

# Array

- Storing variables of same data-type
- elements are stored at contiguous location

10	20	30	35	45
$x$	$x+y$	$x+2y$		

$x \rightarrow$  base address  
 $y \rightarrow$  size of variable

## Advantages

- Random access
- Cache Friendliness
  - ↳ memory closest to CPU (ideal we want all element in cache)
- when we access an element from memory to cache, pre processors generally fetch the elements near to it. In array it gives the advantage as nearby elements are fetched prior.

## Categories of Array (on basis of size)

### (a) fixed size array

- Size can be changed once declared.

Ex:-

```
int arr[100]
int arr[n]
int *arr = new int[n]
int arr[] = {10, 20, 30}
```

### \* Allocated memory in 2 ways

- ① Area in stack segment
- ② Area in heap segment (dynamically allocated memory)

Ex → `new int[n]`

- All others are stack alloc.

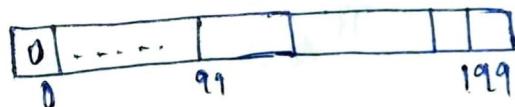
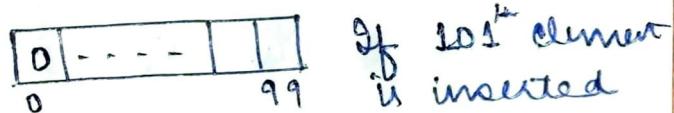
- ★ In Java all the array types (fixed) are heap allocated

### (b) dynamic size array

- Resize themselves automatically

Ex → C++ → vector  
Java → ArrayList  
Python → List

### Example implementation



- If 201<sup>st</sup> element is inserted



Again doubles up the size

## Operations in Array

### (a) Searching an Element (linear)

I/P  $\rightarrow \{10, 20, 30\}$  S  $\rightarrow 20$

O/P  $\rightarrow 1$  (index)

I/P  $\rightarrow \{20, 5, 7, 30\}$  S  $\rightarrow 8$

O/P  $\rightarrow -1$  (not present)

Time complexity -  $O(n)$

### (b) Insert an Array

I/P : arr[] = {5, 7, 10, 20, ...}.

x = 3

p = 2

O/P = {5, 2, 7, 10, 20}.

i = 4 i  $\geq$  2 i --

arr[4] = arr[3] {5, 7, 10, 20, -}  $\downarrow$

arr[3] = arr[2] {5, 7, 10, 20, 20}  $\downarrow$

arr[2] = arr[1] {5, 7, 10, 10, 20}  $\downarrow$

{ } arr[2-1] = arr[1] = 3  $\downarrow$

{5, 3, 7, 10, 20}

$O(n)$

### Insert at end in dynamic size arrays

Initial capacity

			-	-	-
0	1				n-1

Time complexity of insertion

at the end  $\rightarrow O(1)$  [like fixed size arrays]

until array has capacity

## Algorithm

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

If array gets full

(a) copy elements from previous array to new array of double size (ex.)

(b) insert at end.

Average of  $(n+1)$  inserts  $\rightarrow \frac{O(1)+O(1)+O(1)+\dots+O(n)}{n+1}$

$\rightarrow \frac{O(n)+O(n)}{n+1} \rightarrow O(1)$ .

## Time complexity

for searching,  
we traverse the  
left part of array  
2 for deletion  
we traverse the  
rest right part

[10, 20, 30, 40, 50]  
search for 30  
deletion of 30

Hence we traverse  
the whole array  
exactly one

time comp -  $O(n)$

I/P  $\rightarrow \{10, 20, 40, 50\}$

x = 20

O/P  $\rightarrow \{10, 40, 50\}$

return (3)

## Deletion of Array elements

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

## int delete (int arr[], int n, int x)

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

## Check if sorted array is sorted

• sorting is check for non-decreasing order i.e elements may be equal.

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

I/P → {10, 20, 5}  
O/P → false  
10, 20, 30  
10, 20, 15  
10, 20, 5  
Time comp. → O(n)

## Reverse An array

```
void reverse( int arr[], int low, int high) {
    low =
```

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

I/P → 10 5 7 30

↑  
low (0)  
high (3)

30 5 7 10  
↑ ↑  
low (1) high (2)

O/P → 30 7 5 10 .

I/P → {10, 20, 30, 50, 70}  
O/P → {70, 50, 30, 20, 10}

swap  
extreme  
values.

Time complexity  
→ O(n) loop  
run n/2 times  
Auxiliary space  
→ O(1)

## Remove duplicate from a sorted Array

I/P → arr[] = {10, 20, 20, 30, 30, 30} .

O/P → {10, 20, 30, -, -, -, -} .

Auxiliary Space = O(1)

• we have to return the new size.

int removesup( int arr[], int n) {

int last = n-1; int d = arr[0];

for (int i=0; i<n; i++) {

if (arr[i] == d) {  
d = arr[i];

swap (arr[i] → arr[last])

last--;

I/P → {10, 10, 20, 20, 20, 30, 30, 30, 30, 30} .

last = 8      i = 1

d = 10

## Naive Solution

• Create a copy of array and add only distinct elements to it

int removesups( 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; }

## Auxiliary space $O(1)$

```

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

```

$arr[] = \{10, 20, 20, 30, 30, 30\}$

$res = 1$

- $i=1: arr[] = \{10, 20, 20, 30, 30, 30\} res=2 *$
- $i=2: arr[] = \{10, 20, 30, 30, 30, 30\} res=2.$
- $i=3: arr[] = \{10, 20, 30, 30, 30, 30\} res=3 *$
- $i=4: arr[] = \{10, 20, 30, 30, 30, 30\} (no change)$
- $i=5: arr[] = \{10, 20, 30, 30, 30, 30\} (no change).$

O/P  $\rightarrow \{10, 20, 30, 30, 30, 30\}$  return 3.

## Left Rotate an array by one

I/P  $\rightarrow \{1, 2, 3, 4, 5\}$

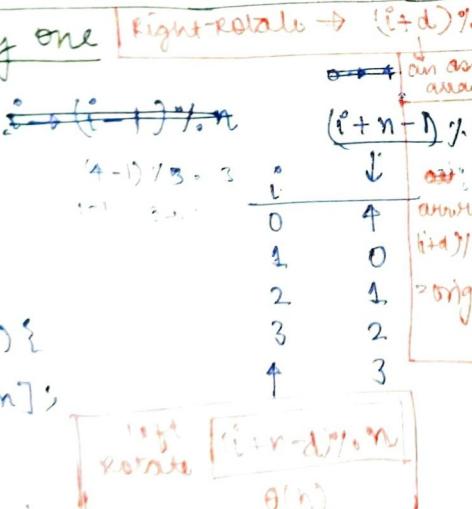
O/P  $\rightarrow \{2, 3, 4, 5, 1\}$

① By shifting array while taking input

```

for (int i=0; i<n; i++) {
    cin >> arr[(i+n-1)%n];
}

```



② By using different array:

```

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

```

## DR

```

int leftShift (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

### ① Naive Solution

Rotate array by 1  $\rightarrow d$  times  
time complexity  $\rightarrow O(nd)$   
auxiliary space  $\rightarrow O(1)$

### ② Time complexity reduced by to $O(n)$

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=d; i<n; i++) {
    arr[i-d] = temp[i];
    arr[n-d+i] = temp[i];
}

