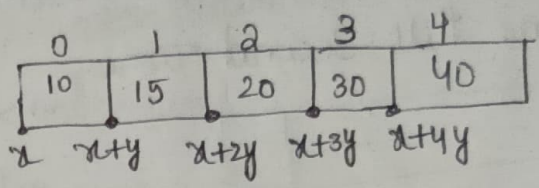
18/4/23 # Array Data Structures

Introduction:

arr[] = [10, 15, 20, 30, 40]

```
a = arr[0];
b = arr[2]; // x+2y
```

Contiguous Memory:



x → Address where Array is stored
y → size of an array element

Advantages:

1) Random Access ⇒ constant time
2) cache friendliness

Array D.S (Types)

- fixed sized Arrays
- Dynamic sized Arrays

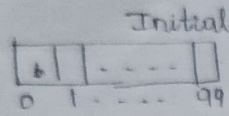
C/C++

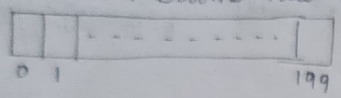
```
int arr[100]
int arr[n]      } stack Allocated
int *arr = new int[n]   } Heap Allocated
int arr[] = [10,15,30,40]   } Stack Allocated
```

Dynamic Sized Arrays:

Resize Automatically.

c++ : vector
Java : ArrayList
python : list





Vectors in C++

⇒ Problems with simple arrays:

```
int arr[100];
int arr[n]
int * arr = new int[n];
```

Vector advantages:-

- Dynamic Size
- Rich library function
  - find
  - erase
  - insert etc
- easy to know size ⇒ Arrays:
  ```
  int n = sizeof(arr)/sizeof(arr[0]);
  ```
  Vectors:
  ```
  int n = V.size();
  ```
- No need to pass size ⇒ Arrays:
  ```
  int fun(int *arr, int n)
  {
  }
  ```
  Vectors:
  ```
  int fun(vector<int>V)
  {
  }
  ```

- can be returned from a function

```
int * fun()
{
    int arr[100];
    return arr;
}
```

Simply array will be deallocated after returning

```
int * fun()
{
    int *arr = new int[100];
    return arr;
}
```

```
Vector<int> fun()
{
    Vector<int> V;

    return V;
}
```

- By default initialised with default values.
- we can copy a vector to other.

$V_1 = V_2$;

Operations on Arrays

Search (unsorted Array):-

I/P: arr[] = {20, 5, 7, 25}
n = 5
O/P : 1

I/P: arr[] = {20, 5, 7, 25}
x = 15
O/P : -1

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

T.C = O(n)

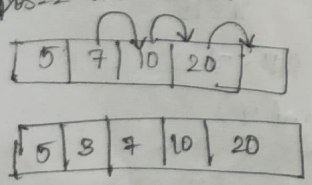
Insert:

⇒ I/P: arr[] = {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}

```
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);
}
```

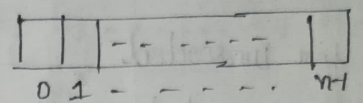
pos=2 x=3 idx=1

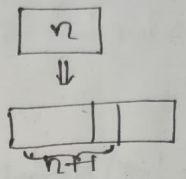


Time Complexity : O(n)
Insert At the End : O(1)
Insert At the Beginning : θ(n)

Insert at the end for Dynamic sized Array:-

Initial capacity : n





Time Complexity of every insert for first n inserts : ⇒ O(1)

Average Time Complexity for (n+1) Inserts at the end = $\frac{\overbrace{O(1)+O(1)\cdots O(1)}^{n}+\overbrace{\theta(n)}^{(n+1)}}{n+1}$

= O(1)

Deletion:

⇒ I/P: arr[] = {3, 8, 12, 5, 6} ⇒ I/P: arr[] = {3, 8, 12, 5, 6}
x = 12 x = 6
O/P: arr[] = {3, 8, 5, 6, –} O/P: arr[] = {3, 8, 12, 5, –}

```
int deleteEle (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);
}
```

Operations on Arrays:

Insert : O(n)

Search : O(n) for unsorted

O(log n) for sorted

Delete : O(n)

Get i-th Element : O(1)

Update i-th Element : O(1)

NOTE: Insert at the end and delete from the end can be done in O(1) time.

Largest Element in an Array:

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

o/p: 2 // Index of 20.

I/p: arr[] = {40, 8, 50, 100}

o/p: 3 // Index of 100

Naive Approach

```
int getLargest(int arr[], int n)
{
  for (int i=0; i<n; i++)
  {
    bool flag = true;
    for (int j=0; j<n; j++)
    {
      if (arr[j] > arr[i])
      {
        flag = false;
        break;
      }
    }
    if (flag == true)
      return i;
  }
  return -1;
}
```

```
int main()
{
  int arr[] = {5, 8, 20, 10};
  cout << getLargest(arr, 4);
  return 0;
}
```

Efficient Approach:

a_0 a_1 a_2 a_3 ..... a_{i-1} a_i .....

