Bit Magic

Bitwise operators in C++ (Part 1)

AND & (When both are 1 then give 1) else 0.
OR |
XOR ^ (When different then 1 if same then 0

Bitwise AND

eg $x = 3$
$y = 6$

Binary Representation

x: 00...0011
y: 00...0110
(x&y): 00...0010

cout << (x&y) // 2

Bitwise OR

x: 3
y: 6

Binary Representation

x: 00...0011
y: 00...0110
(x|y): 00...0111

Page No.:
Date:

Bitwise XOR

x : 3
y : 6
Binary Representation
x : 00...0011
y : 00...0110
(x^y) : 00...0101

cout << (x^y) // 5

Bitwise operation in C++ (Part 2)

→ **Left Shift Operator**

x : 3

x : 000...011

(x<<1) : 000...110 ← Shift to left by 1 add zero at end

(x<<2) : 00...1100 ← Shifted to left by 2 added zeroes

(x<<y) : 00....110000

```
int x = 3
x<<1   // 6
x<<2   // 12
x<<4   // 48
```

* If we assume that the leading y bits are 0, then result of (x<<y) is equivalent to $x * 2^y$

OR,

Assuming that last y bits in the binary representation of x are ze

```
int x = 3
x<<1   // 6  → 3 X 2^1
x<<2   // 12 → 3 X 2^2
x<<4   // 48 → 3 X 2^4
```

→ Right Shift operator >>

x : 000 ... 0100001 ← last bit are ignored. Shift one right
(x>>1): 000 ... 0010000
(x>>2): 000 ... 0001000
(x>>y): 000 ... 0000010

int x = 33
x>>1 // 16
x>>2 // 8
x>>4 // 2

* x>>y is equivalent to floor of $\left\lfloor \frac{x}{2^y} \right\rfloor$

x = 33 CL16
x>>1 // 16 $\left[33/2^1\right] = 16$
x>>2 // 8 $\left[33/2^2\right] = 8$
x>>4 // 2 $\left[33/2^4\right] = 2$

Bitwise Not ~

$x = 1$

x : 00 01

$\sim x$: 11 10

$x = 5$

x : 00 0101

$\sim x$: 11 1010

eg:

```
unsigned int x = 1;
cout << (~x)
  // 4294967294
x = 5
cout << (~x)
  // 4294967290
```

unsigned

$2^{32} - 1$

4294967296

4294967295

Signed Input $(-2^{31}$ to $2^{31}-1)$

last bit → Number type

0 → positive

1 → negative

$x = 1$

x : 00 01

$\sim x$: 11 ..... 10 $(2^{32}-1-1)$

$2^{32}-2$

$x = 5$

x : 00 0101

$\sim x$: 11 ..... 1010 $(2^{32}-1-5)$

2^s complement of x in n bit representation

eg int x = 1

cout << (~x) // -2

x = 5

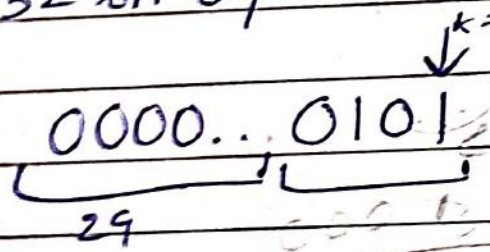
cout << (~x) // -6

Check if k-th bit is set?

1. I/P : n = 5, K = 1

Sol

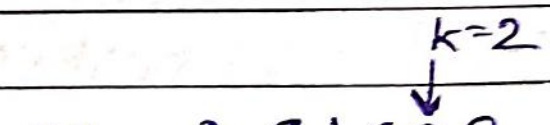
32 bit representation of 5



O/P = Yes

2. I/P : n = 8, K = 2

Sol O/P : NO



3. I/P : n = 0, K = 3

O/P : NO

K ≤ No. of bits in binary representation

Method 1 (left shift)

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

eg. I/P : n = 5, K = 3
O/P = Yes

AND operation

n = 000....0101
1<<(K-1) = 000....0100
000....0100

Method 2 (Righ shift)

```
Void Kth Bit (int n; int K) {
    if ((n >> (K-1)) & 1 == 1)
        print ("Yes");
    else
        print ("No");
}
```

I/P : n = 13, K = 3
O/P : Yes

AND

n = 1101
(n >> (K-1)) = 0011
1 = 0001
0001

Count Set Bits

1. n = 5

Binary representation : 101

o/p = 2

2. I/p = n = 7

Binary representation : 0111

o/p = 3

3 I/p : n = 13

Binary representation : 1101

o/p = 3

Method 1

```
int countSet (int n)
{int cs = 0;
 while (n>0)
 {
    if (n%2 != 0)
       cs++;
    n = n/2;
 }
}
```

OR

```
if((n&1) == 1)
   cs++;
n = n >> 1;
```

OR

```
cs = cs + (n&1);
n = n>>1
```

Method 2

Brian Kernigam's algorithm (Set Bit Count)

$n = 40$

Binary Representation:

initial : 00....0101000

After Ist Iteration: 00...0100000

After IInd Iteration: 00...0000000

```
int CountBits(int n)
{
    int ans = 0;
    While (n > 0)
    {
        n = (n & (n-1));
        ans++;
    }
    return ans;
}
```

101000	n = 40
100111	n-1 = 39
100000	n = 32

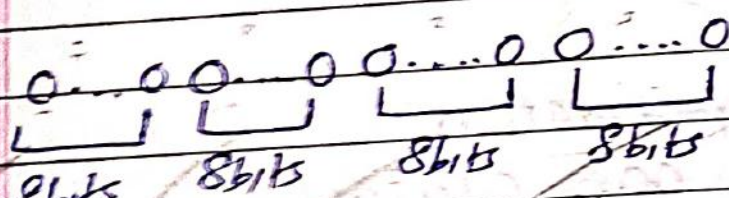
100000	n = 32
011111	n-1 = 31
000000	

Method 3

Lookup Table Method for 32 bits No.

No. represent in 8 bit = 0 .... 2^8-1

n = 13



i = count of set bit

```
int table[256];

void initialize()
{
    table[0] = 0;
    for (int i=1; i<256; i++)
        table[i] = (i&1) + table[i/2];
}
int count(int n)
{
    int res = table[n & 0x ff];
    n = n >> 8;
    res = res + table[n & 0xff];
    n = n >> 8
    res = res + table[n & 0xff];
    n = n >> 8
    res = res + table[n & 0xff];
    return res;
}
```

Complexity

→ O(1).

Power of two

1. I/p : n = 4
   o/p : Yes

2. I/p : n = 6
   o/p : No

Method 1 (Naive)

```
bool isPow2(int n)
{ if (n==0)
      return false
  while (n!=1)
    { if (n%2!=0)
        return false
      n = n/2;
    }
  return true;
}
```

n = 32 bit number

0Xff = 00...0 11111111 (24 bit | 8 bit)

n & 0Xff

↳ last 8 bit (getting right most part)

Method 2

Brian Kernigam's algorithm

the power of 2 element have set count alway equal to 1.

eg: 0000...0100 → 4
0000...1000 → 8

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

if (countBit(n) == 1)
    Print true;
else
    Print false;
```

Method 3

```
bool IsPow2(int n)
{
    if (n==0)
        return false;
    return ((n & (n-1)) == 0);
}
```

example 1

$n = 4$: 00...100

$(n-1) = 3$: 00...011

00...000

example 2

$n = 6$: 00...110

$(n-1) = 5$: 00...101

00...100

Find the only odd occurring number

1. 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

Navive Solution

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

    }
```

number

Method 2

Bitwise XOR ^

$x \wedge 0 = x$

$x \wedge y = y \wedge x$

$x \wedge (y \wedge z) = (x \wedge y) \wedge z$

$x \wedge x = 0$

```
int findOdd (int arr[], int n)
{
    int xrs = 0;
    for (int i=0; i<n; i++)
        xrs = xrs ^ arr[i];
    return xrs;
}
```

Q Variation Question : given an array of n number that has value in range [1.... n+1]. Every no. appear exactly once. Hence one no. is missing. find the missing no.

I/P : arr[] = {1, 4, 3}

O/P : 2

(1 ^ 2 ^ 3 ... ^ (n+1)) ^ arr[0] ^ arr[1] ... arr[n-1]

I/P : arr[] = [1, 5, 3, 2]

O/P : 4.

Sol: Bitwise OR of given no [array] & bitwise of range created.

Find two odd appearing number.

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

Method 1 Naive solution O(n²)

```
void oddAppearing(int arr[], int n)
{
    for(int i=0; i<n; i++)
    {
        int cnt=0;
        for(int j=0; j<n; j++)
            if(arr[i] == arr[j])
                cnt++;
        if(cnt%2 != 0)
            print(arr[i]);
    }
```

Method 2 (O(n))

Discussion: XOR of all no.