```

time comp  $\rightarrow O(n)$   
auxiliary  $\rightarrow O(n)$

### ③ Most Effective Solution

void leftRotate (int arr[], int n, int d) {

```

reverse (arr, 0, d-1);
reverse (arr, d, n-1);
reverse (arr, 0, n-1);

```

For right rotate  
 $rev(arr, 0, d-1) \rightarrow (a, b, ..., d-1)$   
 $rev(arr, d, n-1) \rightarrow (a, b, ..., d-1, n-d)$   
 $rev(arr, 0, n-1) \rightarrow (a, b, ..., n-1)$

```

void reverse (int arr[], int low, int high) {
    while (low < high) {
        swap (arr[low], arr[high]);
        low++;
        high--;
    }
}

```

$arr[8] = \{1, 2, 3, 4, 5\}$

$d=2$

↓ (I)

$\{2, 1, 3, 4, 5\}$

↓ (II)

$\{2, 1, 5, 4, 3\}$

↓ (III)

Rotated arr →  $\{3, 4, 5, 1, 2\}$

### Leader in an Array

I/P →  $arr[] \rightarrow \{7, 10, 4, 3, 6, 5, 12\}$

~~leaders~~ an element is a leader if there is no element greater than that is present in the array in right

O/P → 10, 6, 5, 12 (Rightmost element is always a leader (last el.))

\* If array is sorted in increasing order only last element is leader

\* If array is sorted in decreasing order every elem. is leader.

\* Leader must be strictly greater than all the elements on right, if there is an element eq. to that, then that el. is not a leader. (Equals are not allowed).

### Algo

```
int leader (int arr[], int n) {
    for (int i=0; i<n; i++) {
        flag = 1;
        for (int j=i+1; j<n; j++) {
            if (arr[j] ≥ arr[i]) {
                flag = 0;
            }
        }
        if (flag == 1) print (arr[i]);
    }
}
```

$\Theta(d + (n-d) + n)$

$\Theta(2n) \rightarrow \Theta(n)$

auxiliary sp →  $\Theta(1)$

### Efficient Solution :-

I/P =  $arr[] = \{7, 10, 4, 3, 6, 5, 12\}$

O/P = 2, 5, 6, 10

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

last element  
is always a  
leader

$\Theta(n)$

Drawback → print leader from right to left  
→ To clear this we have to maintain an array or vector which stores the leaders which eventually increases the space complexity as  $\Theta(n)$

### Maximum Difference

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

I/P →  $\{2, 3, 10, 6, 4, 8, 1, 5\}$

O/P → 8 (10-2)

I/P →  $\{7, 9, 5, 6, 3, 2\}$

O/P → 2

### Naive Solution

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

$\Theta(n^2)$

\* If array is sorted

diff = last el - first el

\* If arr. is reverse sorted

diff = max not depends

## Efficient solution

- keep a track of minimum value at the left and subtract it from the elements on right

\* If

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

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

$$res = 1 \quad min = 2$$

$$\boxed{i=1} \rightarrow res = 1 \quad min = 2$$

$$\boxed{i=2} \rightarrow res = 8 \quad min = 2$$

$$\boxed{i=3} \rightarrow res = 8 \quad min = 2$$

$$\boxed{i=4} \rightarrow res = 8 \quad min = 2$$

$$\boxed{i=5} \rightarrow res = 8 \quad min = 2$$

$$\boxed{i=6} \rightarrow res = 8 \quad min = 1.$$

Time comp.  
 $\Theta(n)$

Auxiliary space  
 $\Theta(1)$

## Frequency of Elements in Sorted Array

```
void freq (int arr[], int n) {
    int freq = 1, i=1;
    while (i < n) {
        while ((i < n) && (arr[i-1]
```

$$I/P \rightarrow \{10, 10, 20, 20, 20\}$$

$$O/P \rightarrow 2, 3$$

void frequency (int arr[], int n) {

    int freq = 1, i = 1;

    while (i < n) {

        while (i < n && arr[i-1] == arr[i]) {

            freq++;

            i++;

        cout << "frequency for the element" << freq;

        freq = 1;

        i++;

    }

$$arr[] = \{ \underbrace{10, 10}_{\textcircled{2}}, \underbrace{20, 30}_{\textcircled{1}}, \underbrace{30, 30}_{\textcircled{3}} \}$$

i = 1

freq = 2.

Stock Buy and Sell

$$I/P \rightarrow \{1, 5, 5, 8, 12\}$$

B↑ ↑S ↑B ↑S

O/P  $\rightarrow 4+$

Stock Buy and Sell

$$I/P \rightarrow \{1, 5, 3, 8, 12\}$$

B↑ S ↓ B↑ ↑S

$$O/P \rightarrow (5-1) + (12-3) \rightarrow 9+4 = 13$$

$$I/P \rightarrow \{30, 20, 10\}$$

O/P  $\rightarrow 0$  (if prices are reverse sorted, will will have a profit of 0)

$$I/P \rightarrow \{10, 20, 30\}$$

O/P  $\rightarrow 30-10 = 20$  (if prices are sorted, then profit will be last item - first)

$$I/P \rightarrow \{1, 5, 3, 1, 2, 8\}$$

O/P  $\rightarrow B↑ ↑S B↑ ↑S$

$$O/P \rightarrow 8(5-1) + (8-1) \rightarrow 11$$

## Approach - 1

```

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

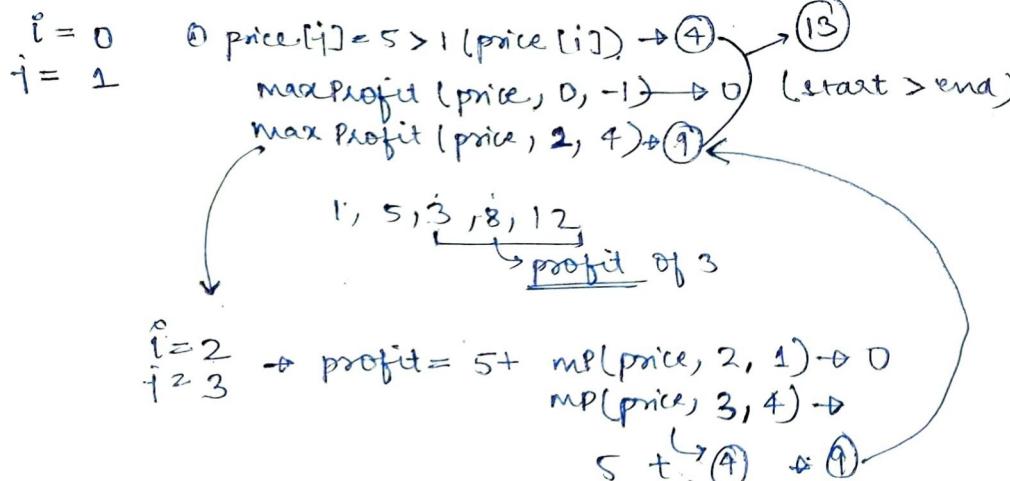
```