$a_{largest}$

$a_i <= a_{largest}$: Ignore

$a_i > a_{largest}$: largest = i

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

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

res = 0:

i=1 : res=1

i=2 : res=2

i=3 : res=2

⇒ Θ(n)

Second Largest Element:

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

o/p: 0 // Index of 10

I/p: arr[] = {20, 10, 20, 8, 12}

o/p: 4 // Index of 12.

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

o/p: -1 // no second largest

Naive Approach:

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

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

⇒ arr[] = {5, 20, 12, 20, 10}
largest = 1
res = 0
res = 2 ✓

⇒ arr[] = {10, 10, 10}
largest = 0
res = -1

Efficient Approach:

```
int SecondLargest(int arr[], int n)
{
  int res = -1, 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;
}
```

Tc = θ(n)
AS = θ(1)

a_0 a_1 a_2 ..... a_{i-1} a_i, ...

(a_0 … a_{i-1}: largest res.)

a[i] > a[largest] : res = largest, largest = i
a[i] == a[largest] : Ignore ⇒ 5, 8, 12, 12 (a_{i-1}, a_i)
a[i] < a[largest]
  → res == -1 : res = i
  → a[i] <= a[res] : Ignore
  → a[i] > a[res] : res = i

5, 8, 12, 20 (a_{i-1}, a_i)

a[i] > a[largest] ⇒ res = largest ⇒ largest = i

5, 8, 12, 7

↳ 12, 12, 12, 7 ⇒ res = -1 (a_{i-1}, a_i) ⇒ res = i

↳ 12, 8, 12, 7 (or) 12, 7, 12, 7 (a_{i-1}, a_i)
a[i] <= a[res] : Ignore.

↳ 12, 7, 12, | 9
a[i] > a[res] ⇒ res = i

⇒ {5, 20, 12, 20, 8}
largest = 0, res = -1
i=1 : res = 0, largest = 1
i=2 : res = 2
i=3 :
i=4

Check if an array is Sorted:

I/P: arr[] = {8, 12, 15}
O/P: yes

I/P: arr[] = {8, 10, 10, 12}
O/P: yes

I/P: arr[] = {100}
O/P: yes

I/P: arr[] = {100, 20, 200}
O/P: No

I/P: arr[] = {200, 100}
O/P: NO

Naive Method

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

{5, 20, 80, 100}

{7, 20, 30, 10} (i=1, i=3)
i=0:
i=1: return false.

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

Efficient Method

$\{a_0, a_1, a_2, a_3 \ldots a_{i-1}, a_i \ldots a_{n-1}\}$

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

{5, 12, 30, 2, 35} (0 1 2 3 4)
i=1:
i=2:
i=3: return false

T.C = $O(n)$ | AS = $\theta(1)$

Reverse an Array

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

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

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

Swap arr[low] and arr[high]

{10, 5, 7, 30}
low=0, high=3
{30, 5, 7, 10}
low=1, high=2
{30, 7, 5, 10}

T.C = $\theta(n)$
A.S = $\theta(1)$

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

Naive Approach

```
int remDups(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;
}
```

O(n)
O(n)

T.C = O(n)
A.S = O(n)

arr[] = {10,20,20,30,30,30}
temp[] = {10,-,-,-,-,-}, res = 1

i=1: temp[] = {10,20,-,-,-,-}, res = 2
i=2:
i=3: temp[] = {10,20,30,-,-,-}, res = 3
i=4:
i=5: 0
arr[] = {10,20,30,-,-,-}

Efficient Solution:

O(n) Time
O(1) Space

```
int remDups(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;
}
```

arr[] = {10,20,20,30,30,30}
        0 1 2 3 4 5

res = 1
i=1: arr[] = {10,20,20,30,30,30}, res = 2
i=2: arr[] = {10,20,20,30,30,30}, res = 2
i=3: arr[] = {10,20,30,30,30,30}, res = 3
i=4: arr[] = {10,20,30,30,30,30}, res = 3
i=5: arr[] = {10,20,30,30,30,30}, res = 3

Move all zeros 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}

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

Naive Solution

```
Void moveToEnd (int arr[], int n)
{
  for (int 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])
    }
  }
}
```

T.C = O(n²)

{10, 5, 0, 0, 8, 0, 9, 0}

i=0:
i=1:
i=2: Swap(arr[i], arr[j]) j=4
i=3: Swap(arr[i], arr[j]) j=6
i=4:
i=5:
i=6:
i=7:

Effecient Solution:

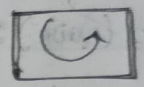
```
Void moveZeros (int arr[], int n)
{
  int count = 0;
  for (int i=0; i<n; i++)
  {
    if (arr[i] != 0)
    {
      Swap (arr[i], arr[count]);
    } count++;
  }
}
```

a0 a1 ... a(i-1) ai

a[i] == 0
a[i] != 0

{10, 8, 0, 0, 12, 0} count = 0
i=0 : {10, 8, 0, 0, 12, 0}, count = 1
i=1 : - - - - - -, count = 2
i=2 : - - - - - - - -, count = 2
i=3 : - - - - - - - -, count = 2
i=4 : {10, 8, 12, 0, 0, 0}, count = 3
i=5 : - - - - - - - -, count = 3

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

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

T.C = Θ(n)
A.S = Θ(1)

Left Rotate an Array by d

I/P: arr[] = {1, 2, 3, 4, 5} ⟲ Counter clockwise
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 elements in the array.

method 1 (Naive):

T.C = θ(nd)
θ(1) space

```
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++)
    leftRotateOne (arr, n);
}
```

arr[] = {1, 2, 3, 4, 5}
d=2

i=0 arr[] = {2, 3, 4, 5, 1}

i=1 arr[] = {3, 4, 5, 1, 2}

method-2 (Better):

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

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

T.C = θ(d + n-d + d)
= θ(n+d)
= θ(n)
A.S = θ(d)

```
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]
}
```

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

method-3 (Reversal algorithm):

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

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

T.C = θ(d + n-d + n)
= θ(2n) = θ(n)
A.S = θ(1)

```
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 1st Reverse
arr[] = {2, 1, 3, 4, 5}
After 2nd Reverse
arr[] = {2, 1, 5, 4, 3}
After 3rd Reverse
arr[] = {3, 4, 5, 1, 2}

Leaders in an array [leader has to be greater than all elements on right side; equals are not allowed]

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

Naive Solution ⇒ T.C = O(n²)

```
void Leaders (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]);
    }
  }
}
```

Efficient Solution:

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

O/p: 2 5 6 10 (Right to left O/P getting)

T.C = Θ(n)
A.S = Θ(1)

```
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);
    }
  }
}
```

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

Naive: T.C = O(n²), A.S = θ(1)

I/p: arr[] = {2, 3, 10, 6, 4, 8, 1}

O/p: 8

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

Efficient: T.C = θ(n), A.S = θ(1)

```
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;
}
```

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

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

minval = arr[0] = 2

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

Frequencies in a Sorted Array:

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

O/p:
10 3
25 1
30 2

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

O/p: 10 4

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

O/p:
10 1
20 1
30 1

```
Void printfreq (int arr[], int n)
{   int freq=1, i=1;    // pseudo code
    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);
}
```

both are stopped at same time

T.C = θ(n)

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

Initially: freq = 1

i=1 : freq = 2

i=2 : freq = 3

i=3 : print (10,3), freq = 1 => 10 3

i=4 : freq = 2

i=5 : print (30,2), freq = 1 => 30 2

After loop:

print (40,1) => 40 1

=> arr[] = {40, 50, 50, 50}

i=1 : print (40 1), freq = 1

i=2 : freq = 2

i=3 : freq = 3

i=4 : print (50 , 3), freq = 1

freq in Unsorted Array?

Stock Buy and Sell (Naive Solution)

(B) (S)

=> I/p: arr[] = [1, 5, 3, 8, 12]

B S B S

4 9

O/p: 13

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

O/p: 0

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

O/p: 20

=> I/p: arr[] = [1, 5, 3, 1, 2, 8]

B S B S

4 7

O/p: 11

s=0 e=n-1

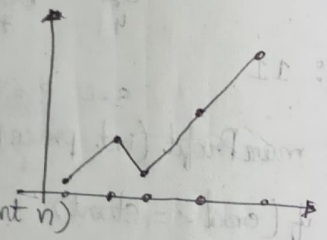
```
int maxProfit (int price[], int start, int end)
{
    if (end <= start)
        return 0;
    int profit = 0;
    for (int i = start; i < end; i++)
    {
        for (int j = i+1; j <= end; j++)
        {
            if (price[j] > price[i])
            {
                int curr_profit = price[j]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

1	5	3	8	12
↑ start = 0				↑ end = 4

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)
= 2 + 0 + 4
= 6

I/p: arr[] = [1,5,3,8,12]