XOR = 3 ^ 4 ^ 3 ..... ^ 7

= 5 ^ 6

= 3

5 : 1 0 1

6 : 1 1 0

0 1 1

← Take this set bit and divide the array into two group such that group1 have ~~set bit and group2~~ didn't have.

group 1 = [3, 3, 5, 7, 7] = 5

group 2 = [4, 4, 4, 4, 6] = 6

Code ON NEXT PAGE

```
void addAppearing (int arr[], int n) {
    int XOR = 0, res1 = 0, res2 = 0;
    for (int i = 0; i < n; i++)
    {
        XOR = XOR ^ arr[i];
    }

    int Sn = XOR & ~(XOR - 1);   // Right most

    for (int i = 0; i < n; i++)
    {
        if ((arr[i] & Sn) != 0)
            res1 = res1 ^ arr[i];
        else
            res2 = res2 ^ arr[i];
    }
    printf (res1, res2);
}
```

Explanation

I/P: arr[] = [3,4,3,4,5,4,4,6,7,7]

O/P: 5 6

XOR = 3 ^ 4 ^ ..... ^ 7
XOR = 5 ^ 6
= 3

XOR = 3 : 0 0 ... 0 1 1
(XOR-1) (3-1) : 0 0 ... 0 1 0
~(XOR-1) : 1 1 ... 1 0 1
3 & ~(3-1) = (00...011) & (11...101)
= (00...001)

xcs1

group 1: [3, 3, (5), 7, 7]
group 2: [4, 4, 4, 4, (6)]

xcs2

I/P: arr[] = [3,4,3,4,8,4,4,32,7,7]

O/P: 8, 32

XOR = 3 ^ 4 ^ 3 ..... ^ 7
XOR = 8 ^ 32
= 40

~~XOR~~ ~~8~~ : 0 ... 0001000
~~XOR~~ 32 : 0 ... 0100000
XOR : 40 0 ... 0101000
(XOR-1) 0 ... 0100111
~(XOR-1) 1 ... 1011000
XOR & ~(XOR-1),
00 ... 01000

Power Set using Bitwise Operator

I/P : s = "abc"
O/P : "", "a", "b", "c", "ab", "ac", "bc", "abc"

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"
4	100	"c"
5	101	"ac"
6	110	"bc"
7	111	"abc"

value varies from 0 to $2^n - 1$

I/P : s = "ab"
O/P : "", "a", "b", "ab"

Counter (Decimal)	Counter (Binary)	Subset
0	00	""
1	01	"a"
2	10	"b"
3	11	"ab"

value varies from 0 to $2^n - 1$

```
void printPowerSet(string str)
{
    int n = str.length();
    int powsize = pow(2,n);

for (int counter =0; counter<powsize;
                        counter++)
{   for (int j=0; j<n; j++)
    {   if (counter & (1<<j)!=0)
            print(str[j]);
    }
    print("\n");
}
}
```

counter = 3

j = 0 to j = 2

j = 0

011 & (001<<0)

011 & 001

001

print → a

j = 1 011 & (010)

010

Print - b

j = 2

011 & (100)

000

ch: 4 Recursion

Write a recursive function to check if a string palindrome.

I/P: str = "aabaa"
O/P: Yes
I/P: str = "geeks"
O/P: No

// Initially s=0, e=n-1

```
bool isPal(String str, int s, int e)
{
    if (s==e) return true;
    if (s>e) return true;
    if (str[s] != str[e])
        return false;
    return isPal(str, s+1, e-1);
}
```

Writing Base Cases in Recursion.

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

nth Fibonacci Number where n >= 0

0, 1, 1, 2, 3, 5, 8, 13.....

I/P : n = 3
O/P : 2

```
int fib(int n) {
    if (n==0) return 0
    if (n==1) return 1;
  return fib(n-1) + fib(n-2);
}
```

Write a recursive function to find sum of digits in a number.

1. I/P : n = 253
   O/P : 10
   2+5+3

2. I/P : n = 9987
   O/P : 33
   9+9+8+7

```
int fun (int n)
    if (n<10)
        return n;
    return fun (n/10) + n%10;
```

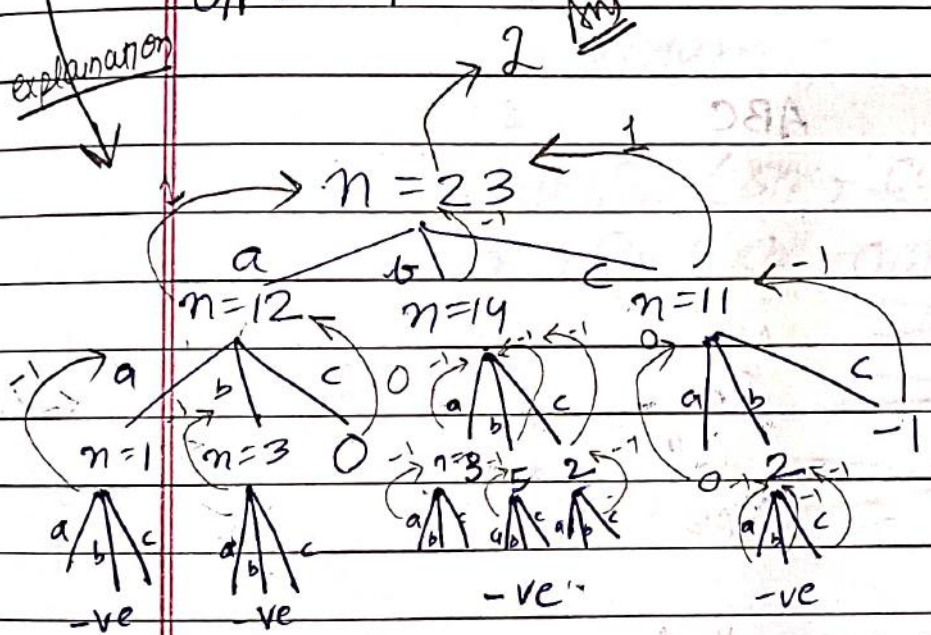
Given a rope of length n, you need to find maximum number of pieces you can make such that length of every piece is in set {a, b, c} for given three value a, b & c

1. I/P : n = 5
   a = 2 , b = 5 , c = 1
   O/P : 5

2. I/P : n = 23
a = 12, b = 9, c = 11
O/P : 2

3. I/P : n = 5
a = 4, b = 2, c = 6
O/P : -1

explanation




```
int maxCuts (int n, int a, int b, int c)
{
    if (n == 0) return 0;
    if (n < 0) return -1;

    int res = max ( maxCuts(n-a, a, b, c),
                    maxCuts(n-b, a, b, c),
                    maxCuts(n-c, a, b, c) );

    if (res == -1) return -1;
    return (res + 1);
}
```

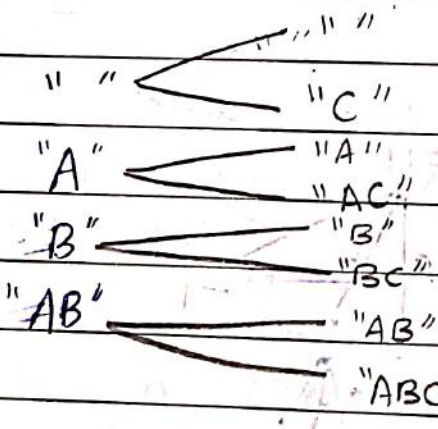
Complexity. → $O(3^n)$

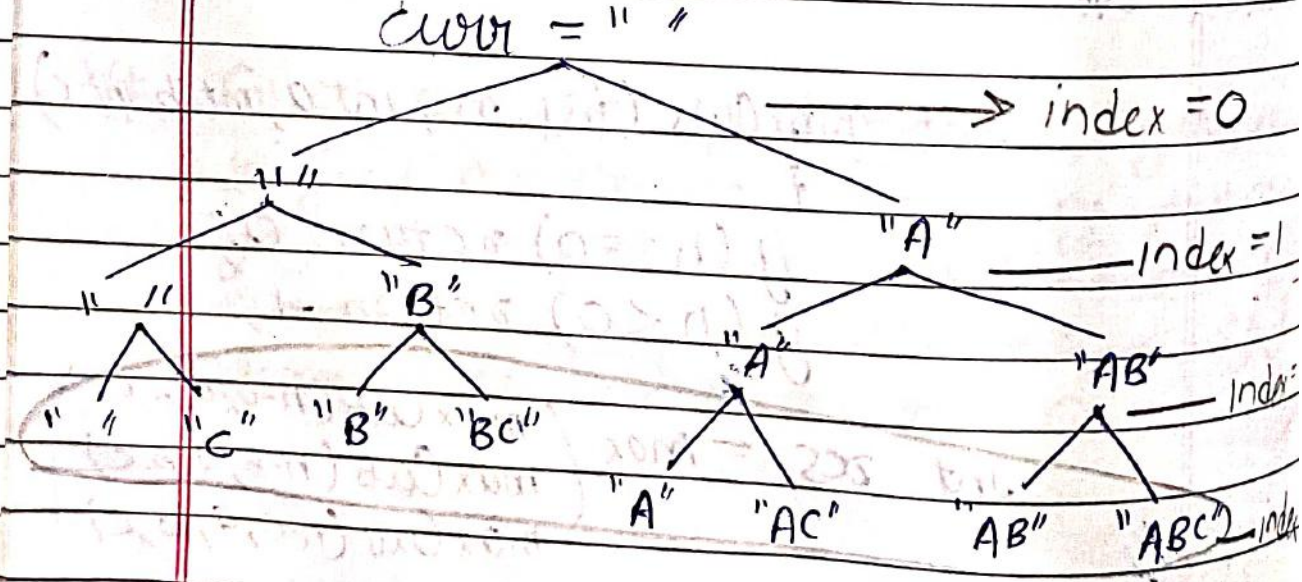
Given a string print all subsets of it (in any order)

I/P : str = "ABC"

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

If I have solution for AB



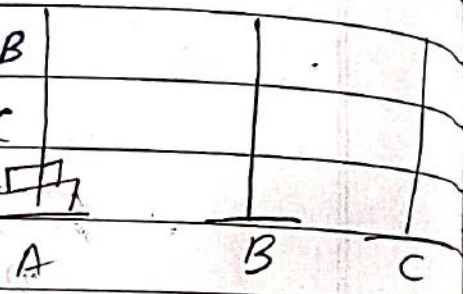


```
void printSub (String str, String curr="",
                int index=0)
{
    if (index == str.length())
    {
        count << curr << " "
        return;
    }
    printSub (str, curr, index+1)
    printSub (str, curr+str[index], index+1);
}
```

Tower of Hanoi

1. I/p : n = 1
   o/p : Move Disc 1 from A to C

2. I/p : n = 2
   o/p : move Disc 1 from A to B
   move Disc 2 from A to C
   move Disc 1 from B to C



Rules:

1. Only one Disc moves at a time
2. No larger Disc above smaller
3. Only the top Disc of a tower can be moved.

Problem 1

I/p n = 1

o/p move Disc from A to C

Problem 2

I/p : n = 2

o/p : move Disc 1 from A to B
move Disc 2 from A to C
move Disc 1 from B to C

Problem 3. I/P : n = 3

move disc 1 from A to C
move disc 2 from A to B
move disc 1 from C to B
move disc 3 from A to C
move disc 1 from B to A
move disc 2 from B to C
move disc 1 from A to C

source Auxillary destination

```
TOH (n, A, B, C)
{
    TOH((n-1), A, C, B)
    move Disc n from A to C
    TOH((n-1), B, A, C)
}
```