## Flow

arr[]  $\rightarrow \{1, 5, 3, 8, 12\}$

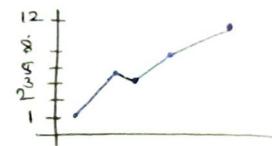
start  $\rightarrow 0$

end  $\rightarrow 4$



## Approach - 2 (Efficient solution) {1, 5, 3, 8, 12}

- \* buy the stock at the bottom and sell it at top of top
- \* when graph is going up add the difference between the points and when its decreasing do nothing



```

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

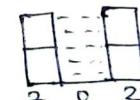
$$\begin{aligned}
 i = 0 & \quad p[1] - p[0] = 5 - 1 = 4 > 0 \quad profit = 4 \\
 i = 1 & \quad p[2] - p[1] = -2 < 0 \quad p = 4 \\
 i = 2 & \quad p[3] - p[2] = 8 - 3 = 5 > 0 \quad p = 4 + 5 \\
 i = 3 & \quad p[4] - p[3] = 12 - 8 = 4 > 0 \quad p = 9 + 4 = 13
 \end{aligned}$$

## Trapping Rain Water

I/P  $\rightarrow$  arr[] = {2, 0, 2}

(3 bars of different heights)

O/P  $\rightarrow$  2x1



O/P  $\rightarrow$  arr[] = {3, 0, 1, 2, 5}

O/P  $\rightarrow$

$$3 + 2 + 1 = 6$$



I/P  $\rightarrow$  arr[] = {1, 2, 3}



array in increasing or decreasing order  
the amount of water stored = 0

## Naive Solution

Check the maximum left bar and check the maximum right bar and take the minimum of them and subtract it from the height of bar



for  $i = 2$

initialise lmax and rmax with height of bar

$lmax = 2 \rightarrow$  on traversing left we found base of height greater than  $lmax = 3$   
 $rmax = 2$   
 $\downarrow$   
 on traversing right we found  $arr[i] = 5$   
 hence  $rmax = 5$

Now for  $i = 2$   $\min(lmax, rmax) -$

$$\begin{aligned} &= \text{height of base} \\ &\min(5, 3) - arr[2] \\ &= 3 - 2 = ① \end{aligned}$$

int getWater(int arr[], int n){

int res = 0;

int lmax = 0, rmax;

for (int i = 1; i < n - 1; i++) {

lmax = arr[i];

for (int j = 0; j < lmax; j++) {

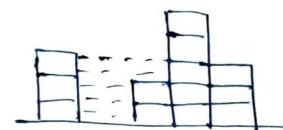
if (arr[j] > lmax) lmax = arr[j]; }

rmax = arr[i];

for (int k = i + 1; k < n; k++) {

if (arr[k] > rmax) rmax = arr[k]; }

I/P  $\rightarrow \{3, 0, 2, 5, 3\}$



start from  
 $i = 1 \neq end$   
 $\rightarrow n - 2$ . Bcz  
 extreme ends  
 can't store  
 anything

I/P  $\rightarrow \{5, 7, 9, 10, 3\}$



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

return res;

Time comp -  $\Theta(n^2)$

## Effective Solution

precompute all the lmax & rmax for an element.

int getWater(int arr[], int n){

int res = 0

int lmax[n], rmax[n]; // initialising arrays for storing lmax & rmax for current element

int lmax[0] = arr[0]

for (int i = 0; i < n; i++) {

lmax[i] = max(arr[i], lmax[i - 1]);

rmax[n - i] = arr[n - i];

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;

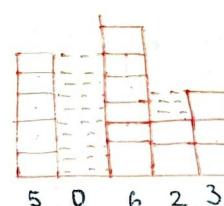
$i = 2, 3$

lmax  $\rightarrow \{5, 5, 6, 6, 6\}$

rmax  $\rightarrow \{6, 6, 6, 6, 5\}$

res  $\rightarrow \{5, 0, 2, 1, 5\}$

$$= ⑦ 5 + 1 \Rightarrow ⑥$$



Time comp  $\rightarrow \Theta(n)$

Auxiliary space  $\rightarrow \Theta(n)$

## Maximum consecutive 1s in Binary Array

```
int maxConsecutiveOnes (boolean arr[])
```

- maintain a current count
  - if arr[i] = 1 curr++
  - if arr[i] = 0 maxl  
res, curr);

```
int maxConsecutiveOnes (boolean arr[], int n){
```

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

3 res = max(res, curr); return res; 3

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

res = 2

curr = 0 ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ 3

## Maximum Sub Array

I/P : arr [] → {2, 3, -8, 7, -1, 2, 3}

O/P → 11

I/P : arr [] → {5, 8, 3} // all elements are +ve O/P  
sum of all elements

O/P → 16

I/P : arr [] → {-6, 1, -8} // all elements negative O/P → maximum element

O/P → -1

## Naive Solution

Find sum of all the subarrays

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

$O(n^2)$

{1, -2, 3, 4, 2, 3}

i = 0:

curr = 0

j = 0	curr = 1	res = 1
j = 1	curr = -1	res = 1
j = 2	curr = 2	res = 2
j = 3	curr = 1	res = 2
j = 4	curr = 3	res = 3

i = 1:

curr = 0

j = 1	curr = -2	res = 3
j = 2	curr = 1	res = 3
j = 3	curr = 0	res = 3
j = 4	curr = 2	res = 3

i = 2:

curr = 0

j = 2	curr = 3	res = 3
j = 3	curr = 2	res = 3
j = 4	curr = 4	res = 4

i = 3:

curr = 0

j = 3	curr = -1	res = 4
j = 4	curr = 1	res = 4

i = 4:

curr = 0

j = 4	curr = 2	res = 4
-------	----------	---------

Effective solution

{ -5, +1, -2, 3, +1, 2, -2 }

for each element calculate the maximum of sum  
all the subsets ending with that element

{ -5, 1, -2, 3, +1, 2, -2 }  
↓      ↓      ↓      ↓      ↓  
①      ②      ③      ④      ⑤  
-4      -1      -1      2      -3

max → maximum of previous + arr[i] or arr[i].

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

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

O(n)

(Kadane's Algorithm)

int maxsum ( int arr[], int sum) {

int res = 0;

int maxEnding = arr[0];

for (int i=1; i<n; i++) {

maxEnding = max ( maxEnding + arr[i],  
 arr[i]);

res = max (Ending, res);

}

return res;

O(n)

O(1)

Maximum length of Even-Odd subarray.

I/P → { 10, 12, 14, 7, 8 }

O/P → 3

I/P → { 10, 12, 8, 4, 6 }

O/P → 1 (because whole array is even)

O(n<sup>2</sup>) → loop through all elements and check all the subarrays starting from that element

O(n) → check all the subarrays where the last element is arr[i]

int longestSub ( int arr[], int n) {

int curr = 1;  
int res = 1;

for (int i=1; i<n; i++) {

if ((arr[i] & arr[i-1]) & 1) {

curr++;

res = max ( res, curr );

else {

curr = 1;

return res;

$\text{arr}[] \rightarrow \{5, 10, 20, 6, 3, 8\}$

$i = 0$	$\text{res} = 1$
	$\text{curr} = 1$
$i = 1$	* $\text{curr} = 2$ $\text{res} = 2$
$i = 2$	$\text{curr} = 1$
$i = 3$	$\text{curr} = 1$
$i = 4$	$\text{curr} = 2$ $\text{res} = 2$
$i = 5$	$\text{curr} = 3$ $\text{res} = 3$

(3)

### Maximum subarray sum

$\{10, 5, -5\}$

↳ normal subarray  $\rightarrow \{10\}, \{5\}, \{-5\}, \{10, 5\}, \{10, 5, -5\}$   
 $\{5, -5\}$

↳ circular subarray  $\rightarrow \{-5, 10, 5\}, \{-5, 10\}, \{5, -5, 10\}$

I/P  $\rightarrow \{5, -2, 3, 4\}$

I/P  $\rightarrow \{-3, 4, 6, -2\}$

O/P  $\rightarrow 12$

O/P  $\rightarrow 10$

I/P  $\rightarrow \{2, 3, -4\}$

I/P  $\rightarrow \{-8, 7, 16\}$

O/P  $\rightarrow 5$

O/P  $\rightarrow 13$

I/P  $\rightarrow \{8, -4, 13, -5, 4\}$

I/P  $\rightarrow \{3, -4, 5, 6, -8, 7\}$

O/P  $\rightarrow 12$

O/P  $\rightarrow 17$

• consider every element as starting of subarray  
and calculate all the sum of all the possible  
subarrays.

$\text{arr}[] \rightarrow \{5, -2, 3, 4\}$

$i = 0$	$\text{curr\_max} = 5$
	$\text{curr\_sum} = 5$
$j = 1$	$\text{curr\_max} \rightarrow 5 + (-2) = 3$ $\text{curr\_max} \rightarrow 5$

$i = 2$	$\text{curr\_sum} = 3$ $\text{curr\_max} = 3$
$j = 1$	$\text{curr\_sum} \rightarrow 3 + 4 = 7$ $\text{curr\_max} \rightarrow 7$
$i = 2$	$(i+j) \% n \rightarrow (2+1) \% 4 \rightarrow 0$

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 + initialised.  
j = 1 because  
we can multiply the  
4 put here  
            curr\_sum = curr\_sum + arr[index];  
            curr\_max = max (curr\_max,  
                         curr\_sum);

        res = max (res, curr\_max);

    return res;

### Maximum circular subarray (Effective solution)

Idea. ① Maximum sum of normal subarray Kadane's algorithm.  
② Maximum sum of only circular subarray

Take max.  
of these two

+ maximum circular sum will be subtracting  
minimum subarray sum from whole array sum.

modified Kadane's  
Algorithm.

OR

just do the  
minimum  
of last sum + arr[i]  
+ ~~last arr[i]~~

invert arr[i]  $\rightarrow -arr[i]$   
and the just  
invert the result  
also -sum

## Algorithm

