qsort(arr, l, p-1); // sort the elements before pivot element
    qsort(arr, p+1, h); // sort the elements after pivot element
  }
}
```

Dry run: {8, 4, 7, 9, 3, 10, 5} (0 1 2 3 4 5 6)

l=0, h=6

| 8 | 4 | 7 | 9 | 3 | 10 | 5 |

// Partition function executes

{4, 3, 5, 9, 8, 10, 7} // result of array after partition

qSort(0,1)

| 4 | 3 |

l=0 h=1

// last element considered as Pivot and put at correct position

{3, 4}

q.Sort(0, -1)

q.Sort (1,1)

| 9 | 8 | 10 | 7 |

l=3 h=6

// {7, 8, 10, 9} Take this sub Array

q.s(3,2)

q.s(4,6)

| 8 | 9 | 10 |

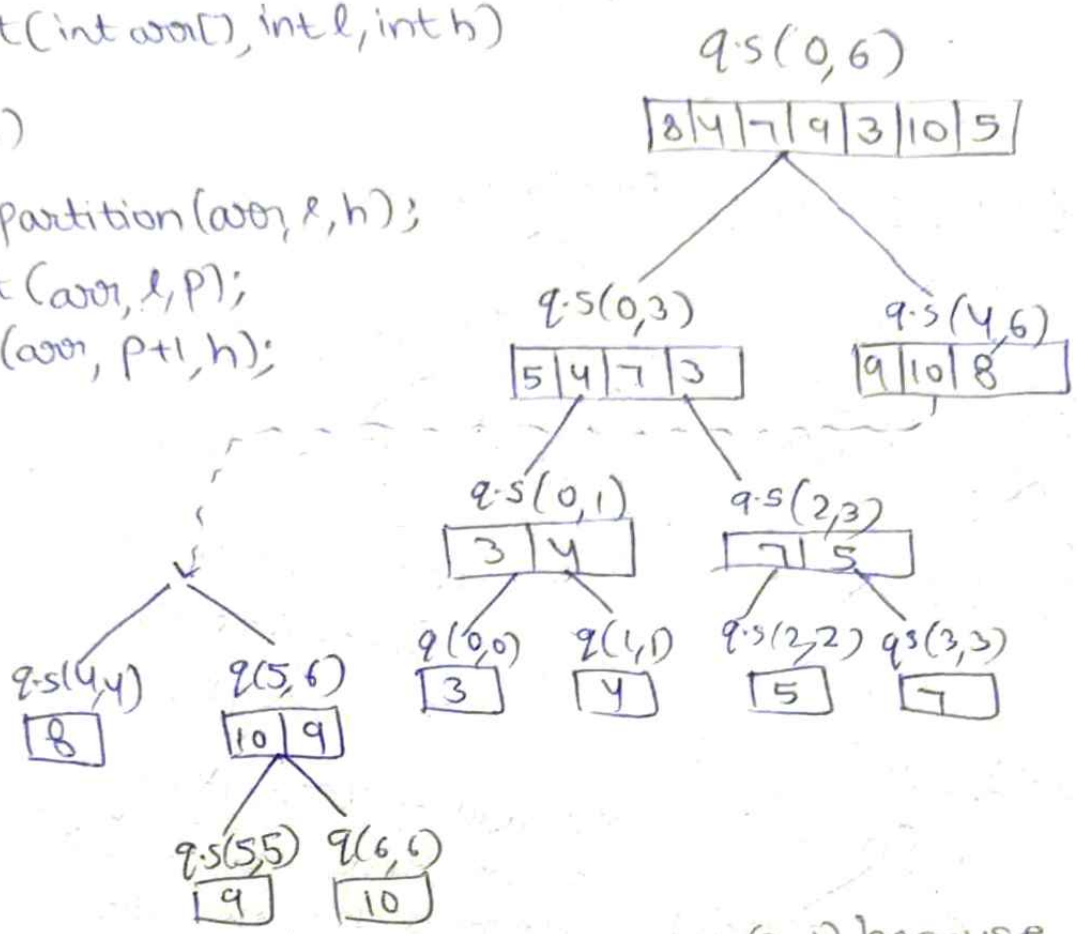
// after partition {8, 9, 10}

q.s(4,4)

q.s(6,6)

Quick sort using Hoare's Partition :