** No. of movement for given n

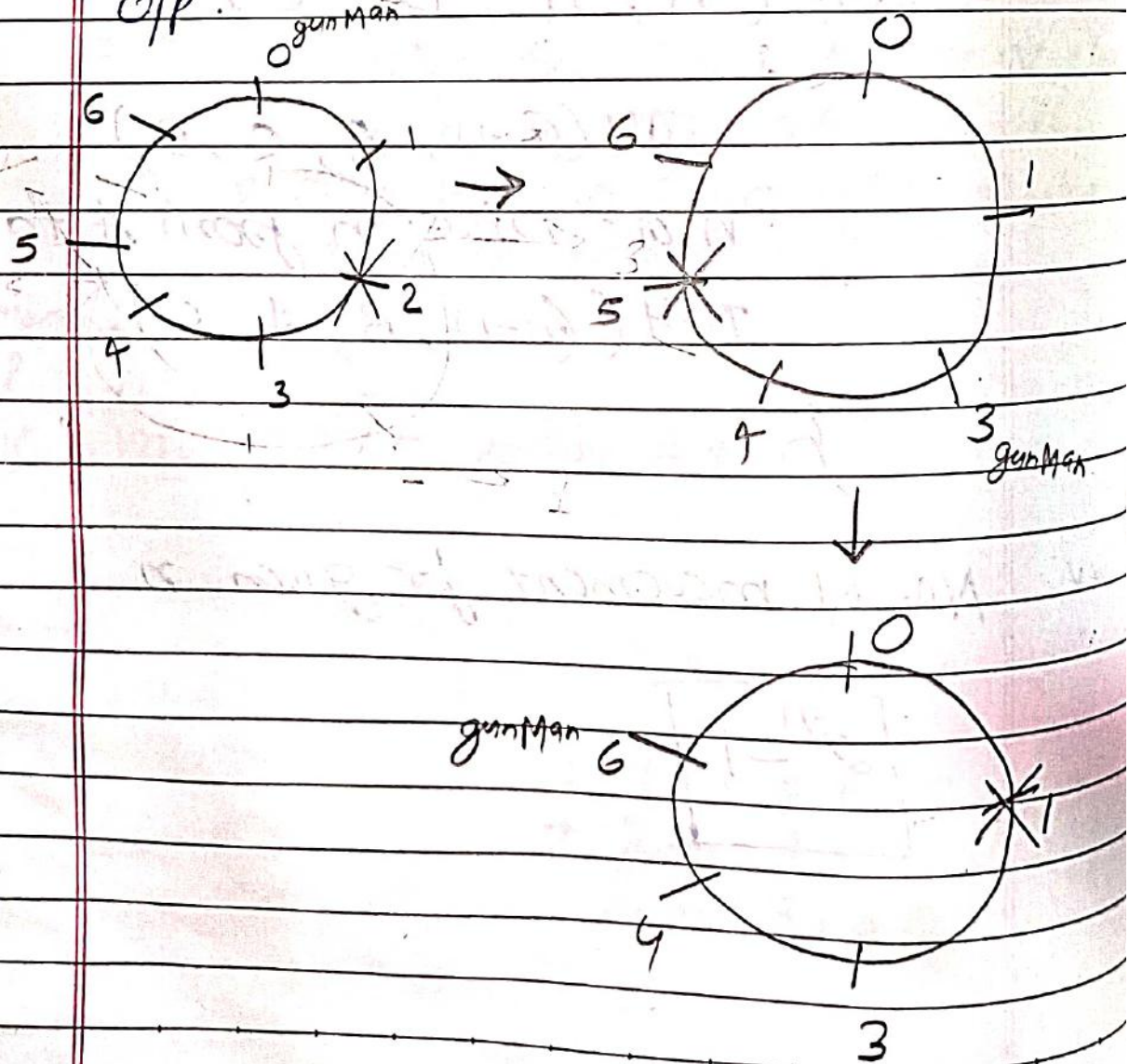
$$2^n - 1$$

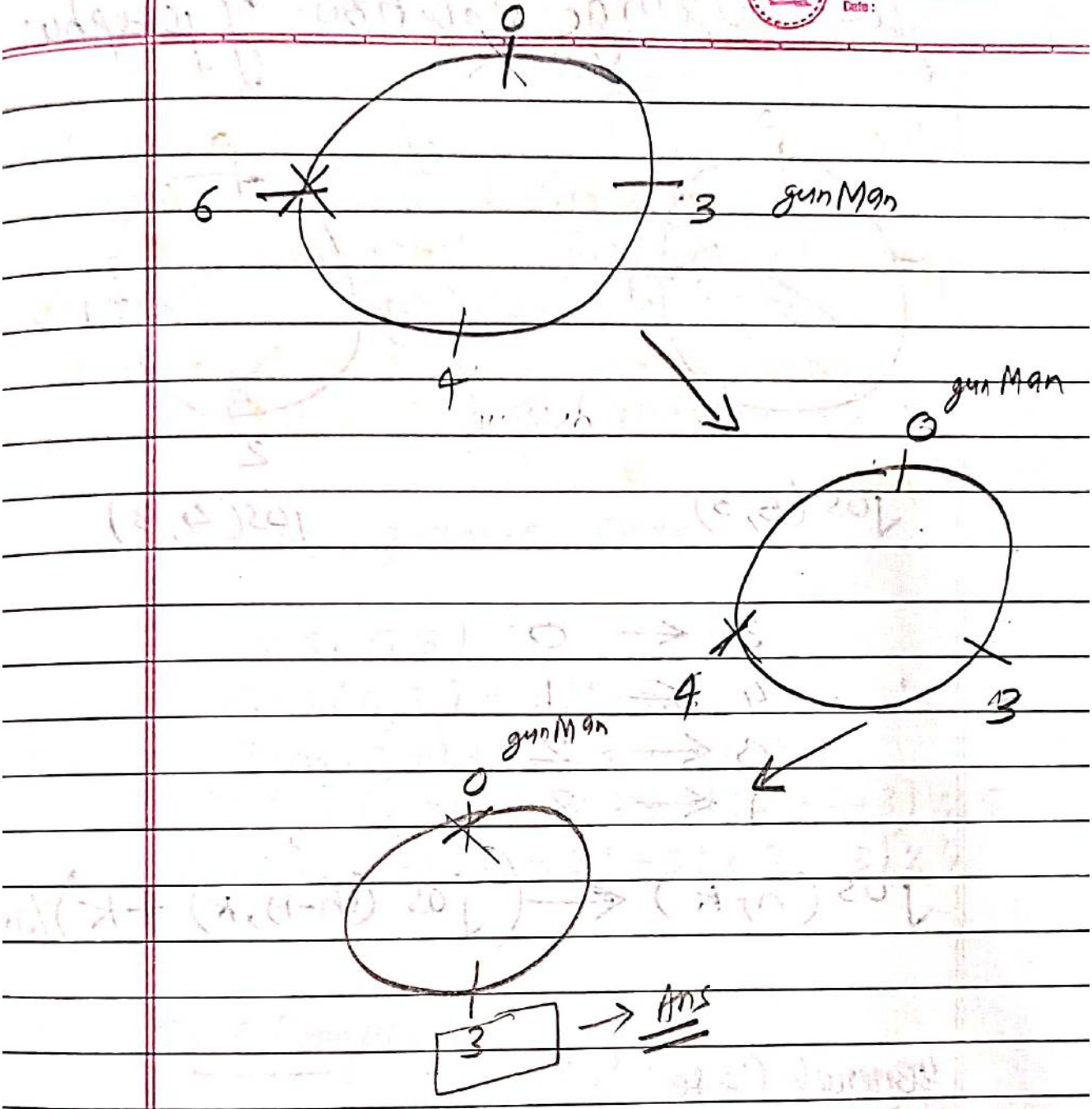
Josephus Problem

There are n people standing in a circle. we need to kill kth person in every iteration and this has to be done in circular manner. After repetately doing this we have to find the position of Survival.

1. I/P : n = 7 // 7 people in circle
   k = 3 // kill the 3 person.

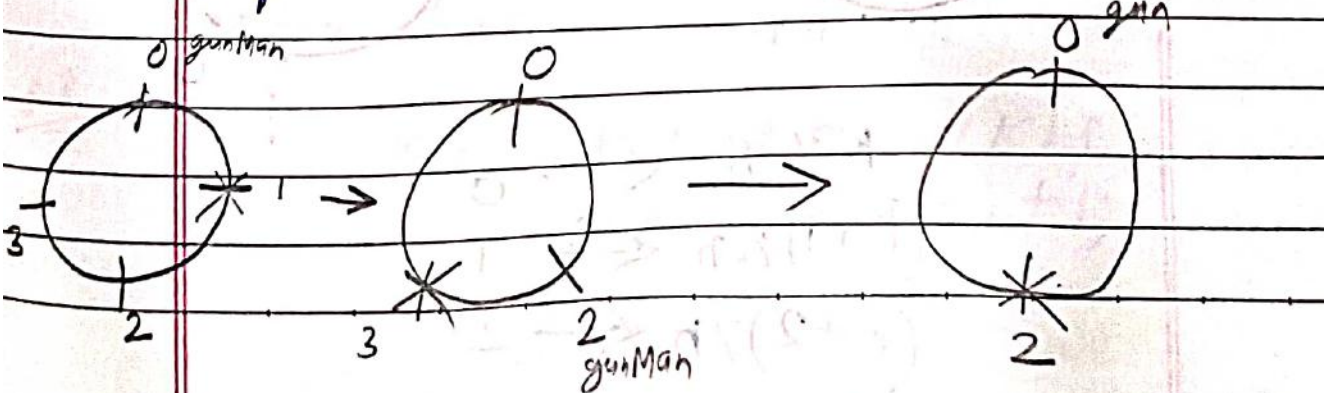
O/P :