```

int Findmax (int arr[], int n){ ← Kadane
    int EndingMax = arr[0];
    for (int i=1; i<n; i++) {
        res = max (res, max);
        maxEnding = max (maxEnding + arr[i], arr[i]);
        res = max (res, maxEnding);
    }
    return res;
}

```

3. int overallMax (int arr[], int n){

```

int max-normal = Findmax (arr, n);
if (max-normal < 0) { } ← All elements are -ve, hence max sum will be the max element itself.
    return max-normal;
}

```

```

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

```

4. max\_circular = Findmax (arr, n) + sum;  
 res = max (max-normal, max\_circular);  
 return res;

5.

max\_circular = sum + Findmax (arr, n)

{8, -4, 3, -5, 12}

inverted

-8, 4, -3, 5, -4  
 ↪ -4      ↪ -2      ↪ 6      arr sum = 6

max\_circular → 6 + 6 = 12  
 max (8, 12) → 12

↓  
 Between the negative of maximum sum.  
 ↪ If there are two or more such numbers, it will directly add it to find max\_circular.

## Majority Element

- At Element which appears  $\lceil n/2 \rceil$  times in a array
- I/P → {8, 3, 4, 8, 8}
- O/P → 0 or 3 or 4 (Return any of index.)
- I/P → {3, 7, 4, 7, 7, 5}
- O/P → -1 (No majority) (7 appears only 3 times <  $\lceil 3.5 \rceil$ )

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; ← loop breaks when
    } ← fun. returns when
    return -1; ← if no element
} ← is found.

```

## Effective Solution

int findMajority (int arr[], int n){

```

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

```

et/contd.

```

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

```

I/P → {8, 8, 6, 6, 4, 6, 5}

- Main concept of this approach is that if an element appears more than  $n/2$  times the count++ for that will be more than count-- and hence at last that will be the result.

$i=0$	$res=0$	$count=1$
$i=1$		$count=2$
$i=2$		$count=1$
$i=3$		$count=0$
	<u><math>res=3</math></u>	<u><math>count=1</math></u>
$i=4$		$count=2$
$i=5$		$count=1$
$i=6$		$count=2$

comparing every element with  $arr[res] \rightarrow arr[3]$

$$6 \rightarrow 4 > [7/2]$$

O/P → 6

Time Complexity →  $O(n)$

I/P → {6, 8, 4, 8, 8}  
 count → 1 res = 0  
 $i=1 \rightarrow$  count → 0  
 $res = 1$  count = 1  
 $i=2$  count = 0  
 $res = 2$  count = 1  
 $i=3$  count = 0  
 $res = 3$  count = 1  
 $i=4$  count = 1

## Minimum 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 → 2 to 3, 5 to 6

I/P → arr[] = {1, 1, 1}

O/P → -

I/P → arr[] = {0, 1}

from 0 to 0 (OR)

from 1 to 1

### Naive Solution

Traverse the array from left to right, as and maintain the count of no of groups of 0's and 1's. Then decide which group it want to flip and then again traverse the array & print the seg. statements.

### Effective Solution

The difference between two group is always going to be 1 or 0.

Suppose we start from 1 and end at one more with no zeroes between the no of zero gap → 0 diff → 1

1<sup>st</sup> case      111 --- (diff = 2)

2<sup>nd</sup> case      1110000 (diff = 0)

3<sup>rd</sup> case      1110001 (diff = 1)

As As the difference is always gonna 1 or 0 so we make a rule to have a flip on second group. (Three grp are less obviously).

## Approach

```

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;
        }
    }
}

```

Check if element is different from previous element  
 (we start by  $i=1$  because 1<sup>st</sup> element will always be the member of 1<sup>st</sup> group)

To check whether it is the starting of group we want to print or ending of group we compare it with 1<sup>st</sup> element  
~~if 1<sup>st</sup> elem if its diff then it is starting~~

similarly here if element is same then it is the starting of groups we don't require

I/P  $\rightarrow$  0 0 1 1 0 0 1 1 0

$\Rightarrow$   $i=1$   $arr[i] == arr[i-1]$   
 $i=2$   $arr[2] \neq arr[1]$

$\hookrightarrow$  end in condition

to check whether it is starting of 2<sup>nd</sup> group so ending we comp it with  $arr[0]$

To have starting element  $\neq arr[0]$

From 1 to

$\xrightarrow{x}$   
 $arr[i] \neq arr[i-1]$

$\& arr[i] == arr[0]$

2 (newline)

$i=3$

$i=4$

Output

From 2 to 3

From 6 to 7

## Window Sliding Technique

Given an array we need to find the sum of 'k' consecutive elements & print the maximum of them

I/P  $\rightarrow$  {1, 8, 30, -5, 20, 7} | I/P  $\rightarrow$  {5, -10, 6, 90, 13}.  
 $k=3$  |  $k=2$   
 $\sum_{i=0}^{i=3} 39, 33, 45, 23$  |  $\sum_{i=0}^{i=1} 5, -10, 6$   
O/P  $\rightarrow$  45 | O/P  $\rightarrow$  96

