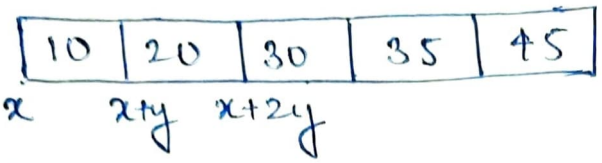
Array

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



x → base address
y → 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.

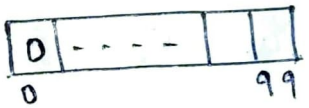
* In Java all the array types (fixed) are heap allocated

(b) Dynamic size array.

Resize themselves automatically

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

Example implementation



If 101st element is inserted

Doubles the size.

- If 201st element is inserted

Again doubles up the size

Operations in Array

(a) Searching an Element (linear)

I/P → {10, 20, 30} S → 20

O/P → 1 (index)

I/P → {20, 5, 7, 30} S → 8

O/P → -1 (not present)

Time complexity - O(n)

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 at the end → $\frac{\overbrace{O(1)+O(1)+O(1)+\cdots}^{n \text{ operation}}+\overbrace{\Theta(n)}^{(n+1)^{th}\text{ op.}}}{n+1}$

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

(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 ≥ 2 i--

arr[4] = arr[3] {5, 7, 10, 20, _} → {5, 7, 10, 20, 20}

arr[3] = arr[2] {5, 7, 10, 20, 20} → {5, 7, 10, 10, 20}

arr[2] = arr[1] {5, 7, 10, 10, 20} → {5, 7, 7, 10, 20}

arr[2-1] = arr[1] = 3

{5, 3, 7, 10, 20}

O(n)

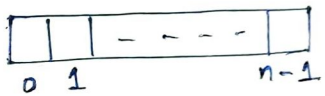
```
int insert (int arr[], int n, int x, int cap, int p){
  if (n == cap){
    return n;
  }.
  for (int i=n-1; i>=p; i--){
    arr[i] = arr[i-1]
  }
  arr[p-1] = x;
  return n+1;
}.
```

Time complexity

(a) worst case → insertion value at front

(b) best case → insertion at back

Insert at end in dynamic size arrays



time complexity of insertion at the end → O(1) [like fixed size arrays. until array has capacity

Deletion of Array elements

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

②
```
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; j<n-1; j++){
    arr[j] = arr[j+1];
  return (n-1);
}
```

Time complexity

for searching, we traverse the left part of array & 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 - Θ(n)

Ex:- {10, 20, 30, 50}
x = 20
O/P → {10, 40, 50}.
return (3)

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-1; i++){
        if (arr[i] < arr[i+1])
            return false;
    }
    return true;
}.
```

I/P → {10, 20, 5}
O/P → false.
10, 20, 30
10, 20, 15
10, 20, 5

Time comp. → O(n)

Reverse An array

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

~~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--;
    }.
}.
```

swap extreme values.

I/P → 10 5 7 30
low (0), high (3)

30 5 7 10
low (1), high (2).

O/P → 30 7 5 10.

Time-complexity → O(n) loop runs 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 RemoveDup (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}.
last → 8, i = 1
d → 10
10 30 10

Naive Solution

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

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

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

O/P → {10, 20, 30, 30, 30, 30} return 3.

Left Rotate an array by one

Right-Rotate → (i+d)%n

I/P → {1, 2, 3, 4 5}
O/P → {2 3 4 5 1}

~~(i-1)%n~~

(4-1)%3 = 3

i	↓
0	4
1	0
2	1
3	2
4	3

(i+n-1)%n

~~arr~~ answer[(i+4)%n] = arr[i]

① ~~en~~ shifting array ~~while during~~ taking input

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

"left" Rotate [i+n-d)%n]
O(n)

② By using different array:

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

OR

```
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

d ≤ no of elements

Must do: d = d%n

① Naive Solution

Rotate array by 1 d times
time complexity → O(nd)
auxiliary space → 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=0; i<d; i++){
        ~~arr[i+d] = temp[i]~~.
        arr[n-d+i] = temp[i];
    }
}
```

time comp → O(n)
auxiliary → O(n)

③ Most Effective solution

```
void leftRotate(int arr[], int n, ~~int~~ 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--; }.
```

For Right Rotate

rev(arr, 0, n-1)	(a, n-d, n+
rev(arr, 0, d-1)	(a, 0, n-d-
rev(arr, d, n-1)	(a, 0, n-1

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}

$O(d + (n-d) + n)$

↓

$O(2n) \rightarrow O(n)$

auxilary sp → O(1)

Leader in an Array

I/P → arr[] → {7, 10, 4, 3, 6, 5, 2}

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

O/P → 10, 6, 5, 2 (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 — $O(n^2)$

```
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]);
    }
}
```

Efficient solution :-

check if current element is greater than last leader.