O/p: 13

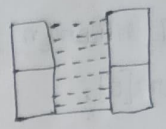
```
int maxProfit(int price[], int n)
{
    int profit = 0;
    for (int i=1; i<n; i++)
        if (price[i] > price[i-1])
            profit += (price[i] - price[i-1]);
    return profit;
}
```

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

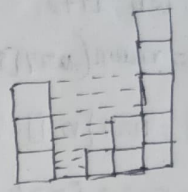
Trapping Rain water:

I/p: arr[] = [2,0,2]



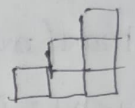
O/p: 2

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



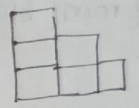
O/p: 6

I/p: arr[] = [1,2,3]



O/p: 0

I/p: arr[] = [3,2,1]

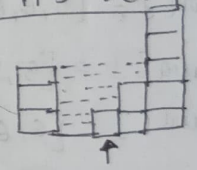


O/p: 0

T.C = O(n²) A.S = O(1)

Naive Solution:

```
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;
}
```



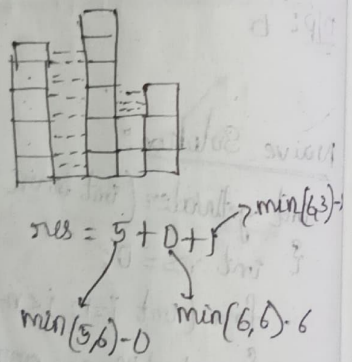
lMax = 3
rMax = 5
min(lMax, rMax) = 3
res[i] = min(lMax, rMax) - a[i]

efficient

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

T.C = $\theta(n)$
A.S = $\theta(n)$

arr[] = [5,0,6,2,3]
lmax[] = [5,5,6,6,6]
rmax[] = [6,6,6,3,3]



Maximum Consecutive 1's

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

I/p: arr[] = [1,1,1,1]
O/p: 4

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

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

Naive solution :-

T.C = $O(n^2)$
A.S = $O(1)$

```
int maxConsecutiveOnes(bool arr[], int n)
{
  int res = 0;
  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 = max(res, curr);
  }
  return res;
}
```

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

Efficient Solution:

TC = Θ(n)
AS = O(1)

```
int maxConsecutiveOnes (bool 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;
}
```

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

i=0, curr = 0

i=1, curr = 1, res = 1

i=2, curr = 2, res = 2

i=3, curr = 0, ~~res = 2~~

i=4, curr = 1, ~~res = 2~~

i=5, curr = 2 ~~res = 2~~

i=6, curr = 3, res = 3

22. Maximum Subarray Sum

What is a Subarray?

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

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

Naive:

TC: O(n²)
AS: O(1)

```
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

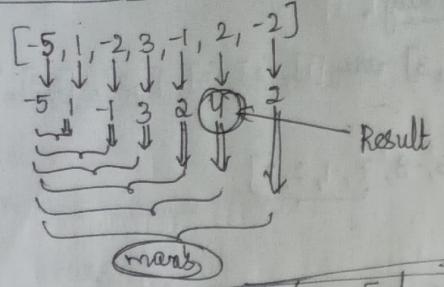
i=0:

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

Efficient soln: T.C = O(n) A.S = O(1)



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

```
⇒ int maxSum (int arr[], int n)
{
  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]

res = -3, maxEnding = -3

i=1: maxEnding = max(-3+8, 8) = 8
res = 8

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

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

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

i=5: maxEnding = max(5+6, 6) = 11
res = 11

23. Longest Even Odd Subarray

I/p: [10, 12, 14, 7, 8]

O/p: 3

I/p: [7, 10, 13, 14]

O/p: 4

I/p: [10, 12, 8, 4]

O/p: 1

Naive Solution: T.C = O(n²) A.S = O(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;
}
```

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

Effecient solution: (Kadanes algorithm) TC = O(n) A.S = O(1)

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

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

24: Maximum circular Subarray Sum

[10, 5, -5]

All circular subarrays are

Normal subarrays: [10] [5] [-5] [10,5] [10,5] [5,-5] [10,5,-5]

Non-circular subarrays: [5,-5,10] [5,10] [-5,10,5]

=> I/P: arr[] = [5, -2, 3, 4] => I/P: arr[] = [-8, 7, 6]

O/P: 12 O/P: 13

=> I/P: arr[] = [2, 3, -4] => I/P: arr[] = [3, -4, 5, 6, -8, 7]

O/P: 5 O/P: 17

=> I/P: arr[] = [8, -4, 3, -5, 4] => I/P: arr[] = [-3, 4, 6, -2]

O/P: 12 O/P: 10

Naive Solution: TC = O(n^2) A.S = θ(1)