### Naive Approach

Run a loop till  $n$  ( $i+k-1 < n$ ) and then again run a loop from  $i$  to  $k$  and calculate the max sum.

```

int maxconsecutiveSum (int arr[], int n){
    int max-sum = INT-MIN; ← used res inside code
    for (int i=0; i+k-1 < n; i++) {
        int sum=0;
        for (int j=i & j < i+k; j++) {
            sum += arr[j];
            res = max (res, sum);
        }
    }
    return max-sum;
}

```

### Time complexity

$O((n-k) \times k) \sim O(n^2)$

### Effective Approach (Window Sliding Technique) [O(n)]

- compute the sum of 1<sup>st</sup> window (1<sup>st</sup>  $k$  elements)
- and then shift the sum only when it is greater in sum than previous one

```
int maxsum (int arr[], int n){
```

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

```
res = curr-sum;
for (int i=1; i < n-k+1; i++) {
    curr-sum += arr[i-k+1] - arr[i-1];
}
```

```
res = max (res, curr-sum);
```

```
return res;
```

90

Q) Window Sliding Technique to print a subarray whose sum is given

I/P  $\rightarrow \{1, 4, 20, 3, 10, 5\}$  Sum = 33

O/P  $\rightarrow \{20, 3, 10\}$  true (20, 30, 3).

We start by 1<sup>st</sup> element and add more elements till the sum of current subarray is less than the required sum. If by adding an element the sum exceeds, then we remove the first element from left and remove it till the curr\_sum is greater than the required sum.

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

T.C  $\rightarrow O(n)$   
A.S  $\rightarrow O(1)$

• only applies on non-negative elements

I/P: N=3 M=8  
3-bonacci  
No of elements to be printed

• first  $n-1$  elements is going to be zero & next element is always  $\neq 1$

O/P  $\rightarrow 0, 0, 1, 1, 2, 4, 7, 13, \dots$

Ques count distinct elements in every window of size k.

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

O/P : 3, 4, 3, 3

3 distinct elements in window 1. 4 distinct elements in window 2.

## Exercise

② N-bonacci Numbers

$\uparrow$   
fibonacci  $\rightarrow$  2-bonacci (every element is sum of previous 2)

3-bonacci  $\rightarrow$  every element is sum of previous 3.

N-bonacci  $\rightarrow$  every element is sum of previous n.

## Prefix Sum

Given a fixed array & multiple queries of following types on the array, how to efficiently perform the queries →  $\text{getSum}(0, 2)$ ,  $\text{getSum}(0, 3)$ ,  $\text{getSum}(2, 6)$ .

$$\text{arr}[] \rightarrow \{2, 8, 3, 9, 6, 5, 4\}$$

$$\text{getSum}(0, 2) \rightarrow 2 + 8 + 3 \\ \rightarrow 13$$

$$\text{getSum}(2, 6) \rightarrow 3 + 9 + 6 + 5 + 4 \\ \rightarrow 27$$

we have to find sum of elements between indices  $l$  and  $r$  including these indices

```
int getSum (int arr[], int l, int r) {
    if (l == 0) {
        prefix-sum[r] - prefix-sum[l-1];
    } else {
        prefix-sum[r]
    }
}
```

// Find if an array has an equilibrium point.

(a)  $\{3, 4, 8, -9, 20, 6\}$   
 $3+4+8+(-9) \rightarrow 6$   
 $= 6$

Yes

[NO]

no equilibrium pt.

(b)  $\{4, 2, -2\}$

$$0 \rightarrow 2+(-2) \\ = 0$$

Yes.

we can check if an element is equilibrium point if

sum = total sum of array  
 $P[] \rightarrow \text{prefix array}$

$$\boxed{\text{sum} - p[i^0] = p[i^0-1]}$$

sum elements before it = sum of element after it

$O(n) + O(n)$  time Auxiliary space

### $O(1)$ Auxiliary Space

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

int l-sum=0

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

~~if (sum == 0)~~

```
-if (l-sum == sum - arr[i]) {
    return true;
}
```

$l-\text{sum} = l-\text{sum} + arr[i]$

$\text{sum} = \text{sum} - arr[i]$  } return false;

To reduce the time complexity, we have to pre-compute values.

$$\text{arr}[] \rightarrow \{2, 8, 3, 9, 6, 5, 4\}$$

$$\text{pre-sum}[] \rightarrow \{2, 10, 13, 22, 28, 33, 37\}.$$

pref-sum[n];

pref-sum[0] ← arr[0]

for (int i=1; i<n; i++) {

    pref-sum[i] = pref-sum[i-1] + arr[i];

Now the calculation of prefix sum is easy.

$$\text{getSum}(l, r) \rightarrow \text{pref-sum}[r] - \text{pref-sum}[l-1]$$

$$\text{getSum}(3, 6) \rightarrow \{2, 8, 3, 9, 6, 5, 4\} \quad 37 - 13 \\ \underline{\hspace{10em}} \quad \text{sum} \quad = 24$$

•  $l=0$  (case explicitly handled).

If given ranges, we have to find max. element in these ranges.

Questions on prefix sums can be divided into three

$L[\cdot] \rightarrow \{1, 2, 3\}$

$$R[] \rightarrow \{3, 5, 7\}$$

- $R[5] \rightarrow [5, 51+]$

  - ① we need to know the upper bound of value in  $R[5]$
  - ② we keep track of starting and ending position of array

```
vector<int> arr [1000];
```

$$\frac{1}{0}, \frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \frac{1}{8}, \frac{1}{9}, \frac{1}{10}, \frac{1}{11}, \frac{1}{12}$$

$$\text{arr(pcf)} \rightarrow \frac{0}{0} \frac{1}{1} \frac{2}{2} \frac{3}{3} \frac{2}{4} \frac{2}{5} \frac{1}{6} \frac{1}{7} \frac{0}{8} \frac{0}{9} \frac{0}{10} \frac{0}{11}$$

- By calculating the prefix sum, we keep track of frequency of certain elements
  - ① if array fall in range ; it value will not decrease, if its out of range, decreased by 1
  - ② if another range starts with previous range count increments

```

int maxOcc (int arr[], 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], res = 0;
    for (int i=1; i<1000; i++) {
        arr [i] += arr [i-1];
        if (maxm < arr [i]) {maxm = arr [i], res = i}
    }
    return res;
}

```

## Searching

### Binary Search

I/P  $\rightarrow \{10, 20, 30, 40, 50, 60\}$   
 $x = 20$

O/P  $\rightarrow 1$

I/P  $\rightarrow \{5, 15, 25\}$   
 $x = 25$

O/P  $\rightarrow 1$

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

I/P  $\rightarrow 10, 20, 30, 40, 50, 60$   $x > 25$   $\begin{matrix} \text{begin} \rightarrow 0 \\ \text{end} \rightarrow 5 \end{matrix}$

$$\text{mid} = (0+5)/2 \rightarrow 2$$

$30 > 25$  ---

O/P  $\rightarrow -1$

\* I/P  $\rightarrow \{10, 10, 20\}$   
 $x = 10$   
O/P  $\rightarrow 0 \text{ or } 1$

### Binary Search (Recursive)

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

use

return bSearch (arr, mid + 1, high, x);