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

O/P = 2, 5, 6, 10

```
void leader (int arr[], int n){
    int curr_ldr = arr[n-1];   // last element is always a leader
    for (int i = n-2; i >= 0; i--){
        if (arr[i] > curr_ldr){
            curr_ldr = arr[i];
            print (curr_ldr);
        }
    }
}
```

$O(n)$

Drawback → print leaders 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 O(n)

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

I/P → {2, 3, 10, 6, 4, 8, 1}

O/P → 8 (10-2)

I/P → {7, 9, 5, 6, 3, 2}

O/P → 2

- If array is sorted diff = last el - first el
- If arr. is reverse sorted diff = max not depends

Naive solution — $O(n^2)$

```
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;
}
```

Efficient solution

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

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

Time comp.
$\Theta(n)$

Auxiliary space
$\Theta(1)$

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

j=1 →	res = 1	min = 2
j=2	res = 8	min = 2
j=3	res = 8	min = 2
j=4	res = 8	min = 2
j=5	res = 8	min = 2
j=6	res = 8	min = 1

Frequency of Elements in Sorted Array

I/P → {10, 10, 20, 20, 20}
O/P → 2, 3

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

```
void frequency (int arr[], int n){
    int freq = 1  int i = 1
    while (i<n){
        while (i<n && a[i-1] == a[i]){
            freq++;
            i++;
        }.
        cout <<' frequency for the element' << freq;
        freq = 1
        i++;
    }.
```

$\Theta(n)$

arr[] = {10, 10, 20, 30, 30, 30}
(2) (1) (3)

i = 1
freq = 2

~~Stock Buy and Sell~~

~~I/P → {1, 5, 3, 8, 12}~~
~~B S B S~~

~~O/P → 4~~

Stock Buy and Sell

I/P → {1, 5, 3, 8, 12}
B S B S

O/P → (5-1) + (12-3) → 4 + 9 = 13

I/P → {30, 20, 10}

O/P → 0 (If prices are reverse sorted, will will have a profit of 0)

I/P → {10, 20, 30}

O/P → 30 - 10 = 20 (If prices are sorted, then, Profit will be last item - first

I/P → {1, 5, 3, 1, 2, 8}
O/P → B S B S

O/P → (5-1) + (8-1) → 11

Approach - 1

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

Flow

arr[] → {1, 5, 3, 8, 12}
start → 0
end → 4

i = 0
j = 1

① price[j] = 5 > 1 (price[i]) → ④ → ⑬ (start > end)
maxProfit (price, 0, -1) → 0
max Profit (price, 2, 4) → ⑨

1, 5, 3, 8, 12
profit of 3

i = 2
j = 3 → profit = 5 + mp(price, 2, 1) → 0
mp(price, 3, 4) →
5 + ④ → ⑨

Approach-2 (Efficient solution)

{1, 5, 3, 8, 12}

- buy the stock at the bottom and sell it at ~~top~~ 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}

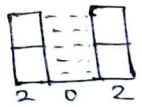
profit = 0

i = 1	p[i] - p[i-1] → 5-1 → 4>0	profit = 4
i = 2	p[2] - p[1] = -2 < 0	p = 4
i = 3	p[3] - p[2] = 8 - 3 → 5>0	p = 4+5 = 9
i = 4	p[4] - p[3] = 12-8 → 4>0	p = 9+4 = 13

Trapping Rain water

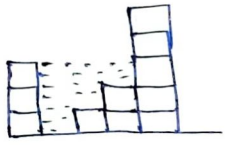
I/P → arr[] = {2, 0, 2} (3 bars of different heights)