0
6
3
gunMan
4
gunMan
0
4
3
gunMan
0
3
Ans

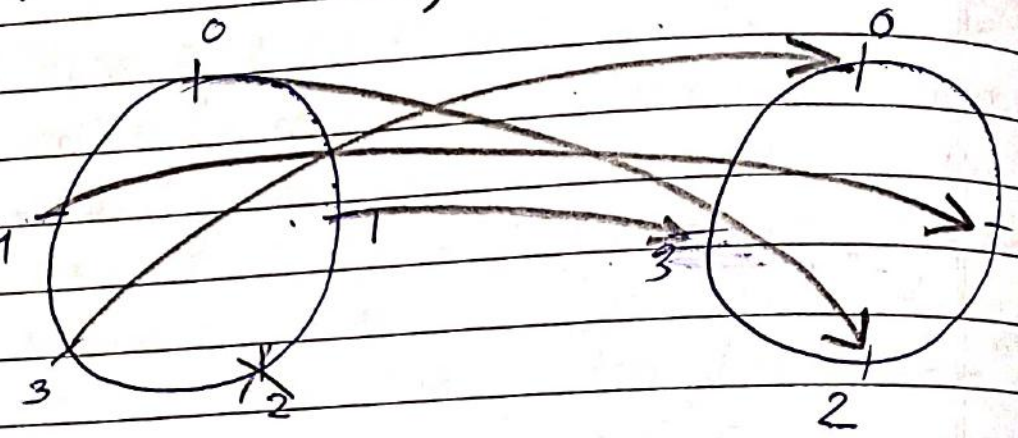
2. I/P : n=4, k=2

o/p : 0


0
gunMan
3
2
0
3
2
gunMan
0
gun
2

Generalizing Solution of Josephus

$n = 5, K = 3$



jos(5,3) jos(4,3)

$3 \leftarrow 0$

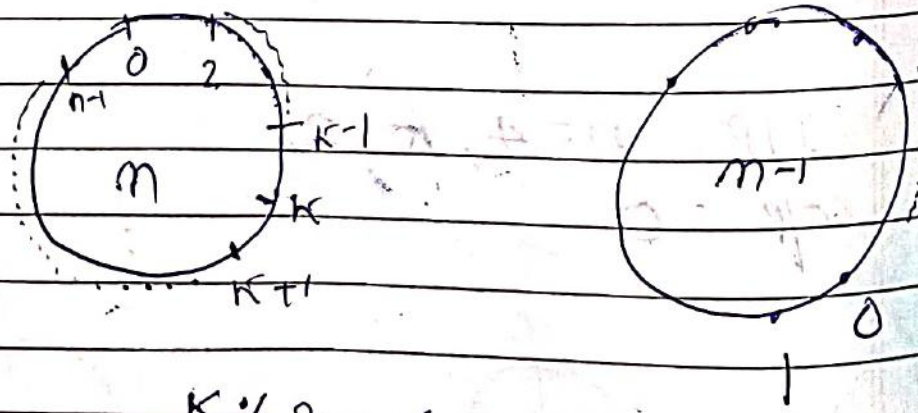
$4 \leftarrow 1$

$0 \leftarrow 2$

$1 \leftarrow 3$

$jos(n,K) \leftarrow (jos((n-1),K) + K)\%n$

general Case



$K\%n \leftarrow 0$

$(K+1)\%n \leftarrow 1$

$(K+2)\%n \leftarrow 2$

$(K+i)\%n \leftarrow i$

```
int jos (int n, int K)
{
    if (n==1)
        return 0
    else
        return (jos(n-1, K)+K) %n
```

For run of above code

jos(5,3)
(jos(4,3)+3)%5
((jos(3,3)+3)%4+3)%5
(((jos(2,3)+3)%3+3)%4+3)%5
((((jos(1,3)+3)%2)+3)%3+3)%4

Time Complexity $T(n) = T(n-1) + C$
$= O(n)$

* Extension to problem: If it is not start with index = 0. It will start with index 1.

Sol: Code

```
int myJobs (int n, int K){
    return jos(n,K)+1;
}
```

Subset sum problem

→ Subsets of [1, 2, 3] are
[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3].

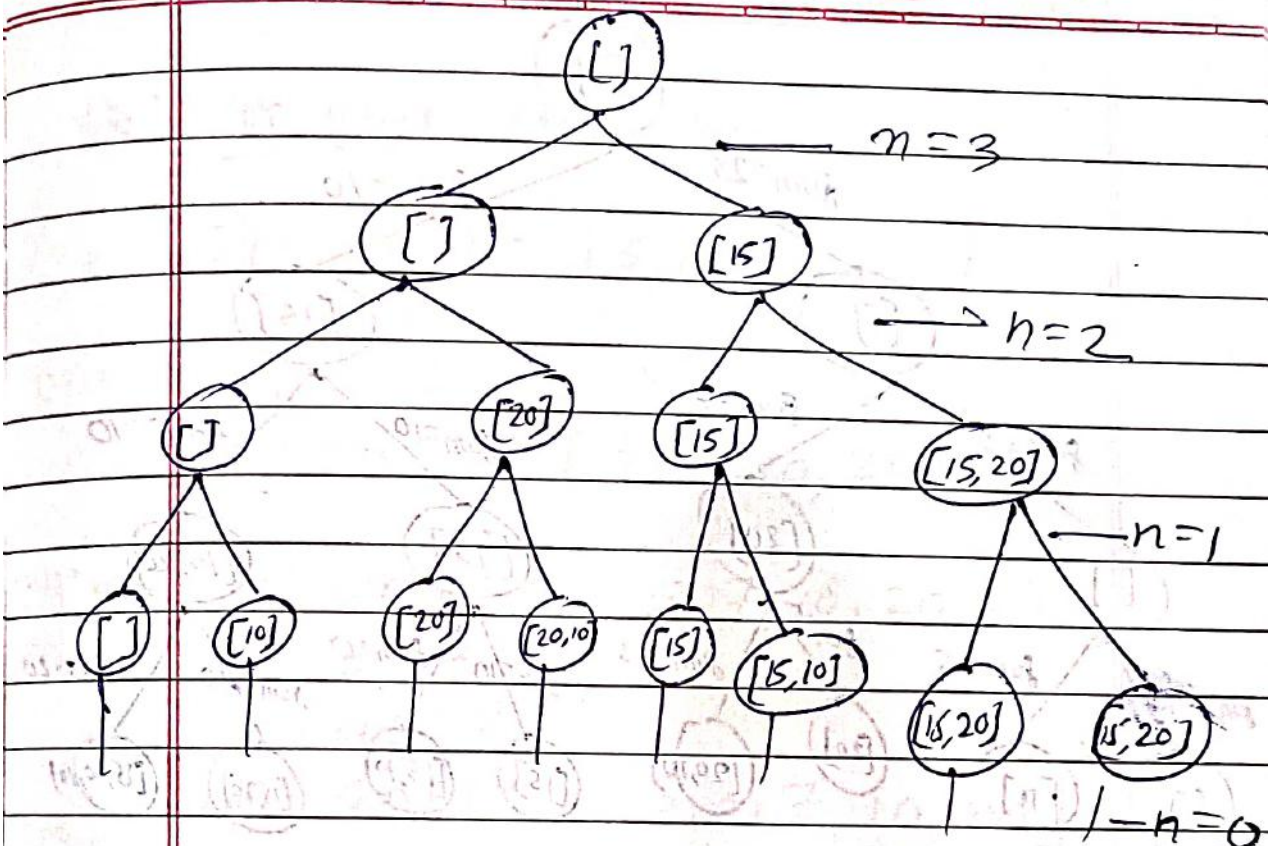
1. I/P: [10, 5, 2, 3, 6]
sum = 8
O/P: 2

2. I/P: [1, 2, 3]
sum = 4
O/P: 1

3. I/P: [10, 20, 15]
sum = 37
O/P: 0

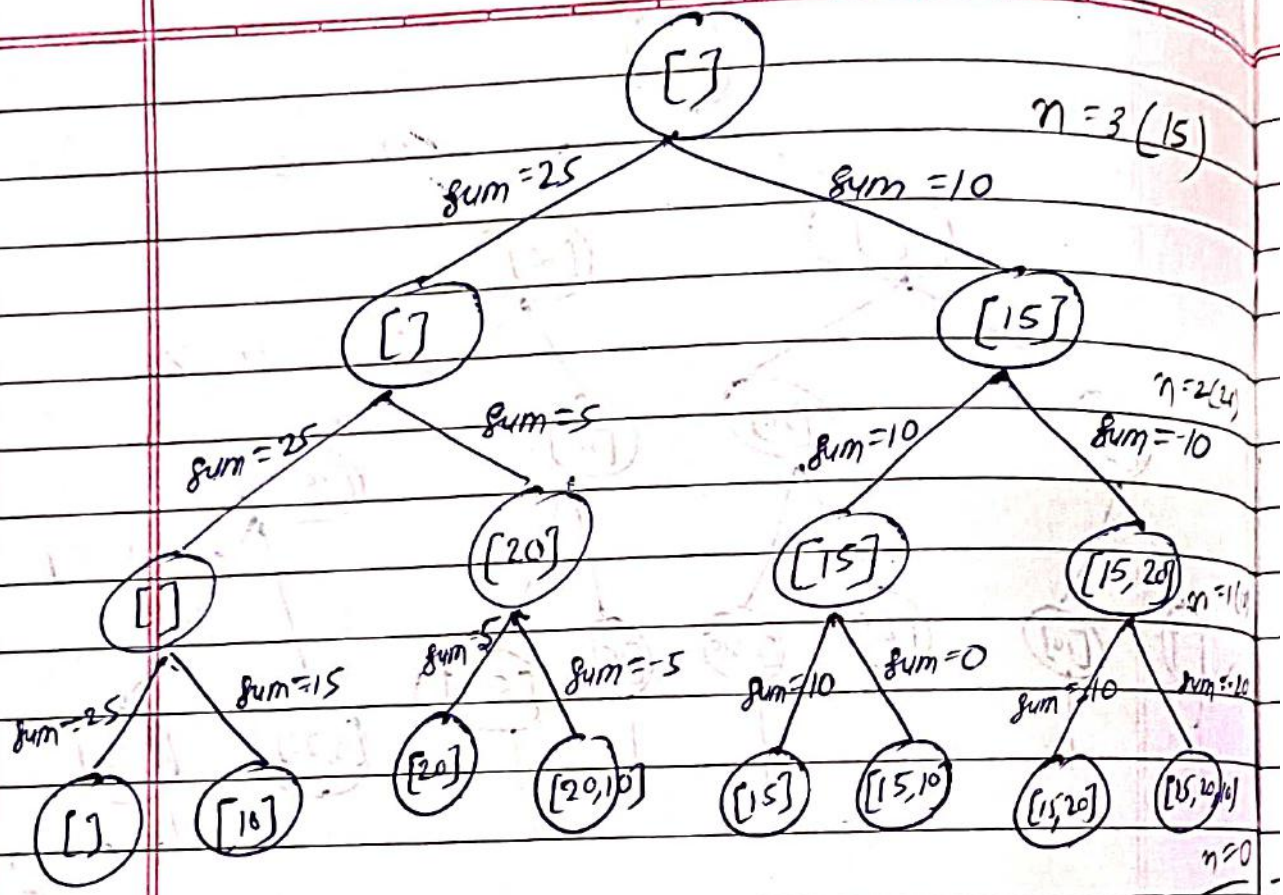
4. I/P: [10, 20, 15]
sum = 0
O/P = 1

Idea for recursive solution for set [10, 20, 15] & sum = 25.