### Comparing Iterative and Recursive Approach

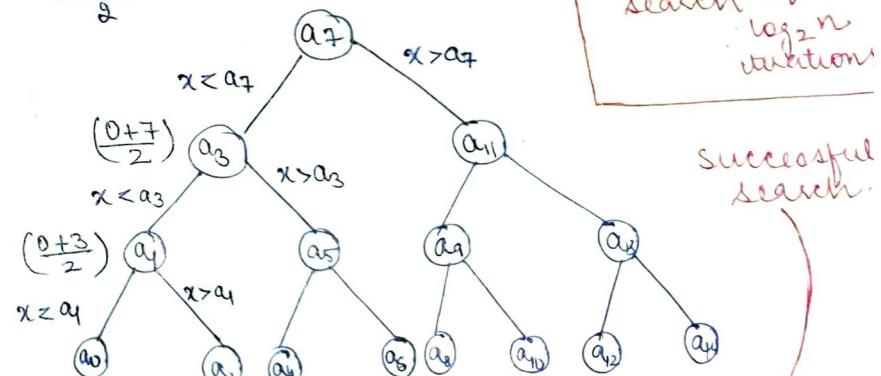
Time-complexity  $\rightarrow$  same in both ( $O(\log n)$ )

Auxiliary space  $\rightarrow$  iterative  $\rightarrow O(1)$  space  
recursive  $\rightarrow O(\log n)$   $\leftarrow$  for storing recursion call stack.

### Analysis of Binary Search

$\boxed{a_0 | a_1 | \dots | a_6 | a_7 | a_8 | \dots | a_{14} | a_{15}}$

$$\text{mid} = \frac{0+15}{2} = 7$$



The no. of iterations for searching a particular element is equal to its height from root

Height of Tree  $\rightarrow \lceil \log_2 n \rceil$

## Index Of First Occurrence

- Naive solution → Traverse through whole array and as soon as you find the element return it.

### Effective approach

- By using binary search, we normally traverse through left & right subparts until we get the element equal to arr[mid]

Here two cases cases arise

- ① mid == 0 (First occurrence confirmed)
- ② arr[mid-1] != arr[mid] (First occ.)

if (arr[mid-1] == arr[mid])  
we will call for left subtree.

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

Checking for first occurrence

Not the first occ.

y  
y  
y

## Iterative Approach

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

## Find the last occurrence of Number in Array

I/P: arr[] → {10, 15, 20, 20, 40, 40}.

x = 20

O/P: 3.

### using Binary Search

```
int lastOcc(int arr[], int begin, int end, int x){
    if (begin > end) return -1;
    if (arr[begin] == x)
        return begin;
    else
        return lastOcc(arr, begin+1, end, x);
}
```

Next page →

① mid == n-1 || arr[mid] == arr[mid+1]

## Recursive

```

int find ( int arr[], begin,
          end, x) {
    int mid = (begin+end)/2;
    if (begin > end)
        return -1;
    if (arr[mid] > x) {
        return find (arr, begin, mid-1, x);
    } else if (arr[mid] < x) {
        return find (arr, mid+1, end, x);
    } else {
        if (mid == n-1 || arr[mid] != arr[mid+1])
            return mid;
        else {
            return find (arr, mid+1, end, x);
        }
    }
}

```

## Count occurrences in a sorted Array

We can call first occurrence and last occurrence b/w m and can find the no of occurrences

```

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

```

check if element is not present

```

else
    return (lastOcc (arr, n, x) - first + 1)
}

```

## Iterative

```

int find ( int arr[], x) {
    int begin = 0;
    int end = n-1;
    while (begin <= end) {
        if (arr[mid] > x) {
            end = mid-1;
        } else if (arr[mid] < x) {
            begin = mid+1;
        } else {
            if (mid == n-1 || arr[mid] != arr[mid+1])
                return mid;
            else {
                begin = mid+1;
            }
        }
    }
    return -1;
}

```

## Count 1's in sorted Binary Array

Naive solution → find index of first 1 and then subtract it from size

$$\{0, 0, 1, 1, 1, 1, 1\} \rightarrow 6 - 2 = 4$$

Time complexity →  $O(n)$  (In worst case)

\* calculate the first occurrence using binary search  
time comp →  $O(\log n)$

## calculate the square root

\* if square root exists then return it  
otherwise take the ceil of it and return

### ① Naive solution

iterate from  $i=1$  to  $n$  until  $i*i > num$

```

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

```

$x = 9$	
$i = 1$	$1 \leq 9$ ✓
$i = 2$	$4 \leq 9$ ✓
$i = 3$	$9 \leq 9$ ✓
$i = 4$	$16 \leq 9$ ✗
	return $i-1 = 3$ .

## using binary approach

```

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

```

\* check if  $x/2$  is square root  
→ if  $x/2$  then check  $x/4$   
then  $x/8$  ...

storing temporarily  
ans & waiting for  
largest ans.

## Search in Infinite sized sorted Array

I/P  $\rightarrow \{1, 10, 15, 20, 40, 80, 90, 100, 120, 150, \dots\}$   $x = 100$

O/P  $\rightarrow ?$

I/P  $\rightarrow \{20, 40, 100, 300, \dots\}$   $x = 50$

O/P  $\rightarrow -1$

### Naive Solution

Time complexity  $\rightarrow O(\text{position})$ .

- ① If element is present then simply
- ② If el. is not present then pos  $\rightarrow$

```
int FindElement(int arr[], int x){
    for (int i=0; i < arr.length; i++){
        if (x == arr[i]){
            return i;
        } else if (x < arr[i]){
            return -1;
        }
    }
    return -1;
}
```

$O(\text{position})$

where element  
should have  
present in  
sorted array

```
while (true){
    if (arr[i]==x)
        return i;
    if (arr[i]>x)
        return -1;
    i++;
}
```

int search (int arr[], int x) {

if (arr[0] == x) return 0;

int i = 1;

while (arr[i] < x)

i = i \* 2;

if (arr[i] == x) return i;

return binarySearch(arr, x, i/2+1, i-1);

}

Time complexity  $\rightarrow O(\log(\text{pos}))$

Popularly known as  $\rightarrow$

Unbounded  
Binary Search

because  
element  
was greater  
than pos.  
 $i$ .  
↑  
element  
was smaller  
than pos.

## Search in sorted + Rotated Array

I/P  $\rightarrow \{10, 20, 30, 40, 50, 8, 9\}$

x = 30

O/P  $\rightarrow 2$

I/P  $\rightarrow \{100, 200, 300, 10, 20\}$

x = 40

O/P  $\rightarrow -1$

Array is sorted +  
we rotate it counter  
clockwise (left)  
direction.

### Effective Approach

Initially we start with  $i=1$  (explicitly handling the case  $i=0$ ). If element  $> arr[i]$  then we double the index. We keep doubling the index till we reach an index where either the element  $= arr[i]$  or element  $< arr[i]$ . By this way we get an upper bound.

```
int i ≠ 1;  
while :
```

- Here one half of the array is always sorted. If array is not rotated at all then both half are sorted. If it is rotated by 1 unit then left half is sorted. If it is sorted by  $> N/2$  elements then right half would be sorted. We check the middle element with left most element
  - if  $a[middle] > a[left]$  (left is sorted)
  - if  $a[middle] < a[left]$  (right is sorted)
- ①  $\{100, 200, 300, 10, 20\}$  ( $300 > 100$ ) left is sorted
- ②  $\{100, 500, 10, 20, 30\}$  ( $10 < 100$ ) right is sorted

• we ignore that part of array where element is not present. by using binary search.