```
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;
}
```

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

for every element, 4 subarrays are there including current

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

Efficient Solution: O(n)

Idea: Take the maximum of the following two

① Maximum Sum of a Normal Subarray (Easy: kadane)

② Maximum Sum of a circular Subarray (How to find this?)
also (modified kadane)

$[a_0, a_1, \ldots a_i, a_{i+1}, \ldots a_{j-1}, a_j, a_{j+1} \ldots a_{n-1}]$

circular (connecting last element with the first element)

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

Idea:

Maximum Sum of a circular Subarray = Sum of all the elements of array − minSum of Normal Subarray (modified kadane)

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

=> arr[] = [3, -4, 5, 6, -8, 7]

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

Standard kadane's

```
int overallMaxSum(int arr[], int n)
{
    int max_normal = normalMaxSum(arr, n);   => Normal Sum
    if (max_normal < 0)
        return max_normal;

    int arr_sum = 0;
    for (int i=0; i<n; i++)                  => circular sum
    {
        arr_sum += arr[i];
        arr[i] = -arr[i];
    }
    int max_circular = arr_sum + normalMaxSum(arr, n);
    return max(max_normal, max_circular);
}
```

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

8, -4, 3, -5, 4
↓ ↓ ↓ ↓ ↓
(8) 4 7 2 6

max_normal = 8

arr_sum = 6

After inversion:

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

-8, 4, -3, 5, -4
↓ ↓ ↓ ↓ ↓
8 4 1 (6) 2

max_circular = 6 + 6 = 12

res = max(max_normal, max_circular)
= max(8, 12)
= 12

arr[] = [-5, -3]

max_normal = -3

If we remove the condition if(max_normal<0)

arr_sum = -11

arr[] = [8, 3]

max_circular = -11 + 11 = 0

res = max(-3, 0) = 0

It returns 0, so that's why condition is imp.

T.C = $\theta(n+n+n) = \theta(3n) = \theta(n)$

25. Majority Element

↳ elements appear in the array more than n/2 times

⇒ I/p: arr[] = [8, 3, 4, 8, 8] (0 1 2 3 4)

o/p: 0 OR 3 OR 4 (any Indexs of 8)

⇒ I/p: arr[] = [3, 7, 4, 7, 7, 3]

o/p: -1 (No majority)

⇒ I/p: arr[] = [20, 30, 40, 50, 50, 50, 50] (0 1 2 3 4 5 6)

o/p: 3 OR 4 OR 5 OR 6 (any index of 50)

Naive: T.C = $O(n^2)$, A.S = $\theta(1)$

```
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;
}
```

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

n = 8, n/2 = 8/2 = 4

i=0 count = 2

i=1 count = 1

i=2 count = 5

o/p: 2

efficient: T.C = O(n), A.S = O(1)

(Moore's Voting Algorithm)

```
int findMajority (int arr[], int n)
{
  int res=0, count=1;
  for (int i=1; i<n; i++)        // find a
  {  if (arr[i] == arr[res])     // candidate
       count++;
     else
       count--;
     if (count == 0) {res=i; count=1;}
  }
  count=0;
  for (int i=0; i<n; i++)
     if (arr[res] == arr[i])     // check if the
       count++;                  // candidate is
                                 // actually a
  if (count <= n/2)              // majority
     res=-1
  return res;
}
```

arr[] = [8, 8, 6, 6, 6, 4, 6]
0 1 2 3 4 5 6

Initially: count=1, res=0

i=1 : count=2

i=2 : count=1

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

i=4 : count=2

i=5 : count=1

i=6 : count=2

arr[] = [6, 8, 4, 8, 8]
0 1 2 3 4

Initially : 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

working of algorithm

traversing L→R & maintaining count & majority var's

while traversing
- some elements increment count
- some elements decrement count,
- -s elements decrement the count will cancel the votes of others

ex-1: 6, 8, 7, 6, 6 - - -

(6 ✓, 8 ✗, 7 ✓, 6 ✗, 6 ✓)

increment count, decrement count

To show majority: dividing elements into 2 sets

cancel out each other: 6,8 and 7,6

not cancel out each other: 6

ex-2: 8, 8, 6, 6, 6, 6 - - -

(8 ✓, 8 ✓, 6 ✗, 6 ✗, 6 ✓, 6 ✓)

{8,8 ; 6,6} and {6,6}

To prove its not a majority set contains pairs of elements & every set contain diff pairs

n → x_1 x_2 ; x_2 x_1 ; x_3 x_1 ⇒ no element can appear more than n/2 times (atmost)

26. Minimum Consecutive Flips

Minimum Group Flips to Make Same

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

O/p: from 2 to 4

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

O/p: from 1 to 3
from 5 to 6

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

O/p:

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

O/p: from 0 to 0
from (or) 1 to 1

~~Naive => O(n²)~~

Group Counts Differ by One:

1 1 0 0 0 1 1 1 0 0 1 => 1's (Group counts) = 3, 3-2=1
0 0 1 1 0 0 0 1 1 0 0 0's (" ") = 2
1 1 1 1
0 0 0 0

Group Counts are Same:

0 0 1 1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1 1 1 0

Only Two possibilities;

Idea:
If array 1st element & last element are same => then Group Count differby 1
" " " different => differ by same

Always flip Second group elements

Efficient Solution: TC = O(n), AS = O(1)