```
arr[] = {8,4,7,9,3,10,5}
void qsort(int arr[], int l, int h)
{
  if(l<h)
  {
   int p = partition(arr, l, h);
    qsort(arr, l, p);
    qsort(arr, p+1, h);
  }
}
```



*In Lomuto partition we do the quick sort upto (p-1) because pivot element is fixed at correct position and we do quick sort for (p+1) elements

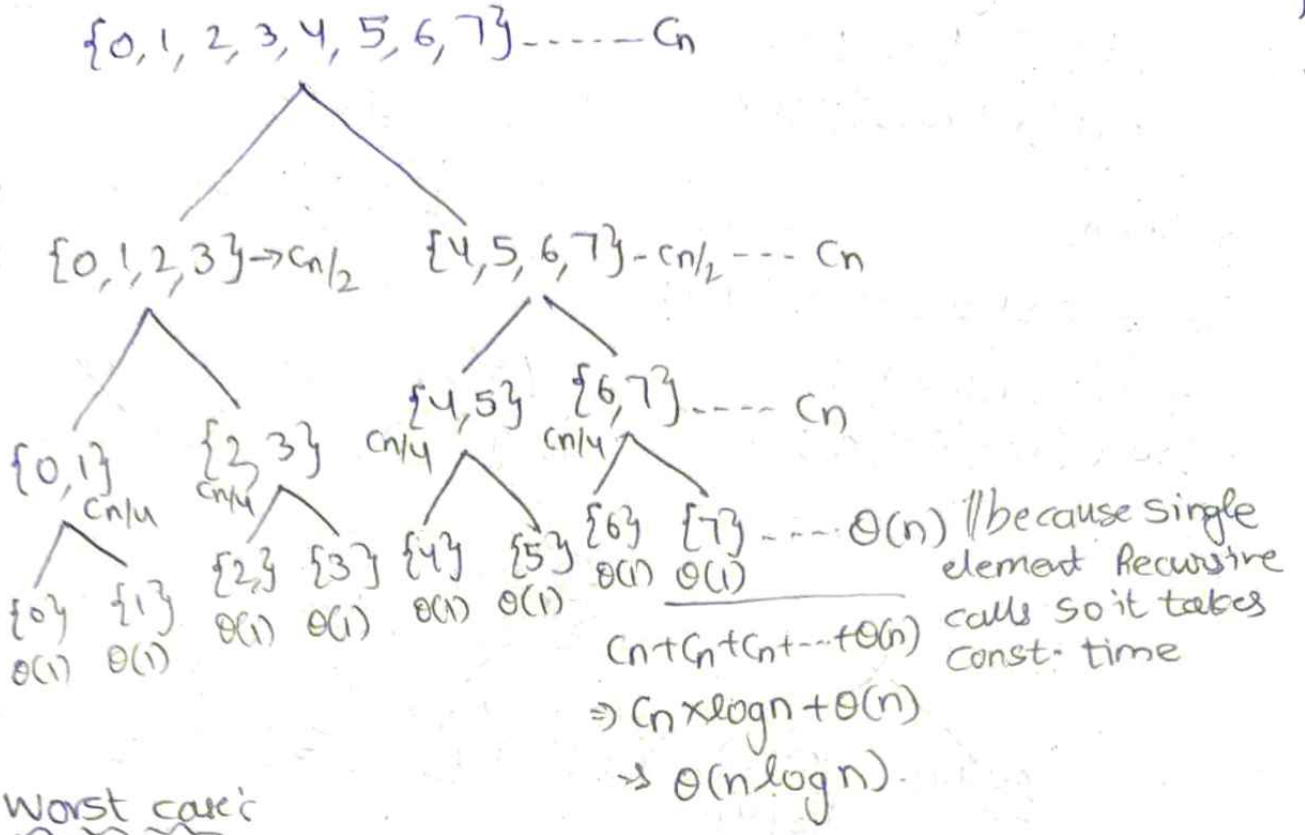
*Here in Hoare's partition no element is fixed we do quick sort upto "p" and and we do from "p+1" to h.

*on average Hoare's partition is "3" times faster than lomuto Partition

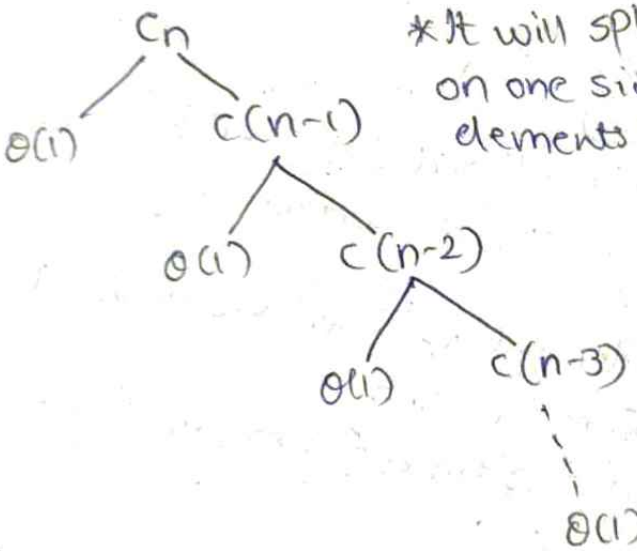
Analysis of quick sort :