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

left half sorted

right half sorted

Peak Element in Array → Not smaller than neighbours.

I/P → {5, 10, 20, 15, 7} ↑  
peak

I/P → {10, 20, 15, 5, 23, 9} 6 7

I/P → {80, 70, 60}

• If element is leftmost element then we need to check the right index only.

### Naive Solution

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

O(n)

### Efficient Solution

the idea is if we go to middle element, and check the left and right indices if both the smaller than return mid but if one of them is greater than arr[mid] then definitely there must be a peak on that side.

```
int getAPeak ( int arr[], int n) {  
    int low = 0, high = n-1;  
    while ( low <= high) {  
        int mid = (low + high)/2;  
  
        mid is peak element  
        if ((mid == 0 || arr[mid-1] <= arr[mid]) &&  
            (mid == n-1 || arr[mid+1] <= arr[mid]))  
            return mid;  
  
        left part contains peak  
        if (mid > 0 && arr[mid-1] >= arr[mid])  
            high = mid - 1;  
        else  
            right part contains peak.  
            low = mid + 1;  
    }  
    return -1;  
}
```

Unsorted Array, we have to find number of pairs with sum =  $x$

I/P  $\rightarrow \{3, 5, 9, 2, 8, 10, 11\}$  num = 17

O/P  $\rightarrow$  yes (9, 8)

I/P  $\rightarrow \{8, 4, 6\}$  x = 11

O/P  $\rightarrow$  NO.

\* We can solve this problem using hashing, where before putting an element into the hash table we check whether  $(x - arr[i])$  is present or not.

If we are given a sorted array, there is a better approach

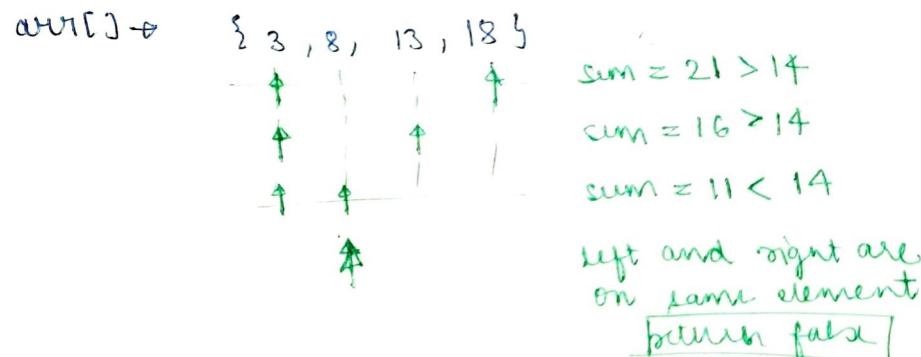
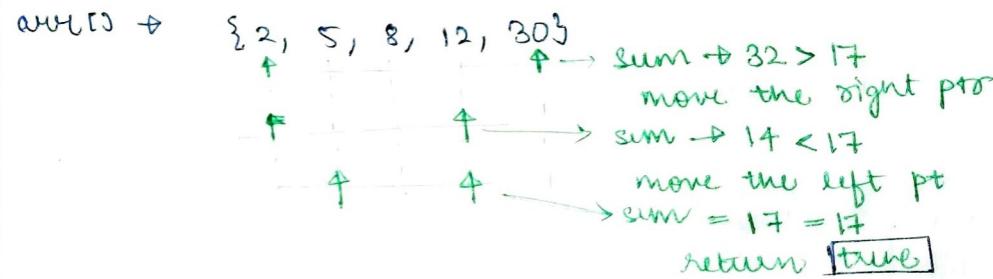
I/P  $\rightarrow \{2, 5, 8, 12, 30\}$

x = 17 O/P  $\rightarrow$  Yes (5, 12)

I/P  $\rightarrow \{3, 8, 13, 18\}$

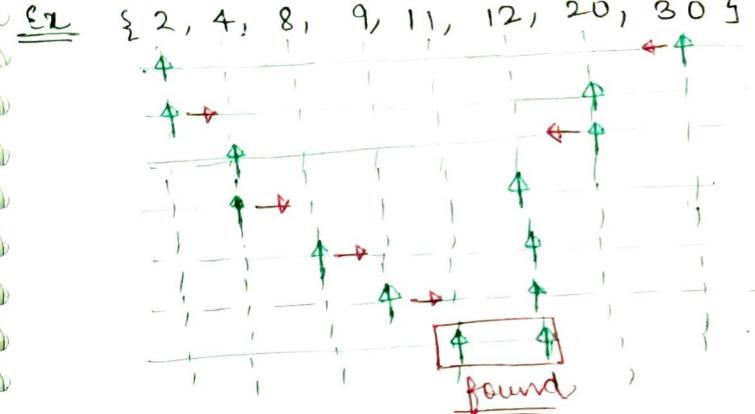
x = 14 O/P  $\rightarrow$  NO

### Two Pointer Approach



### Naive Solution

```
for (i=0 to n)
    for (j=i+1 to n)
        if (arr[i] + arr[j] == x)
            return true;
    return false;
```



### Algorithm

```
bool isPair (int arr[], int n, int x) {
    int low = 0, high = n-1;
    while (low < high) {
        if ((arr[low] + arr[high]) == x)
            return true;
        if ((arr[low] + arr[high]) > x)
            high--;
        if ((arr[low] + arr[high]) < x)
            low++;
    }
    return false;
}
```

### TRIPLETS having sum = x

#### Naive Solution

Run three loops ( $O(n^3)$ )

#### Efficient

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

I/P  $\rightarrow \{2, 3, 4, 8, 9, 20, 40\}$   
x = 32

O/P  $\rightarrow$  Yes (4 + 8 + 20)

## Median of two sorted Arrays

$$\text{Ans} \quad a_1[] = \{10, 20, 30, 40, 50\}$$

$$a_2[] = \{5, 15, 25, 35, 45\}$$

$$\text{O/P} \rightarrow \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$$

$$\text{median} \rightarrow 25 + 30 = 27.5$$

as elements were even hence median will be mean of middle two

$$\text{I/P} \rightarrow a_1[] = \{1, 2, 3, 4, 5, 6\}$$

$$a_2[] = \{10, 20, 30, 40, 50\}$$

$$\text{O/P} \rightarrow \{1, 2, 3, 4, 5, 6, 10, 20, 30, 40, 50\}$$

median

$$\text{I/P} \rightarrow a_1[] = \{10, 20, 30, 40, 50, 60\}$$

$$a_2[] = \{1, 2, 3, 4, 5\}$$

$$\text{O/P} \rightarrow \{1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 60\}$$

median

### Naive Solution

copy elements of  $a_1$  and  $a_2$  to temp and merge sort temp and if  $(n_1 + n_2) \rightarrow$  even median  $\rightarrow$  mean of middle two

If  $(n_1 + n_2) \rightarrow$  odd then median  $\rightarrow$  middle element.

### Efficient Solution

we will used binary search for the same.

we will divide elements in the left half and right half and ensure that both halves contain equal elements.

If elements are odd then left half contains one more elements

i<sub>1</sub> → starting of right set in  $a_1$   
i<sub>2</sub> → starting of right set in  $a_2$ .



$$\left[ \begin{matrix} \text{middle} \\ \frac{n_1+n_2}{2} \end{matrix} \right] = i_1$$

we try to achieve a situation where all the elements on left half is smaller than all the elements on right half.