Code for the above discuss algorithm but in tricky way:-

```
int countSubsets(int arr[], int n, int sum)
{
    if (n == 0)
        return (sum == 0) ? 1 : 0;

    return countSubsets(arr, n-1, sum) +
           countSubsets(arr, n-1, sum - arr[n-1]);
}
```



Complexity $\rightarrow 2^n + (2^n - 1) = O(2^n)$

Ch: 5 Arrays

Operation on Arrays

1. I/P: arr[] = [5, 10, 20, -, -]
   x = 7
   pos = 2
   O/P: arr[] = [5, 7, 10, 20, -]

2. I/P: arr[] = [5, 7, 10, 20, -]
   x = 3
   pos = 2
   O/P: arr[] = [5, 3, 7, 10, 20]

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

n → size of array
cap → array maximum capacity
pos → position where we want to insert
x → the element

Time Complexity ⇒ O(n)

Operation on Arrays (Part 2)

Deletion

If: arr[] = [3, 8, 12, 5, 6]

x = 12

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

```
int deleteElement ( 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 ; j++)
                  arr[j] = arr [j+1];
              return (n-1)
            }
```

Complexity → O(n)

Revere Array

I/P: arr[] = [30, 7, 6, 5, 10]
o/p: arr[] = [10, 5, 6, 7, 30]

```
void xverse (int arr[], int n)
    {
    int law = 0, high = n-1;
    while (law < high)
        {
        int temp = arr[law];
        arr[law] = arr[high];
        arr[high] = temp;
        law++;
        high--;
        }
    }
```

Remove Duplicates from a Sorted Array

I/P: arr[] = [10, 20, 20, 30, 30, 30, 30]
size = 7

O/P: arr[10, 20, 30, -, -, -, -]
size = 3

I/P: arr[] = [10, 10, 10]
size = 3

O/P: arr[] = [10, -, -]
size = 1

Method 1 Naive Solution

O(n) Time
O(n) Space

```
int removeDups(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;
}
```

Method 2

```
int removeDups(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++;
        }
    }
}
```

Left Rotate an Array by One

I/P : arr[] = [1, 2, 3, 4, 5]
O/P : arr[] = [2, 3, 4, 5, 1]

I/P : arr[] = [30, 5, 20]
O/P : arr[] = [5, 20, 30]

```
void leftRotateOne (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 d

I/p : arr[] = [1, 2, 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]

* We may assume d ≤ No. of element in the array

Method 1 (Naive) O nd

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

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

Method 2 — O(n) ← Complexity

I/P: arr[] = [1, 2, 3, 4, 5]
d = 2
O/P: arr[] = [3, 4, 5, 1, 2]

temp[] = [1, 2]
arr[] = [3, 4, 5, 4, 5]
arr[] = [3, 4, 5, 1, 2]

```
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];

}
```

Page No.:
Date:

Method 3 $\Theta(n)$

```
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--;
    }
}
```

arr[] = [1, 2, 3, 4, 5]

after first Reverse

arr[] = [2, 1, 3, 4, 5]

after second Reverse

arr[] = [2, 1, 5, 4, 3]

after third reverse

arr[] = [3, 4, 5, 1, 2]

Leader in an array

Let there is an array arr[] in which arr[i] can be leader if and only if right side element must be smaller than it.

1. I/P arr[] = [7, 10, 4, 10, 6, 5, 2]
   o/p: 10 6, 5 2

2. I/P: arr[] = [10, 20, 30]
   o/p: 30

3. I/P: arr[] = [30, 20, 10]
   o/p: 30, 20, 11

Method 1

```
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) {
      print(arr[i]) }
  }
}
```

1. I/p : arr[] = [7, 10, 4, 10, 6, 5, 2]
O/p : 2 5 6 10

Q(a) Method 2

```
void leaders (int arr[], int n)
{
    int current_ldr = arr[n-1];
    print(current_ldr);

    for(int i = n-2; i >= 0; i--)
    {
        if (current_ldr < arr[i])
        {
            current_ldr = arr[i];
            print(current_ldr);
        }
    }
}
```

Maximum Difference

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

I/P : arr[] = [2, 3, 10, 6, 4, 8, 1]
O/P : 8

I/P : arr[] = [7, 9, 5, 6, 3, 2]
O/P : 2

I/P : arr[] = [10, 20, 30]
O/P : 20

I/P : arr[] = [30, 10, 8, 2]
O/P : -2

Method 1 Naive:

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

Page No.:
Date:

Method 2 (Efficient)

```
int maxDiff (int arr[], int n)
{
int res = arr[1] - arr[0];
    minval = arr[0];

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

explain

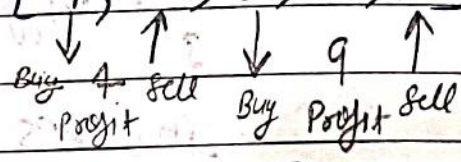
j = 1 : res = max(1, 3-2) = 1
minval = min(2, 3) = 2

j = 2 : res = max(1, 10-2) = 8
minval = min(2, 10) = 2

j = 3 : res = max(8, 6-2) = 8
minval = min(2, 6) = 2

Stock Buy & Sell Problem

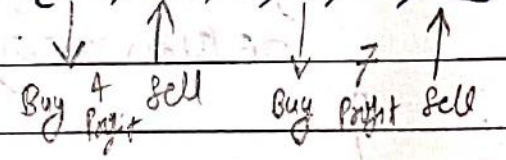
1. I/P : arr[] = [1, 5, 3, 8, 12]
   O/P : 13

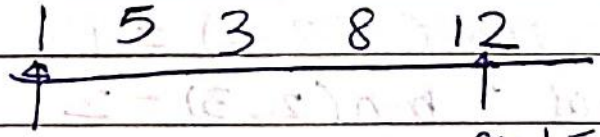


2. I/P : arr[] = [30, 20, 10]
   O/P : 0

3. I/P : arr[] = [10, 20, 30]
   O/P : 20

4. I/P : arr[] = [1, 5, 3, 1, 2, 8]
   O/P : 11





i = 0
j = 1

curr_profit = (5-1) + maxProfit(arr, 0, -1) + maxProfit(arr, 2, 4)

⇒ 4 + 0 + 9 = 13

i = 0
j = 2

curr_profit = (3-1) + maxProfit(arr, 0, -1) + maxProfit(arr, 3, 4)

Method 1 Naive

```
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[i] +
                maxProfit(price, start, i-1)
                + maxProfit(price, j+1, end);
                profit = max(profit, curr_profit);
            }
        }
    }
    return profit;
}
```

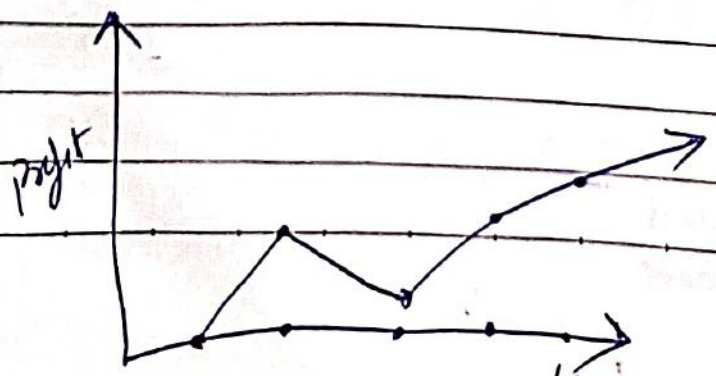
Stock Buy and Sell (Efficient Solution)

I/P : arr[] = [1, 5, 3, 8, 12]
O/P : 13

1 5 3 8 12
profit = 0

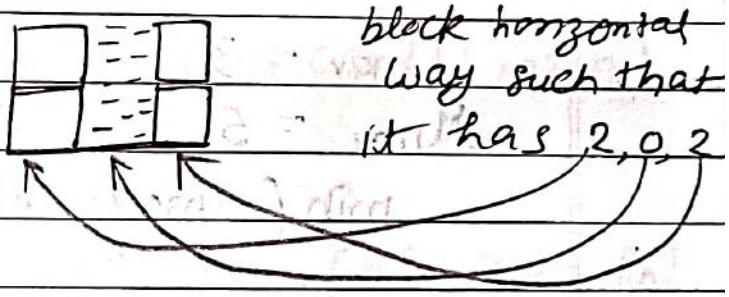
i = 1 : profit = 0 + (5-1) = 4
i = 2 :
i = 3 : profit = 4 + (8-3) = 9
i = 4 : profit = 9 + (12-8) = 13

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

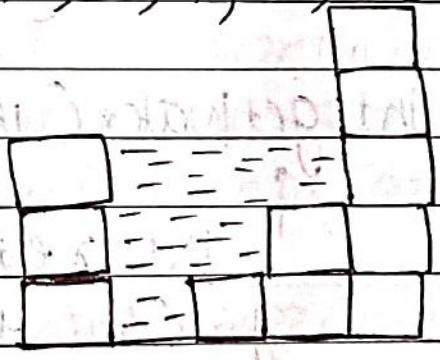


Trapping Rain Water

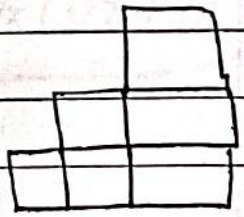
1. I/P : arr[] = [2, 0, 2]
O/P : 2



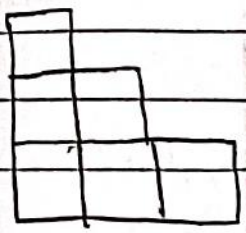
2. I/P : arr[] = [3, 0, 1, 2, 5]



3. I/P : arr[] = [1, 2, 3]
O/P : 0



4. I/P : arr[] = [3, 2, 1]
O/P : 0



I/P : arr[] = [3, 0, 1, 2, 5]