```
Void printGroups (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;
}
```

0 0 1 1 0 0 1 1 0 1

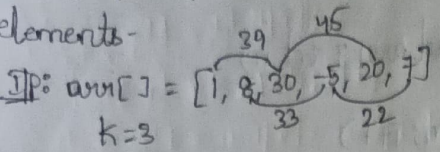
i=1:
i=2: From 2 to
i=3:
i=4: 3 (newline)
i=5:
i=6: from 6 to
i=7:
i=8: 7 (newline)
i=9: from 9 to 9

O/p: From 2 to 3
From 6 to 7
From 9 to 9

27. Window Sliding Technique

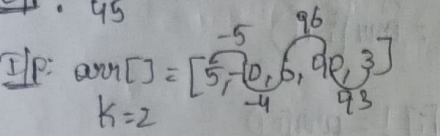
Given an array of integers and a number k, find the maximum sum of k consecutive elements-

I/P: arr[] = [1, 8, 30, -5, 20, 7]
k=3



O/P: 45

I/P: arr[] = [5, -2, 6, 9, 3]
k=2



O/P: 96.

Naive:

```
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;
```

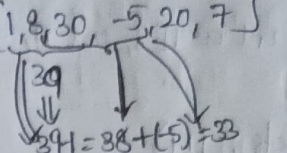
T.C = O((n-k) x k)

Imagine => k = n/2

O((n - n/2)(n/2))

O(n²/4) => T.C = O(n²)

I/P: arr[] = [1, 8, 30, -5, 20, 7]
k=3



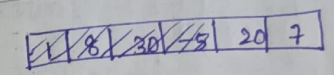
O/P: 45

[10, 5, -2, 20, 1]
K=3

max_sum
i=0, sum = 13, res = 13
i=1, sum = 23, res = 23
i=2, sum = 19

Idea of sliding window:

Compute the sum of current window using sum of previous window in O(1) Time



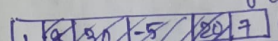
K=4

Initially: curr = 1+8+30-5 = 34, res = 34

1st slide: curr = 34+20-1 = 53, res = 53

Adding last of current window

Removing first of previous window.

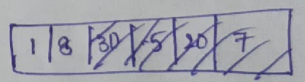


window

2nd slide: curr = 53+7-8 = 52

Adding last of current window

Removing first of previous window.



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

T.C = $\theta(n)$
A.S = $\theta(1)$

[1, 8, 30, -5, 20, 7]

k=4

After I st loop : curr = 34
res = 34

2nd loop I st iteration i=4

curr = 34+20-1
= 53

res = 53

II nd Iteration i=5

curr = 53+7-8
= 52

Subarray with Given Sum :

[There are no negative elements in the array]

I/P: [1, 4, 20, 3, 10, 5]
sum = 33
O/P : Yes

I/P: [1, 4, 0, 0, 3, 10, 5]
sum = 7
O/P: Yes

I/P: [2, 4] sum = 3
O/P No

Naive Solution:

T.C = $O(n^2)$
A.S = $O(1)$

```
function isSubSum (int arr[], int n, int sum)
{ for (i=0; i<n; i++)
  { int curr=0;
    for (j=i; j<n; j++)
    { curr += arr[j];
      if (curr == sum)
        return true;
    }
  }
  return false;
}
```

[3, 2, 0, 4, 7] sum=6

i=0; j=0 : curr=3
j=1 : curr=5
j=2 : curr=5
j=3 : curr=9
j=4 : curr=12

i=1 j=1 ; curr=2
j=2 : curr=2
j=3 : curr=6
sum=6

return true

Idea for efficient solution:

We use window sliding technique with a window of variable size.

[1, 4, 20, 3, 10, 5]

Sum = 33

S=0, e=0, curr=1
S=0, e=1, curr=5
S=0, e=2, curr=25
S=0, e=3, curr=28
S=0, e=4, curr=38

} while curr is smaller than sum, extend the window by increasing e.

S=1, e=4, curr=37
S=2, e=4, curr=33

} while curr is greater, shrink the window by increasing s

efficient solution:

T.C = O(n)
A.S = O(1)