### Assumptions

\*  $A_1$  is of smaller size than  $A_2$ , if user provides  $A_2$  of bigger size we can swap pointers in

- we divide elements in two sets,
- 1<sup>st</sup> set → some elements from left of  $a_1$  + some elements from left of  $a_2$

- 2<sup>nd</sup> set → some element from right of  $a_1$  + some element from right of  $a_2$ .

- we start by middle index of first array  
ie  $i_1 = \frac{(0+n)}{2}$

and then we will pick  $i_2$  such that elements are equal in both the sets.

for every  $i_1$  to achieve this condition the corresponding  $i_2 = \left[ \frac{n_1+n_2+1}{2} \right] - i_1$

- If we have all the elements smaller on left side than right side, we stop and find our median

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

Left Half

$a_1[i_2 \dots n_1-1]$   
 $a_2[i_2 \dots n_2-1]$

Right Half

- We will start binary search with array 1 then we'll compare  $i_1, i_1-1, i_2, i_2+1$  to check the base case

$i_1 \rightarrow$  beginning of ~~left~~ right side  $a_1$  (min 1)

$i_1-1 \rightarrow$  end of left side  $a_1$  (max 1)

$i_2 \rightarrow$  beginning of right side of  $a_2$  (min 2)

$i_2+1 \rightarrow$  end of left side  $a_2$  (max 2)

double getMedian(int a1[], int a2[], int n1, int n2)

int begin = 0, int end = n1

while (begin <= end) {

int i1 = (begin + end)/2;

int i2 = (n1 + n2 + 1)/2 - i1;

corner cases when whole array falls in one set

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 ((double)(max(max1, max2) + min(min1, min2))/2);

else if

return (double) max(max1, max2);

else if (max1 > min2) end1 = i1-1;  
 else begin = i1+1;

two cases, whether it's to be shifted right or left

### Majority Element

→ An element is called majority if it appears more than  $n/2$  times (we can have max 1 majority element)

#### Naive Solution

Run two loops and increment the count for every element and return the index if count >  $n/2$ .

$\Theta(n^2)$

#### Efficient solution (Moore's Voting Algorithm)

- If there is a majority element in an array then the candidate we have calculated is going to be the majority element.

→ Phase - 1 (calculate the candidate which may be a majority)

→ Phase - 2 (checks whether the candidate is majority or not)

- We initialise the first element itself as majority and run a loop for  $i = 1 \rightarrow n$ . Then if arr[i] == arr[majority] count++ arr[i] != arr[majority] count--

If after certain steps count reaches to zero we change the majority index = i & count = 1

then we compare that element arr[majority] and with calculate the count if count >  $n/2$  return true.

```
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)  
        return res;  
    else  
        return -1;
```

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

output = 6(majority). [index returned = 3]

Time complexity  $\rightarrow O(n)$

### Phase-1

calculating  
the expected  
majority  
if exists

### Phase-2

for checking  
if the  
candidate  
is majority  
or not.

### Working of this algorithm

Ex-1 6, 8, 7, 6, 6.

all the elements whose frequency are less tend  
to cancel each other count. And at last only  
that element whose frequency is greater than  
other element have count != 0.

### Repeating Element

- Array size  $\geq 2$
- All elements are present  
exactly once except one  
element with repeats  
(any number of times)
- $0 \leq \max(\text{arr}) \leq n-2$

### Super Naive Solution

Run 2 loops and check if any element is  
present twice, if we find one, we return it

- $O(n^2)$  time complexity
- $O(1)$  space complexity

### Naive Solution

- Sort the array, and check if adjacent element  
are same.
  - if ( $\text{arr}[i] == \text{arr}[i+1]$ )  $\rightarrow$  return  $\text{arr}[i]$ .
  - $O(n \log n)$  time comp.
  - $O(1)$  space complexity

### Efficient approach

- have a boolean array of same size as that of  
array, and keep the elements as indexes

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

for (int i = 0; i < n; i++) {

    if (visited[~~i~~]) return  $\text{arr}[i]$ ;  $\rightarrow \text{arr}[i]$  is present  
    visited [ $\text{arr}[i]$ ] = true;  $\exists$

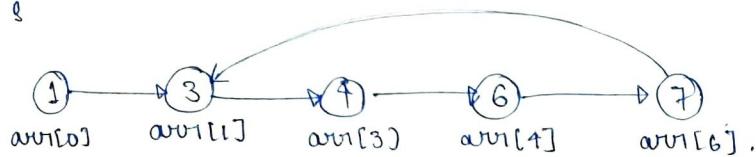
-  $O(n)$  time complexity

-  $O(n)$  space complexity

Efficient Solution (we are calculating when smallest count is 1)

- we will form a chain and then search for a loop.

arr[]  $\rightarrow \{1, 3, 2, 4, 6, 5, 7, 3\}$ .



It shows that two occurrences have same value, that's why we have a loop on '3'.

To find the loop we

① check if loop is present.  
slow  $\rightarrow$  move by 1  
fast  $\rightarrow$  move by 2

② calculate the value of starting of loop

```
int findRepeating (int arr[], int n), {  
    int slow = arr[0], fast = arr[0];
```

```
do {  
    slow = arr[slow];  
    fast = arr[arr[fast]];
```

3 while ( $slow \neq fast$ );

slow = arr[0];

while ( $slow \neq fast$ ) {

slow = arr[slow];

fast = arr[fast];

}

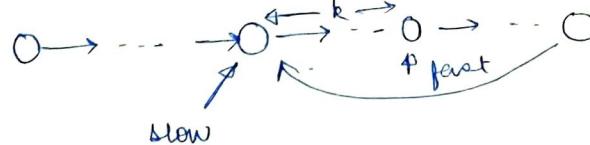
return fast;

} Phase-1

} Phase-1

} Phase-2

Proof of 1st phase



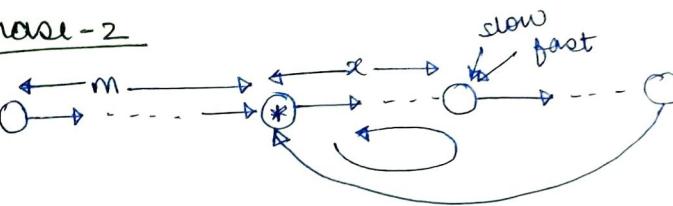
slow

fast

- fast will enter the loop first or at the same time if (loop starts at first element)

- in each iteration - the gap between - increases by 1 and will definitely become  $m$  (the no of nodes in the ~~no~~ cycle).

Phase-2



- move slow to the front  $\Rightarrow$  move both of them at same speed.

- Before first meet

$$\text{Fast distance} = 2(\text{slow distance})$$

$$m + 2x + i(c) = 2(m + x + cx)$$

$$m + x + ci = 2(m + x) + 2(cj)$$

$$m + x = c(i - 2j)$$

it loop covered by fast  
j  $\rightarrow$  loop covered by slow

$m + x$  is the multiple of  $c$  (cycle length).

now if fast start to move only by one  $\Rightarrow$  then where will it reach after one iteration  $\Rightarrow$  at the same point right? Now if we subtract  $x$  from it (to mean if it move then it will be at  $x$  distance before or at  $\sqrt{x}$  root (and in the meanwhile slow will also reach \* by covering the same distance

Implementation when smallest element is zero

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