$l_{max} = 3$

$r_{max} = 5$

$\min(l_{max}, r_{max}) = 3$

$res[i] = \min(l_{max}, r_{max}) - arr[i]$

Method 1:

```
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[j]);

        res = res + (min(lmax, rmax) - arr[i]);
    }
    return res;
}
```

Complexity

→ $\Theta(n^2)$

Method 2 O(n)

arr[] : [5, 0, 6, 2, 3]
arr[] = [5, 5, 6, 6, 6]
arr[] = [6, 6, 6, 3, 3]
~~arr[]~~

```
int getWater(int arr[], int n)
{
    int res = 0;
    int lmax[n], 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]);
    return res;
}
```

Maximum consective 1s in a binary array

1. I/p : arr[] = [0, 1, 1, 0, 1, 0]
   o/p: 2

2. I/p : arr[] = [1, 1, 1, 1]
   o/p: 4

3. I/p : arr[] = [0, 0, 0]
   o/p: 0

4. I/p : arr[] = [1, 0, 1, 1, 1, 1, 0, 1, 1]
   o/p: 4

Method 1 Naive $O(n^2)$

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

arr[] = [0, 1, 1, 1, 0, 1, 1],
res = 0

i = 0, curr = 0
i = 1, curr = 3; res = 3
i = 2, curr = 2
i = 3, curr = 1
i = 4, curr = 0
i = 5, curr = 2
i = 6, curr = 1

Method 2 O(n)

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

arr[] = [0, 1, 1, 0, 1, 1, 1]

res = 0

curr = 0

i = 0 curr = 0

i = 1 curr = 1, res = 1

i = 2 curr = 2, res = 2

i = 3 curr = 0

i = 4 curr = 1

i = 5 curr = 2

i = 6 curr = 3, res = 3

Maximum Sum Subarray

I/P: arr[] = [2, 3, -8, 7, -1, 2, 3]
O/P: 11

I/P: arr[] = [5, 8, 3]
O/P: 16

I/P: arr[] = [-6, -1, -8]
O/P: -1

What is a subarray?

Subarrays of [1, 2, 3] are
[1], [2], [3], [1, 2], [2, 3] &
[1, 2, 3]

Method 1 Naive : $O(n^3)$

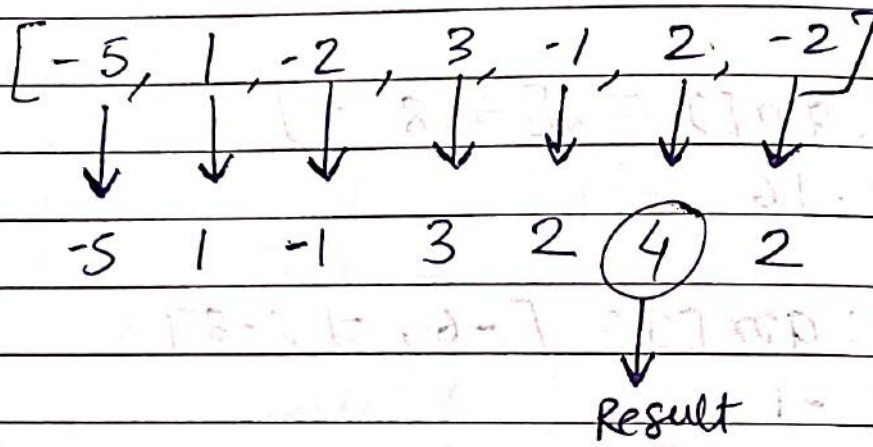
```
int maxSum(int arr[], int n)
{
    int res = arr[0];
    for(int i=0; i<n; i++)
    {
        int curr = 0;
        for(int j=0; j<n; j++)
        {
            curr = curr + arr[j];
            res = max(res, curr);
        }
    }
    return res;
}
```

[1, -2, 3, -1, 2]
res = 1
i = 0:
curr = 0
curr = 1
curr = 1-2 = -1
curr = -1+3 = 2, res = 2
curr = 2-1 = 1
curr = 1+2 = 3, res = 3
i = 1:
curr = 0
curr = -2
curr = -2+3 = 1
curr = 1-1 = 0
curr = 0+2 = 2

Kandane's Algorithm

Efficient : O(n)

Method



maxEnding(i) = max(maxEnding(i-1) + arr[i], arr[i])

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

[-3, 8, -2, 4, -5, 6]

MCS = -3

MaxEnding = -3

i=1 MaxEnding = max(-3+8, 8) = 8
MCS = 8

i=2 maxEnding = max(8-2, -2) = 6

i=3 maxEnding = max(6+4, 4) = 10
MCS = 10

i=4 maxEnding = max(10-5, -5) = 5

i=5 maxEnding = max(5+6, 6) = 11
MCS = 11

}

An interesting array problem to find the length of the longest subarray that has alternating even odd elements.

1. I/P : [10, 12, 14, 7, 8]
   O/P : 3

2. I/P : [7, 10, 13, 14]
   O/P : 4

3. I/P : [10, 12, 8, 4]
   O/P : 1

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

arr[] = [5, 10, 20, 6, 3, 8]
res = 1
i = 0 : curr = 2, res = 2
i = 1 : curr = 1
i = 2 : curr = 1
i = 3 : curr = 3, res = 3
i = 4 : curr = 2
i = 5 : curr = 1

```
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;
}
```

Kadane's Algo

Method 2 : O(n) :

arr[] = {5, 10, 20, 6, 3, 8}

i=1 : curr = 2, res = 2
i=2 : curr = 1,
i=3 : curr = 1
i=4 : curr = 2
i=5 : curr = 3, res = 3

```
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;
}
```

The task is to find maximum circular sum subarray of a given array. Two approaches are discussed, one is $O(n^2)$ & other $O(n)$

[10, 5, -5]

All circular subarray are:-

Normal [[10,], [5], [-5], [10, 5], [5, -5], [10, 5, -5]]

Only circular [[5, -5, 10], [-5, 10], [-5, 10, 5]]

1. I/P : arr[] = [5, -2, 3, 4]
   O/P : 12

2. I/P : arr[] = [2, 3, -4]
   O/P : 5

3. I/P : arr[] = [8, -4, 3, -5, 4]
   O/P : 12

4. I/P : arr[] = [-3, 4, 6, -2]
   O/P : 10

5. I/P : arr[] = [-8, 7, 6]
   O/P : 13

I/P : arr[] = [3, -4, 5, 6, -8, 7]
O/P : 17

Method 1 Naive Solution $O(n^2)$

arr[] = [5, -2, 3, 4]
res = 5

i = 0 : curr_max = 10, res = 10
i = 1 : curr_max = 10
i = 2 : curr_max = 12, res = 12
i = 3 : curr_max = 10

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

Method 2 : O(n)

Idea: Take the maximum of the following two.

→ Maximum sum of a Normal subarray (easy: Kandane algo.)

→ Maximum sum of a circular subarray (How to find this?)

arr[] = [5, -2, 3, 4] (subtract -2, 3)

arr[] = [8, -4, 3, -5, 4] (subtract -4, 3, -5)

arr[] = [3, -4, 5, 6, -8, 7] (subtract -8)

$[x_0, x_1 .. x_i \cdots x_{j+1} \cdots x_{i-1} \; x_i, x_{j+1} \cdots x_{n-1}]$

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

int overalMaxSum(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[i];
        arr[i] = -arr[i];
    }
    int max_circular = arr_sum + normalMaxSum(arr, n);
    return max(max_normal, max_circular);
}
```

Normal Sum

if it means every thing is negative

Circular Sum

Basically finding minimum sum in one

1. arr[] = [8, -4, 3, -5, 4]

max_normal = 8

arr_sum = 6

after inversion

arr[] = [-8, 4, -3, 5, -4]

max_circular = 6+6

= 12

res = max (max_normal, max_circular)

= max (8, 12)

= 12

Majority Element

Majority element is an element that appear more than n/2 times in an array of size n. In this we will study 2 methods to find majority element element in an array are discussed

ex If an array of size 5
then it must appear 3 times

If an array of size 6
then it must appear 4 times

1. I/P : arr[] = [8, 3, 4, 8, 8]
O/P : 0 or 3 or 4 (Any index of 8)

2. I/P : arr[] = [3, 7, 4, 7, 7, 5]
O/P : -1 (NO majority)

3 I/P : arr[] = [20, 30, 40, 50, 50, 50, 50]
O/P : 3 or 4 or 5 or 6 (Any index of 5)

Method 1 Naive ($O(n^2)$)

```
int findMajority (int arr[], int n)
{
    int 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 > n/2)
            return i;
    }
    return -1;
}
```

1. arr[] = [8, 7, 6, 8, 6, 6, 6, 6]

n = 8

i = 0; count = 2

i = 1, count = 1

i = 2, count = 5

Method 2 O(n)

moore's voting algorithm.

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

Check if candidate is actually a majority

eg. [6, 8, 4, 8, 8]

Initial : count = 1, res = 0

i = 1 : count = 0.
count = 1, res = 1

i = 2 : count = 0
count = 1, res = 2

i = 3 : count = 0
count = 1, res = 3

i = 4 : count = 2

Explanation : In the moore's algorithm they basically cancelled out the votes.

1. 6, 8, 7, 6, 6
✓ x ✓ x

(6 8, 7 6) (6)

cancelling each other

2. 8, 8, 6, 6, 6, 6

(8 8, 6 6) (6, 6)

cancelling each other.

Window Sliding Technique

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

1. arr[] = [1, 8, 30, -5, 20, 7]
   K = 3
   O/P : 45

2. I/P : arr[] = [5, -10, 6, 90, 3]
   K = 2
   O/P : 96

Method 1 Naive O((n-K) x K)

```
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;
```

Method 2

① compute first window sum.

I/P : arr[] = [1, 8, 30, -5, 20, 7]
o/p : 45

```
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;
```

Given an unsorted array of non-negative integers. find if there is a subarray with given sum.

I/p : arr[] = [1, 4, 20, 3, 10, 5]
sum = 33
o/p : Yes

I/p : arr[] = [1, 4, 0, 0, 3, 10, 5]
sum = 7
o/p : Yes.

I/p : arr[] = [2, 4]
sum = 3
o/p : No

Method 1 $O(n^2)$