```
ex: void qsort(int arr[], int l, int h)
    {
      if(l<h)
      {
       int p = partition(arr, l, h);
        qsort(arr, l, p);
        qsort(arr, p+1, h);
      }
    }
```

Best case:



Worst case:



* It will split the Array by one element on one side and all other (n-1) elements by one side.

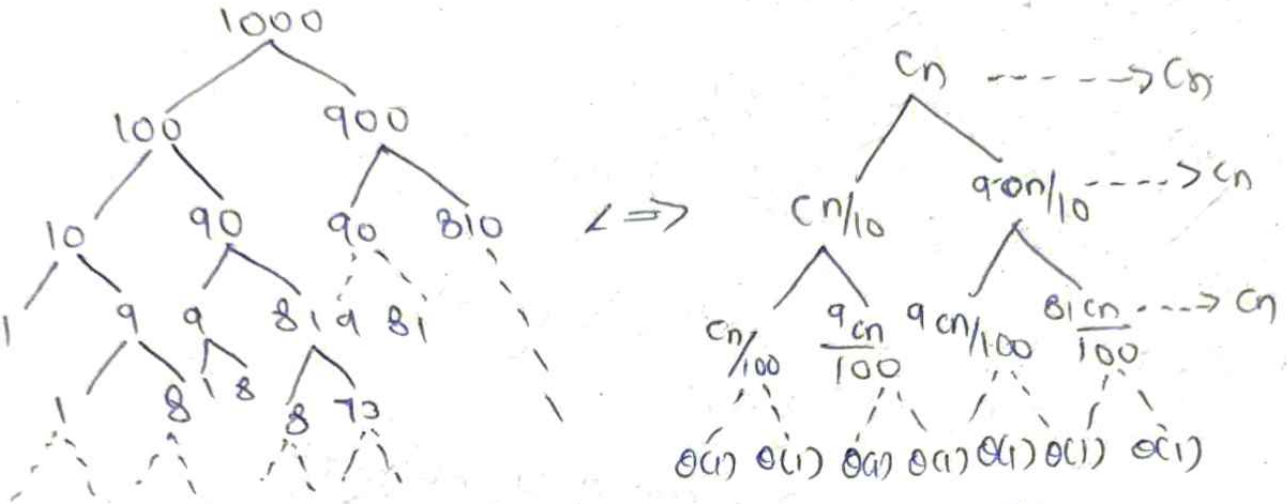
* for worst case, $T(n) = T(n-1) + O(n)$, because (n-1) elements on ~~one~~ side and "1" element on other side, here $O(n)$ is for partitioning the Array.

* For, Best case $T(n) = T(n/2) + T(n/2) + O(n)$

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

Idea for Average case:

* In this case we have $n/10$ elements on one side and remaining elements on another side.



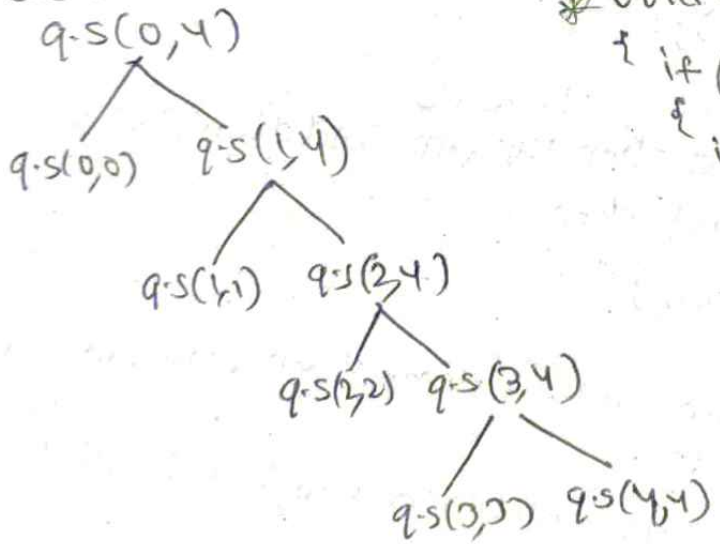
$\Rightarrow O(cn \times \log_{10/9} n)$

$\Rightarrow O(n \log n)$.

Space Analysis of Quick sort:

* In worst case the no. of recursion calls is equal to the no. of elements in the Array

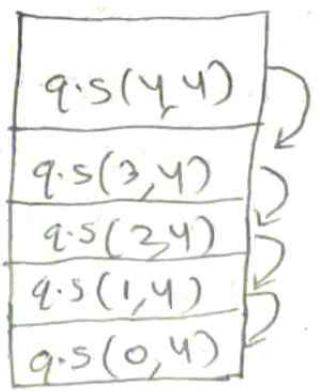
Worst case:



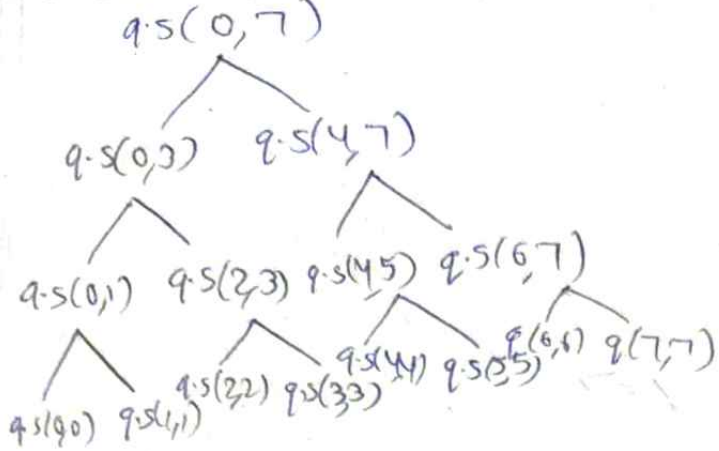
Auxiliary space required: $O(n)$