```
function subSum(arr, sum)
{
  int s=0, curr=0;
  for(int e=0; e<arr.length; e++)
  {
    curr += arr[e];
    while (sum < curr)
    {
      curr -= arr[s];
      s++;
    }
    if (curr == sum)
      return true;
  }
  return false;
}
```

Removing & adding an item takes O(1) time & every element is removed & added atmost once

[4, 8, 12, 5]
sum = 17
s=0, e=0, curr=4
e=1, curr=12
e=2, curr=24
s=1, e=2, curr=20
s=2, e=2, curr=12
s=2, e=3, curr=17
return true

Prefix Sum:

Given a fixed array and multiple range sum queries, how to answer the queries efficiently.

I/P: arr[] = [2, 8, 3, 9, 6, 5, 4]
(indices 0 1 2 3 4 5 6)

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

O/P: 13 20 27

Naive solution:

T.C = θ(r-l+1) (or) O(n)
A.S = O(1)

```
int getSum(int l, int r)
{
  int res=0;
  for(int i=l; i<=r; i++)
    res += arr[i];
  return res;
}
```

arr = [2, 8, 3, 9, 6, 5, 4]
(indices 0 1 2 3 4 5 6)

getSum(1,3):
Initially: res=0
i=1: res=8
i=2: res=11
i=3: res=20

Idea for getSum() in O(1) Time

arr = [2, 8, 3, 9, 6, 5, 4]

① precompute prefixSum Array

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

$pSum[i] = \sum_{0}^{i} arr[i]$

② $getSum(l,r) = \begin{cases} pSum[r] & \text{if } l==0 \\ pSum[r] - pSum[l-1] & \text{otherwise} \end{cases}$

getSum(0,2) = pSum[2] = 13
getSum(1,3) = pSum[3] - pSum[0] = 22 - 2 = 20
getSum(2,6) = pSum[6] - pSum[1] = 37 - 10 = 27

preprocessing

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

```
int getSum(int l, int r)
{ if (l==0)          // querying in O(1) time
    return ps[r];
  return ps[r] - ps[l-1];
```

arr = [2,8,3,9,6,5,4]

ps[0] = arr[0]

ps[1] = ps[0] + arr[1]
= 2+8
= 10

ps[6] = ps[5] + arr[6]
= 33+4
= 37

getSum(2,6) = ps[6] - ps[1]
= 37 - 10
= 27

Exercise problems:

① How to answer the below weighted sum queries efficiently.

arr = [2,3,5,4,6,1]

getwSum(0,2) = 23
[1*arr[0] + 2*arr[1] + 3*arr[2]]

getwSum(2,3) = 13
[1*arr[2] + 2*arr[3]]

getwSum(l,r) = 1*arr[l] + 2*arr[l+1] ---- [r-l+1]*arr[r]

$$= \sum_{i=l}^{r} (i-l+1) * arr[i]$$

getwSum(l,r) = $\sum_{i=l}^{r} (i-l+1) * arr[i]$

$$= \sum_{i=l}^{r} i * arr[i] - (l-1) \sum_{i=l}^{r} arr[i]$$

psum[i]

we need to build two prefix sum arrays

pwSum[i] = i*arr[i] + pwSum[i-1]

Equilibrium point:

I/P: arr = [3,4,8,-9,20,6]

O/P: True

I/P: arr = [4,2,-2]

O/P: True

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

O/P: False

Naive solution: T.C = $O(n^2)$, A.S = O(1)

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

arr[] = [3,4,8,-9,9,7]

i=0 : ls=0, rs=21

i=1 : ls=3, rs=15

i=2 : ls=7, rs=7

return true

Basic Idea for the Effecient Solution:

arr = [3, 4, 8, -9, 9, 7]

① Compute prefix sum

PS = [3, 7, 15, 6, 15, 22]

② Compute suffix sum

SS = [22, 19, 15, 7, 16, 7]

③ for every element (except corner ones), check if PS[i-1] is same as SS[i+1]

Effecient Solution: TC = O(n), AS = O(1)

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

arr = [3, 4, 8, -9, 9, 7]

rs = 22, ls = 0

i=0: ls = 0, rs = 19

i=1: ls = 3, rs = 15

i=2: ls = 7, rs = 7

return true

Problem:

Given an array, check if it can be partitioned into three different parts with equal sum.

I/P: arr = [5, 2, 6, 1, 1, 1, 4]

O/P: True

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

O/P: False.

Maximum Appearing Element in Ranges:

$0 \le left[i] \le right[i] < 100$

$0 \le i < 10^5$

I/P: left = [1, 2, 5, 15]
right = [5, 8, 7, 18]
O/P : 5

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

I/P : left = [1, 2]
right = [5, 4]
O/P : 2

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

Naive Solution:

```
int maxAppear(int left[], int right[], int n)
{ int freq[100] = {0};
  for (int i=0; i<n; i++)
    for (int j=left[i]; j<=right[i]; j++)
      freq[j] += 1;
  int res = 0;
  for (int i=1; i<100; i++)
    if (freq[i] > freq[res])
      res = i;
  return res;
}
```

left[] = [1, 2, 4]
right[] = [4, 5, 7]
freq[] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ... 0]

After processing all Ranges:
freq[] = [0, 1, 2, 2, 3, 2, 1, 1, 0 ... 0]

T.C = O(n * max)
where Max = 100

Idea for efficient solution:

left[] = [1, 2, 4]
right[] = [4, 5, 7]
freq[] = [0, 0, 0, ... 0, 0]

① Traverse through all ranges and mark beginning and ending in freq[].

freq[left[i]]++ , freq[right[i]+1]--

i=0 : freq[] = {0, 1, 0, 0, 0, 0, -1, 0, 0, 0 ... 0}
i=1 : freq[] = [0, 1, 1, 0, 0, -1, -1, 0, 0, 0, ... 0]
i=2 : freq[] = [0, 1, 1, 0, 1, -1, -1, 0, -1, 0 ... 0]

② Do prefix sum on freq[]

freq[] = [0, 1, 2, 2, 3, 2, 1, 1, 0, 0, 0 ... 0]

③ Return index of the maximum value: return 4.

```
int maxAppear(int left[], int right[], int n)
{ int freq[101] = {0};
  for (int i=0; i<n; i++)
  { freq[left[i]]++;
    freq[right[i]+1]--;
  }
  int res=0;
  for (int i=1; i<100; i++)
  { freq[i] = freq[i-1] + freq[i];
    if (freq[i] > freq[res])
      res = i;
  }
  return res;
}
```

T.C = θ(n + max)
where max = 100
A.S = O(max)

left[] = [1, 2, 4], right = [4, 5, 7]
freq[] = [0, 0, 0, 0 ... 0, 0]
After 1st loop:
freq[] = [0, 1, 1, 0, 1, -1, -1, 0, -1, 0]
After 2nd loop:
freq[] = [0, 1, 2, 2, 3, 2, 1, 1, 0, ... 0]
res = 4

Sum of odd length / even length subarrays

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

⇒ [1], [4], [2], [5], [3], [1,4,2], [4,2,5], [2,5,3], [1,4,2,5,3]

1 + 4 + 2 + 5 + 3 + 7 + 11 + 10 + 15

Ans = 58

Naive approach:

Consider element included in all odd length subarrays

[1], [1,4,2], [1,4,2,5,3] → 1 + 7 + 15 = 23

[4] [4,2,5] → 4 + 11 = 15

[2] [2,5,3] → 2 + 10 = 12

[5] → 5

[3] → 3

23 + 15 + 12 + 5 + 3 = 58

T.C = O(n²)

A.S = O(1)

```
sum=0
for(i=0; i<n; i++)
{   sum=0
    for(j=i; j<n; j++)
    {   sum += a[j];
        if((i+j)%2==0)
            res += sum;
    }
}
return res
```

i=0:
j=0: sum=1, res=1
j=1: sum=5
j=2: sum=7, res=7
j=3: sum=12
j=4: sum=15, res=15

i=1
j=1: sum=4, res=19
j=2: sum=6
j=3: sum=11 res=30
j=4: sum=14

i=2;
j=2 sum=2 res=32
j=3 sum=7
j=4 sum=10 res=42

i=3
j=3: sum=5 res=47
j=4 sum=

Efficient approach:

T.C = O(n)

A.S = O(1)

length	even	odd	Total
[1] ← 1	0	1	1
[1,4] ← 2	[1,4] → 1	1	2
[1,4,2] 3	1	2	3
[1,4,2,5] 4	2	2	4

$\frac{1}{2} = 0 \left(\frac{n+1}{2}\right) = 1$

even = total/2

odd = (total+1)/2

T → left * right

→ (i+1) * (n-i)

odd: Sum = Sum + ((i+1)*(n-i)+1)/2 * a[i]

even: Sum = Sum + (i+1)*(n-i)/2 * a[i]

[1, 4, 2, 5, 3] (indices 0 1 2 3 4)

left subarrays: [2], [2,4], [2,4,1] → (i+1) subarrays

right: [2] [2,5] [2,5,3] included in this request → (n-i) subarrays

Total subarrays: (i+1)*(n-i)

n-i-1+1 = n-i

```
Sum=0
for(i=0; i<n; i++)
{   Sum += ((i+1)*(n-i)+1)/2 * a[i]
```