```
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;
    }
```

Window Sliding Technique

Method 2 O(n)

for current window

s -> start

e -> end

```
bool isSubSum(int arr[], int n, int sum)
{
    int curr_sum = arr[0], s = 0;
    for (int e = 1; e < n; e++)
    {
        // Clean the prev. window
        while (curr_sum > sum &&
               s < e - 1)
        {
            curr_sum = curr_sum - arr[start];
            s++;
        }
        if (curr_sum == sum)
            return true;
        if (e < n)
            curr_sum += arr[e];
    }
    return curr_sum == sum);
}
```

N-bonacci Numbers Print first m n-bonacci number

eg.: I/P : N = 3, m = 8

O/P: 0, 0, 1, 1, 2, 4, 7, 1, 3

O(N×M)

I/P: N = 4, M = 10

O/P: 0, 0, 0, 1, 1, 2, 4, 8, 15, 29

Count distinct element in every window of size K

I/P: arr[] = [1, 2, 1, 3, 4, 3, 3]

K = 4

O/P: 3 4 3 2

O(n×K

Given a fixed array and multiple queries of following types on the array, have to efficiently perform these queries.

I/P: arr[] = [2, 8, 3, 9, 6, 5, 4]
i r

getSum(0, 2) // 13
getSum(1, 3) // 20
getSum(2, 6) // 27

prefix_sum[] = [2, 10, 13, 22, 28, 33, 37]

```
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];
```

if(i!=0) prefix_sum[r] - prefix_sum[i-1];
else
prefix_sum[r];

Given an array of integer, find if it has an equilibrium point.

1. I/p : arr[] = [3, 4, 8, -9, 20, 6]
   sum = 6 (3, 4, 8, -9) | 20 | 6
   o/p : yes

2. I/p : arr[] = [7, 2, -2]
   No element sum = 0 | 7 | (2, -2) sum = 0
   o/p : yes

3. I/p : arr[] = [2, -2, 4]
   sum = 0 (2, -2) | 4 | No element sum = 0

4. I/p : arr[] = [4, 2, 2]
   o/p : NO

Method 1 $O(N^2)$

```
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;
}
```

Method 2 O(N)

```
bool isEqPoint(int arr[], int n)
{
    int sum = 0;
    for(int i = 0; i<n; i++)
        sum += arr[i];

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

Given n ranges, find the maximum appearing element in these range.

Ip: L[] = [1, 2, 5, 15]
R[] = [5, 8, 7, 18]

Op: 5

1-5 [1, 2, 3, 4, (5)]
2-8 [2, 3, 4, (5), 6, 7, 8]
5-7 [(5), 6, 7]
15-18 [15, 16, 17, 18]

eg L[] = [1, 2, 3]
R[] = [3, 5, 7]

arr[] = [0, 1, 1, 1, 0 ...]
arr[] = [0, 1, 1, 1, -1, 0, -1, 0, ...]

arr[] after prefix sum [0, 1, 2, 3, 2, 2, 1, 1, 0 ...

```
int maxOcc(int L[], int R[],
           int n)
{
    Vector<int> arr[1000];
    for(int i=0; i<n; i++)
    {
        arr[L[i]]++;
        arr[R[i]+1]--;
    }
    int maxm = arr[0], ans=0;
    for(int i=1; i<1000; i++)
    {
        arr[i] += arr[i-1];
        if(maxm < arr[i])
        {
            maxm = arr[i];
            ans = i;
        }
    }
    return ans
}
```

Ch: 05
Searching

→ For Unsorted Array

O(n)

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

→ given a sorted array and an element x to be search, find index of x if present. Otherwise return -1

I/P : arr[] = {1, 10, 20, 40, 50, 70, 80}
find x = 10

O/P : 1 // x is present at index 1

```
bool binarySearch (int arr[], int l, int h, int x);
                   index low  index high  element to search
{
    int mid = (l+h)/2;
    // mid = l+(h-l)/2
    if (arr[mid]==x)
        return mid;
    if (arr[mid]>x)
        return binarySearch(arr, l, mid-1, x);
```

else
return binarySearch (arr, mid+1, h, x);

bool binarySearch (int arr[], int l, int h, int x)
{
if (low > high) return -1;
int mid = (l+h)/2;
// mid = l+(h-l)/2
if (arr[mid] == x)
return mid;
if (arr[mid] > x)
return binarySearch (arr, l, mid-1, x);
else
return binarySearch (arr, mid+1, h, x);

1. Given a sorted array with repetitions find left most index of an element

I/P: arr[] = [2, 3, 3, 3, 3]
x = 3
o/p: 1 (left most occorence)

```
bool binarySearch (int arr[], int l, int h,
                   int x)
{
    if (low > high)
        return -1
    int mid = (l+h)/2
    if (arr[mid] == x) && (mid == 0 || arr[mid-1]
                           != x)
        return mid;
    if (arr[mid] > x)
        return binarySearch (arr, l, mid-1, x);
    else
        return binarySearch (arr, mid+1, h, x);
```

Given a sorted array with repitations find right most index of an element

I/p: arr[] = [2, 3, 3, 3, 3]
x = 3
o/p: 4

```
bool binarySearch (int arr[], int l, int h,
                   int x, int n
{
    if (low > high)
        return -1;
    int mid = (l+h) / 2;
    if (arr[mid] == x && (mid == (n-1) ||
        arr[mid+1] != x))
        return mid;
    if (arr[mid] > x)
        return binarySearch (arr, l, mid-1,
                             x, n);
    else
        return binarySearch (arr, mid+1, h, x,
                             n);
}
```

3. Count 1's in a sorted binary array

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

o/p: 4

I/p: arr[] = [1, 1, 1, 1, 1]
o/p: 5

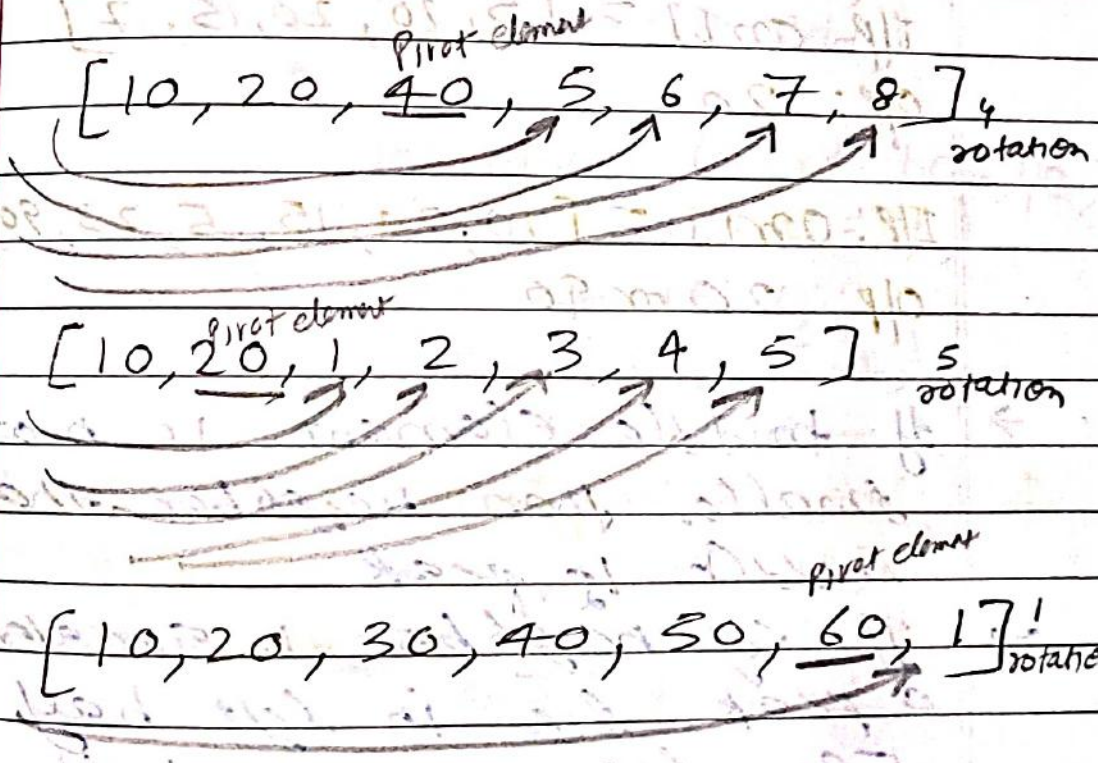
4. Given an infinite size sorted array and an element x, find if x is present in the array or not.

I/p: arr[] = [10, 20, 25, 50, 70, 80 ....]
x = 20

o/p: 1

```
int findInInfiniteArr (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;
    else return binarySearch (arr, i/2, i);
```

5. given a sorted and rotated array of distinct element & an element x, find if x is present in the array or not.



5 Peak Element

Not smaller than neighbour

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

I/p: arr[] = [10, 20, 15, 5, 23, 90, 67]
o/p: 20 or 90

→ If middle element is not smaller than neighbor than middle is peak
→ If left neighbor is greater than a peak lies in left half.
Else lies in right half.

7 Square Root : If I/p is not perfect square return floor of square root

I/p : n = 4
o/p : 2

I/p : n = 12
o/p : 3

```
if (x == 0 || x == 1)
    return x;

int start = 1, end = x;
while (start <= end)
{
    int mid = (start + end)/2;
    if (mid * mid == x)
        return mid;
    if (mid * mid < x)
    { start = mid + 1;
      ans = mid;
    }
    else
        end = mid - 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 equal to x.

→ I/P : arr[] := [3, 5, 9, 2, 8, 10, 11]
x = 17 ;

O/P : Yes

There is pair (9, 8)

→ I/P : arr[] = [8, 4, 6]
x = 11.

O/P : NO

$O(n^2)$

```
for (int i=0 ; i<n ; i++)
    for (int 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.

i/p: arr[] = [2, 5, 8, 12, 30]
n = 17

left = 0
right = n-1

o/p: Yes
pair is (5, 12)

```
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.

I/P : arr[] = [2, 3, 4, 8, 9, 20, 40]
n = 32

O/P : Yes
4 + 8 + 20

$O(n^3)$

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

for(int i=0; i<n; i++)
    if(isPair(a, i+1, n-1, x-a[i]))
                    low  high
        return true;   Sum
    return false;
```

4. Count Pair with given sum.
5. Count Triplets with given sum.
6. find if there is a triplet a, b, c such that $a^2 + b^2 = c^2$

Amazon Question.