```
* void qsort(int arr[], int l, int r)
  { if (r > l)
    {
      int p = partition(arr, l, r);
      qsort(arr, l, p);
      qsort(arr, p+1, r);
    }
  }
```

function call stack:



Best case:



*Height of this Recursion tree is log(n).

*Auxilary Space : O(log n).

choice of Pivot and worst case of Quick sort:

```
*void qsort( int arr[], int l, int r)
  {
    if(l<r)
    {
      int p= partition (arr, l, r);
      qsort (arr, l, p);
      qsort (arr, p+1, r);
    }
  }
```

*when the Array is sorted then Hoare's partition picks the first element as a pivot then

{10, 50, 100, 200}

*n-1 elements on one side and "1" element on one side so it's leads to worst case

*This is same for lomuto partition also, but it picks pivot as a last element

*To avoid this issue we can do

```
int p= random(l,r); // to generate a pivot number
                       between "l" to "r");
```

then If you want to apply Hoare's partion then

```
swap(arr[i],arr[p]);
```

If you want to apply lomuto partition then

```
swap(arr[r],arr[p]);
```

Tail call elimination in Quick sort:

* If we donot do any after the Recursion such that functions are tail Recursive functions
* So, we can always optimize them using tail call elimination
* The main Idea of Tail call elemination is to avoid such cases like if the child recursive call retuns to last parent Recursive call then there is nothing to do by parent reursive call. It is called Tail recusive func. So to avoid this by Tail call elimination by giving some optimizations.

```
*void qsort(int arr[], int l, int r)
  {
    Begin: // optimization
     if(l<r)
     {
       int p= partition(arr, l, r);
       qsort(arr, l, p);
       l=p+1; // optimization
       goto begin //optimization
  } }
```

* Using this we can also optimize the req. space for Quick sort

Kth Smallest element:

I/p: arr[]={10,5,30,12}
K=2
O/p: 10 // Here the 2nd smallest number in Array is 10

I/p: arr[]={30,20,5,10,8}
K=4
O/p: 20 // Here the 4th smallest number is 20

Naive:

```
int kthsmallestelement(int arr[], int n, int K)
{
  sort(arr, arr+n);
  return arr[k-1];
}
```

T.C: O(n log n)

efficient: O(n)

```
int kthsmallest(int arr[], int n, int K)
{
  int l=0, r=n-1;
  while(l<=r)
  {
    int P= partition(arr, l, r); // Lomuto
    if(P==k-1)
    {
      return P;
    }
    else if(P>K-1)
    {
      r= P-1;
    }
    else
    {
      r=P+1;
    }
  }
  return -1;
}
```

arr[] = {10, 4, 5, 8, 11, 6, 26}, k=5 (indices 0 1 2 3 4 5 6)

l=0, r=6

⇒ {10, 4, 5, 8, 11, 6, 26}

P=6 ⇒ l=0, r=5

⇒ {4, 5, 6, 8, 11, 10, 26}

P=2, l=3, r=5

⇒ {4, 5, 6, 8, 10, 11, 26}

P=4 which is equal to "k-1".

*This an Quick select Algorithm

Minimum Difference in a Array:

I/p: arr[] = {1, 8, 12, 5, 18}

O/p: 3 // 8-5=3

I/p: arr[] = {8, -1, 0, 3}

O/p: 1 // 0-(-1)=1

I/p: arr[] = {10}

O/p: INF

Naive:

```
int getmindiff(int arr[], int n)
{
int res = INF;
for(int i=1; i<n; i++)
{
  for (int j=0; j<i; j++)
  {
    res = min (res, abs (arr[i]-arr[j]));
  }
}
return res;
}
```

ex: arr[] = {5, 3, 8}

res = INF;

i=1; j=0

res = min(INF, abs(3-5));

= 2

i=2; j=0

res = min(2, abs(8-5))

= 2

j=1

res = min (2, abs(8-3));

= 2

efficient:

arr[] = {10, 3, 20, 12}

1) Sort the Array

arr[] = {3, 10, 12, 20}

2) compute the differences blw adjacent elements

10-3 = 7

12-10 = 2

20-12 = 8

3) And return the minimum difference

```
* int getdiff(int arr[], int n)
{
  sort(arr, arr+n); // Array.sort(arr) in Java
  int res = INF; // INT_MAX in c++
  for(int i=1; i<n; i++)
  {
    res = min(res, arr[i]-arr[i-1]);
  }
  return res;
}
```

Working: arr[] = {10, 8, 1, 4}

After sorting

arr[] = {1, 4, 8, 10}

i=1 : min(res, 4-1);
= 3

i=2 : min(3, 8-4);
= 3

i=3 : min(3, 10-8);
= 2

Chocolate Distribution Problem:

I/p: arr[] = {7, 3, 2, 4, 9, 12, 56}
m=3

O/p: 2 // Difference blw 3, 2, 4, min=2, max=4 → 4-2=2

Rules:

* The each element in Array is the no. of chocolates in each packet
* every children get only one packet
* The difference of chocolates blw children should be minimized
* Here 'm' is the no. of children.

I/p: arr[] = {3, 4, 1, 9, 56, 7, 9, 12}
m=5

O/p: 6 // consider 3, 4, 7, 9, 9

Code:

```
int mindiff(int arr[], int n, int m)
{
  if(m>n)
   {return -1;
    }
  sort(arr, arr+n);
 int res = arr[m-1] - arr[0];
  for(int i=1; (i+m-1)<n; i++)
   {
     res = min(res, (arr[i+m-1]-arr[i]));
    }
  return res;
}
```

T.C: O(n log n).

Working:

res = arr[3-1] - arr[0]

= 2

Sort an Array with Two types:

1) Segrigate positive and negative

I/p: arr[] = {15, -3, -2, 18}

O/p: arr[] = {-3, -2, 15, 18}

2) Segrigate even and odd

arr[] = {15, 14, 13, 12}

O/p: {14, 12, 15, 13}

3) Sort a Binary Array.

I/p: arr[] = {0, 1, 1, 1, 0}

O/p: {0, 0, 1, 1, 1}.

Naive solution:

```
Void segrigate(int arr[], int n)
 {int temp[n], i=0;
  for(int j=0; j<n; j++) //coping negative elements from
    {  if(arr[j]<0)        original Array to temp. Array
       {
         temp[i]=arr[j]
          i++;
       }
    }
  for (int j=0;j<n;j++) //coping +ve elements from original Array to
   { if(arr[j]>=0)        temp Array
     {
       temp[i]= arr[j];
       i++;
   } }
  for(int j=0; j<n; j++) //coping temp. Array elements to
    {
      arr[j]= temp[j];   Original Array
    }
```

Efficient:

*Using Hoare's partition