Median of two sorted arrays

→ median of two sorted arrays

I/p : a1[] = [10, 20, 30, 40, 50]
a2[] = [5, 15, 25, 35, 45]
o/p : 27.5 // [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

I/p : a1[] = [1, 2, 3, 4, 5, 6]
a2[] = [10, 20, 30, 40, 50]
o/p : 6.0 // [1, 2, 3, 4, 5, 6, 10, 20, 30, 40, 50]

I/p : a1[] = [10, 20, 30, 40, 50, 60]
a2[] = [1, 2, 3, 4, 5]
o/p : 10.0 // [1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 60]

Algo time example

a1[] = [10, 20, 30], a2[] = [5, 15, 25]
temp[] = [10, 20, 30, 5, 15, 25]
after sorting
temp[] = [5, 10, 15, 20, 25, 30]
if $(a_1 + a_2)$ is odd then return middle of temp.
else return average of middle two element.

Efficient : $O(\log n_1)$ where $n_1 <= n_2$

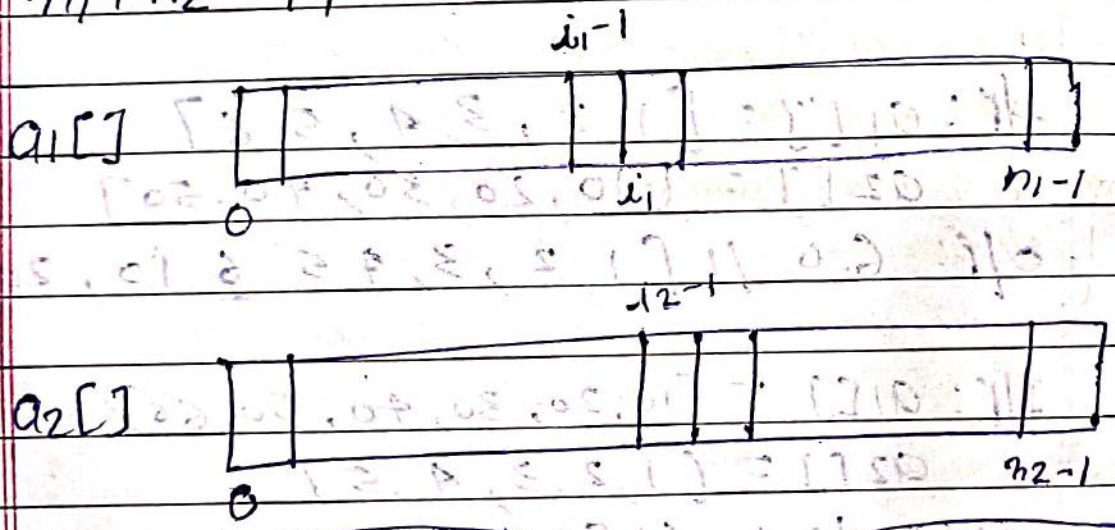
i_1

$n_1 = 5$ $a_1[] = [10, 20, 30, 40, 50]$

$n_2 = 9$ $a_2[] = [5, 15, 25, 35, 45, 55, 65, 75, 85]$

i_2

$n_1 + n_2 = 14$



$a_1[0 \ldots i_1 - 1]$
$a_2[0 \ldots i_2 - 1]$

Left half

$a_1[i_1 \ldots n_1 - 1]$
$a_2[i_2 \ldots n_2 - 1]$

Right half

$$i_2 = \left| \frac{n_1 + n_2 + 1}{2} \right| - i_1$$

$$= 7 - 2$$

$$= 5$$

min1: minimum element in right side of a_1

max1: maximum element in left side of a_1

min2: minimum element in right side of a_2

max2: maximum element in left side of a_2

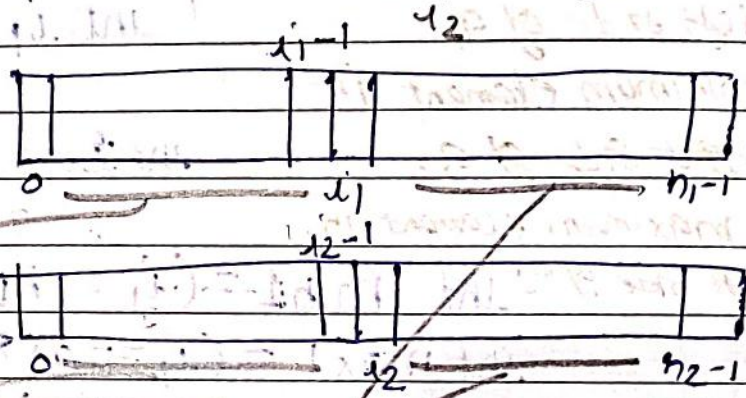
```
double getMed(int a1[], int a2[],
              int n1, int n2)
{ int begin1 = 0, end1 = n1
  while (begin1 <= end1)
  {
    int i1 = (begin1 + end1)/2
    int i2 = (n1 + n2 + 1)/2 - i1

    int min1 = (i1 == n1) ? INT_MAX : a1[i1];
    int max1 = (i1 == 0) ? INT_MIN : a1[i1-1];
    int min2 = (i2 == n2) ? INT_MAX : a2[i2];
    int max2 = (i2 == 0) ? INT_MIN : a2[i2-1]

if (max1 <= min2 && max2 <= min1)
    {
      if ((n1 + n2) % 2 == 0)
        return (max(max1, max2) +
                min(min1, min2))/2
      else
        return (max(max1, max2));
    }
  else if (max1 > min2) end1 = i1 - 1;
  else begin1 = i1 + 1;
```

Page No.:
Date:

efficient : $O(\log n_1)$ where $n_1 <= n_2$

$n_1 = 5$ $a_1[] = [\overset{0}{10}, \overset{1}{20}, \overset{2}{30}, \overset{3}{40}, \overset{4}{50}]$ i_1

$n_2 = 9$ $a_2[] = [\overset{0}{5}, \overset{1}{15}, \overset{2}{25}, \overset{3}{35}, \overset{4}{45}, \overset{5}{55}, \overset{6}{65}, \overset{7}{75}, \overset{8}{85}]$ i_2



$a_1[0 \ldots\ldots i_1-1]$
$a_2[0 \ldots\ldots i_2-1]$
left side

$a_1[i_1 \ldots\ldots n_1-1]$
$a_2[i_2 \ldots\ldots n_2-1]$
right side

$$i_2 = \left| \frac{n_1 + n_2 + 1}{2} \right| - i_1$$

let take $i_1 = 2$ (index) then index of i_2

$= 7 - 2$

$= 5$ index.

→ In left side we get all number smaller than right side

→ In right side we get all number — Large than left side.

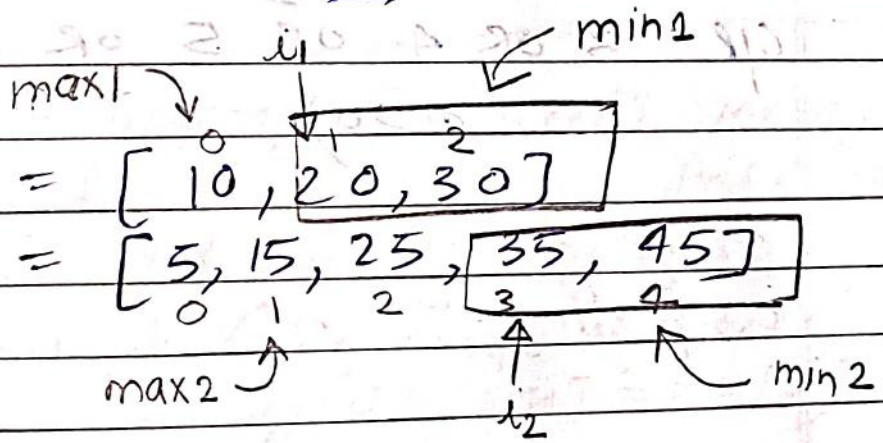
eg $a_1[] = [10, 20, 30, 40, 50]$

$a_2[] = [5, 15, 25, 27, 28, 55, 65, 75, 85]$

max of left side

min of right side

median $= \frac{28 + 30}{2}$



$i_1 = 1$

$i_2 = (3+5-1)/2 = 3$

minimum element in right side of a_1 — int min1 = $a_1[i_1]$

maximum element in left side of a_1 — int max1 = $a_1[i_1 - 1]$

minimum element in right side of a_2 — int min2 = $a_2[i_2]$

maximum element in left side of a_2 — int max2 = $a_2[i_2 - 1]$

```
if (max1 <= min2 && max2 <= min1)
{   if ((n1+n2)%2 == 0)
        return (max(max1, max2) +
                min(min1, min2))/2;
    else return (max(max1, max2));
}
else if (max1 > min2) end1 = i1 - 1;
```

Majority Element

I/p : arr[] = [8, 3, 4, 8, 8]
o/p : 0 OR 3 OR 4 (Any Index of 8)

I/p : arr[] = [3, 7, 4, 7, 7, 5]
o/p : -1 (No majority)

I/p arr[] = [20, 30, 40, 50, 50, 50, 50]
o/p : 3 OR 4 OR 5 OR 6 (Any Index of 50)

array of size 5 then must have atleast 3
array of size 6 then must have atleast 4

O(n²) Naive solution

```
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 > n/2)
      return i
  }
  return -1;
}
```

Efficient O(n)

arr[] = [8, 8, 6, 6, 6, 4, 6]

Initially: count = 1, cw = 0

i = 1; count = 2

i = 2; count = 1

i = 3; count = 0

count = 1, cw = 3

i = 4, count = 2

i = 5: count = 1

i = 6 : count = 2

```
int findmajority (int arr[], int n)
{                                   find a candidate
    int cw = 0, count = 1;
    for (int i = 1; i < n; i++)
    { if (arr[cw] == a[i])
        count++;
      else
      if (count == 0)
        { cw = i; count = 1; }
    }
```

```
count = 0;                          check if the
for (int i=0 ; i<n ; i++)           candidate is
    if (arr[cw] == arr[i])          actually
        count++;                    a majority.
    if (count <= n/2)
        cw = -1;
    return cw;
```