```
void seggrigate (int arr[], int n)
{ int i=-1, j=n;
while (True)
 {
  do
  { i++;
  }while(arr[i]<0);
  do
  { j--;
  }while(arr[j]>=0);
  if(i>=j)
  {
   return;
  }
  swap(arr[i], arr[j]);
 }
}
```

arr[] = {-12, 18, -10, 15}

i = -1, j = 4

I st iteration : i = 1, j = 2

arr[] = {-12, -10, 18, 15}

II nd iteration : i = 2, j = 1

Time: O(n)

Aux. space : O(1)

*Single Traversal

Sort an Array with three types of elements:

1) Sort an Array's of 0's, 1's and 2's.

I/p: arr[] = {0, 1, 0, 2, 1, 2}

O/p: {0, 0, 1, 1, 2, 2}

2) Three way Partitioning

I/p: arr[] = {2, 1, 2, 20, 10, 20, 1}, pivot = 2

O/p: {1, 1, 2, 2, 20, 10, 20}

3) Partition Around a Range

I/p: {10, 5, 6, 3, 20, 9, 40}, range: [5, 10]

O/p: {3, 5, 6, 9, 10, 20, 40}

Naive:

```
void sort(int arr[], int n)
{ int temp[n], i=0;
for(int j=0; j<n; j++)
 { if(arr[j]==0)
   {
    temp[i]=arr[j];
     i++;
   }
}
for(int j=0; j<n; j++)
 { if(arr[j]==1)
   {
    temp[i]=arr[j];
   }}
 for(int j=0; j<n; j++)
  { if(arr[j]==2)
    {
     temp[i]=arr[j];
      i++;
    }}
  for(int j=0; j<n; j++)
   { arr[j]=temp[j];
   }
```

arr[] = {0, 1, 1, 2, 0, 1}

After 1st loop:

temp[] = {0, 0, -, -, -, -}

After 2nd loop:

temp[] = {0, 0, 1, 1, -}

After third loop:

temp[] = {0, 0, 1, 1, 1, 2}

After 4th loop:

arr[] = {0, 0, 1, 1, 2}.

Time: $\Theta(n)$

Aux Space: $\Theta(n)$

* four Traversal's is required

efficient:

* using Dutch National flag Algorithm, it is nothing but Modified Hoare's partition Algo.

```
void sort(int arr[], int n)
{ int low=0, hi=n-1, mid=0;
 while(mid<=hi)
 { if(arr[mid==0])
   { swap(arr[low], arr[mid]);
     low++;
     mid++;
   }
  else if(arr[mid]==1)
     { mid++;
     }
  else{
    swap(arr[mid], arr[hi]);
    hi--; } } }
```

Working:

⇒ {0, 1, 2, 1, 1, 2} ⇒ low=0, mid=0, hi=5
(low, mid; high)

⇒ {0, 1, 2, 1, 1, 2} ⇒ low=1, mid=1, hi=5
(l, m; h)

⇒ {0, 1, 2, 1, 1, 2} ⇒ low=1, mid=2, hi=5
(l; m; hi; high)

⇒ {0, 1, 2, 1, 1, 2} ⇒ low=1, mid=2, high=4
(l; m; h)

⇒ {0, 1, 1, 1, 2, 2} ⇒ low=1, mid=2, hi=3
(l; m; h)

⇒ {0, 1, 1, 1, 2, 2} ⇒ low=1, mid=3, high=3
(l; h; m)

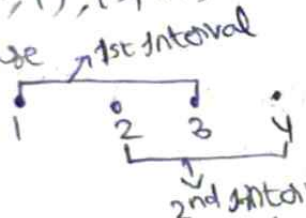
mid=4, hi=3 // While loop breaked

Merge overlaping Intervals:

I/p: arr[] = {{1,3}, {3,4}, {5,7}, {6,8}}

O/p: {(1,4), (5,8)}

* Because

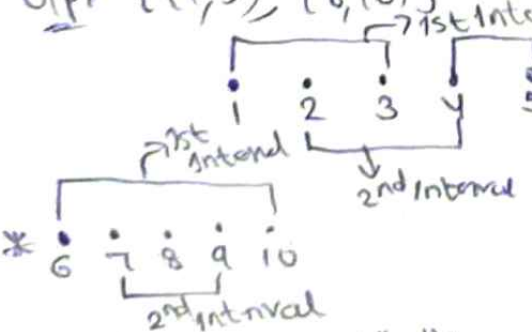


so, 2, 3 are common in two Intervals

so we can consider "1" as min. and "4" as max. elements here

I/p: arr[] = {(7,9), (6,10), (4,5), (1,3), (2,4)}

O/p: {(1,5), (6,10)}



→ So Here 2, 3, 4 are common in another Intervals we can consider "1" as min. element and "5" as max. element Here.

→ Here "6" is min. and "10" is max. element for this Interval

How to check if two Intervals overlap?

$i_1 = \{5, 10\}$

$i_2 = \{1, 7\}$

*first we have to check larger start value blw two Intervals here, it is 5. After we have to check this "5" is present in second Interval or not. If it present then two Intervals are overlap otherwise not.

second method:

$i_1 = \{5, 10\}$

$i_2 = \{1, 7\}$

*we have find smaller end value blw two Intervals. After check it if it is present in another Interval. If it present then they are overlapping Intervals otherwise not.

Efficient solution:

*The idea is to sort the Intervals by start time in increasing order or sort the Intervals by end time in the decreasing order

```
Void MergeIntervals(Interval arr[], int n)
  {
    sort(arr, arr+n, mycomp);
    int res=0;
    for(int i=1; i<n; i++)
    {
      if( arr[res].end >= arr[i].start)
        {
          arr[res].end = max(arr[res].end, arr[i].end);
          arr[res].start = min(arr[res].start, arr[i].start);
        }
        else
        {
          res++;
          arr[res] = arr[i];
        }
      for(int i=0; i<=res; i++)
        {
          cout << arr[i].start << " " << arr[i].end << endl;
        }
```

arr[] = {(9,14), (3,15), (5,10), (18,30)}

Meeting Maximum Guests:

Ip: arr[] = {900, 940} // Arrival time of guest
dpr[] = {1000, 1030} // Departure time of guest
Op: 2 // If we go b/w 9:40 to 10:00 we can meet 2 guests

Ip: arr[] = {800, 700, 600, 500}
dpr[] = {840, 820, 830, 530}
Op: 3 // If we go b/w 8:00 to 8:20 we can meet 3 guests

```
*int MaxGuest(int arr[], int dep[], int n)
 {
  Sort(arr, arr+n);
  Sort(dep, dep+n);
 int i=1; j=0, res=1; cuur=1;
 while (i<n && j<n)
 {
   if (arr[i]<=dep[j])
    {
      cuur++;
      i++;
    }
   else
    { cuur--; j++;
    }
    res = max(cuur, res);
    }
  }
 return res;
}
```

Timeline	A/D	No. of guests meet
6:00	A	1
7:00	A	2
7:30	D	1
8:00	D	0
9:00	A	1
10:00	D	0

Working:
arr[] = {900, 600, 700}
dep[] = {1000, 800, 730}
After sorting:
arr[] = {600, 700, 900}
dep[] = {730, 800, 1000}
Initially, i=1, j=0, cuur=1, res=1

cycle sort

* $O(n^2)$ in worst case
* Does minimum memory writes and can be useful for cases where memory write is costly.
* Inplace and Not-stable
* Useful to solve qn's like find min. no. of swap's req. to sort an Array.

```
* void cycleSort(int arr[], int n):
{
  for(int cs=0; cs<n-1; cs++)
  {
    int item=arr[cs];
    int pos=cs;
    for(int i=cs+1; i<n; i++)
    {
      if(arr[i]<item)
      {
        pos++;
      }
    }
    swap(item, Arr[pos]);
    while(pos!=cs)
    {
      pos=cs;
      for(int i=cs+1; i<n; i++)
      {
        if(arr[i]<item)
        {
          pos++;
        }
      }
      swap(item, arr[pos]);
    }
  }
}
```

arr[] = {10, 20, 50, 40, 30}

Heap sort:

* It is nothing but the modification of selection sort
* In selection sort we can find max. element by linear search
* Here, Instead of using linear search "max Heap" Data structure is used
* Building a "max Heap" or "min Heap" from an random Array is O(n) time
* If we want to sort an Array in Increasing order we use "max heap", In decreasing order we use "min Heap".
* O(n log n) in worst case.

Counting Sort:

I/p: arr[] = {1, 4, 4, 1, 0, 1}
K = 5
o/p: arr[] = {0, 1, 1, 1, 4, 4} // Numbers Ranging from 0 to 4 then output array must be in sorted order.

I/p: arr[] = {2, 1, 8, 9 4}
K = 10
o/p: arr[] = {1, 2, 4, 8, 9}.

Naive Implementation:

```
* void countsort (int arr[], int n, int K)
{ int count[K];
  for(int i=0; i<K; i++)
  {
    count[i]=0;
  }
  for(int i=0; i<n; i++)
  {
    count[arr[i]]++;
  }
  int index=0;
```

arr[] = {1, 4, 4, 1, 0, 1}
K = 5
⇒ count[] = {1, 3, 0, 0, 2}
0 1 2 3 4

```
for (int i=0; i<k; i++)
 {
   for (int j=0; j< count [i]; j++)
   {  arr[index] = i;
      index++;
   }
 }
```

i=0
arr[] = {0, -, -, -, -, -}
i=1
arr[] = {0, 1, 1, 1, -, -}
i=2
i=3
i=4
arr[] = {0, 1, 1, 1, 4, 4}

T.C : O(n+k).

efficient:

```
void count sort(arr, n, k)
 {
  int count [k];
 for(int i=0; i<k; i++)
  {
    count[i]=0; }
 for (int i=0; i<n; i++)
  {
    count [arr[i]]++;
  }
 for(int i=1; i<k; i++)
  {
    count[i] = count[i-1] + count[i];
  }
  int output [n];
  for (int i=n-1; i>=0; i--)
  {
    output [count[arr[i]]-1] = arr[i];
     count[arr[i]]--;
  }
  for(int i=0; i<n; i++)
   {
     arr [i] = output [i];
   }.
 }
```

Radix sort:

arr[] = {319, 212, 6, 8, 100, 50}

Re-writing numbers with leading zero's

319, 212, 006, 008, 100, 050

stable sort according to last digit

100, 050, 212, 006, 008, 319

stable sort according to middle digit

100, 006, 008, 212, 319, 050

stable sort according to the first digit

006, 008, 050, 100, 212, 319.

```
* void RadixSort(int arr[], int n)
  {
    int max = arr[0];
    for (int i=1; i<n; i++)
     {
       if(arr[i]>max)
        {
          max = arr[i];
        }
     }
   for (int exp=1; max/exp>0; exp= exp*10)
    {
      counting sort (arr, n, exp);
    }
   void countingSort (int arr[], int n, int exp)
   {
     Int count[10], output[n];
    for(int i=0; i<10; i++)
     {
       count[i]=0;
     }
    for(int i=0; i<n; i++)
    {
      count [(arr[i]/exp)%10]++;
    }
    for(int i=0; i<10; i++)
    {
      count[i] = count[i]+count[i-
```

```
for (int i=n-1; i>=0; i--)
{
  output [count [(arr[i]/exp)%10] - 1] = arr[i];
   count [(arr[i]/exp)%10]--;
}
for(int i=0; i<n; i++)
{
  arr[i] = output[i];
```

Working:

arr[] = {319, 212, 6, 8, 100, 50}

max = 319

counting sort (arr, 6, 1) ⇒ n=6, exp=1

counting sort (arr, 6, 10) ⇒ n=6, exp=10

countingsort (arr, 6, 100) ⇒ n=6, exp=100.

i) countingsort (arr, 6, 1) // Index = (arr[i]/1)%10

count[10] = {2, 0, 1, 0, 0, 0, 1, 0, 1, 1}
(indices 0 1 2 3 4 5 6 7 8 9)

count [10] = {2, 2, 3, 3, 3, 3, 4, 4, 5, 6}

arr[] = output[] = {100, 50, 212, 6, 8, 319}.

ii) counting sort(arr, 6, 10) // Index = (arr[i]/10)%10

count[10] = {3, 2, 0, 0, 0, 1, 0, 0, 0, 0}

count[0] = {3, 5, 5, 5, 5, 6, 6, 6, 6, 6}

arr[] = output[] = {100, 6, 8, 212, 319, 50}

iii) countingSort (arr, 6, 100) //Index = (arr[i]/100)%10

count[10] = {3, 1, 1, 1, 0, 0, 0, 0, 0, 0}

count[10] = {3, 4, 5, 6, 6, 6, 6, 6, 6, 6}

arr[] = output[] = {6, 8, 50, 100, 212, 319};

T.C = $\theta(d*(n+b))$

A.S = $\theta(n+b)$.

* It is a stable Algorithm