O/P → 2 x 1



O/P → arr[] = {3, 0, 1, 2, 5}

O/P →
3+2+1 = 6



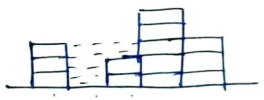
I/P →
arr[] → {1, 2, 3}

O/P → 0

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 = 0 2

start from i = 1 & end to n-2. Bcz extreme ends can't store anything

- initialise lmax and rmax with height of bar

lmax = 2 → on traversing left we found ~~no~~ bar of height greater than lmax = 3

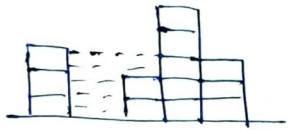
rmax = 2 → on traversing right we found arr[i] = 5 hence rmax = 5

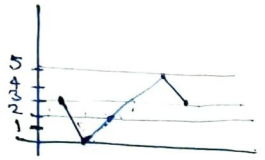
Now for i = 2 min(lmax, rmax) − height of bar

= min(5, 3) − arr[2]

= 3 − 2 = (1)

I/P → {3, 0, 2, 5, 3}





I/P → {5, 7, 9, 10, 3}

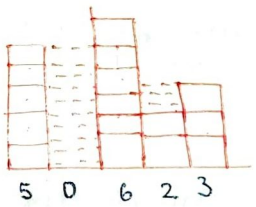
```
int getWater(int arr[], int n){
    int res = 0;
    int lmax = 0, rmax;
    for (int i = 1; i < n-1; i++){
        lmax = arr[i];
        for(int j = 0; j < lmax; j++){
            if(arr[j] > lmax) lmax = arr[j]; }
        rmax = arr[i];
        for (int k = i+1; k < n; k++){
            if (arr[k]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



5 0 6 2 3

i = 1 2 3

lmax → {5, 5, 6, 6, 6}

rmax → {6, 6, 6, 3, 3}

res → {* 5 0 1 *}

= (7) 5+1 → (6)

Time comp → $\theta(n)$

Auxiliary space → $\theta(n)$

Maximum Consecutive 1s in Binary Array

~~int maxConsecutiveOnes (boolean arr[]~~

- maintain a current count • if arr[i] = 1 curr++
  if arr[i] = 0 max(res, cur);

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

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

res = ~~0~~ 2

curr = ~~0~~ ~~1~~ ~~2~~ ~~0~~ ~~1~~ ~~2~~ 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 → -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(n²)

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

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 (Kadane's Algorithm)

{-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}

-5 → 1 → -2 → 3 → 4

1 → -4 → +1 → -1

-2 → -6 → 2

3 → -3

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

~~max~~

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

```
int findmax (int arr[], int n){
    int maxEnding[n]; int res = 0;
    ~~maxEnding[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++){
        ~~max~~ res = max (res, maxEnding[i]);
    }
    return res;
}
```

O(n)

```
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 → {7, 10, 13, 14}

O/P → 4

I/P → {10, 12, 8, 4, 6}

O/P → 1 (because whole array is even)

O(n²) → 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]

```
~~for~~ 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;
        }
    }
    Returns res;
}
```

arr[] → {5, 10, 20, 6, 3, 8}

i = 0 res = 1, curr = 1

i = 1 curr = 2, res = 2 (3)

i = 2 curr = 1

i = 3 curr = 1

i = 4 curr = 2, res = 2

i = 5 curr = 3, res = 3

Maximum subarray sum

{10, 5, −5}

↳ normal subarray → {10}, {5}, {−5}, {10,5}, {10,5,−5}, {5,−5}

↳ circular subarray → {−5,10,5}, {−5,10}, {5,−5,10}

I/P → {5,−2,3,4}
O/P → 12

I/P → {2,3,−4}
O/P → {5}

I/P → {8,−4,3,−5,4}
O/P → 12

I/P → {−3,4,6,−2}
O/P → 10

I/P → {−8,7,6}
O/P → 13

I/P → {3,−4,5,6,−8,7}
O/P → 17

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

arr[] → {5, −2, 3, 4}

i = 0 curr_max = 5, curr_sum = 5

j = 1 curr_sum → 5 + (−2) = 3, curr_max → 5

i = 2 curr_sum = 3, curr_max = 3

j = 1 curr_sum → 3 + 4 = 7, curr_max → 7

j = 2 (i+j)%n → (2+2)%4 → 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 nullify the effect 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 + a[i] & last a[i]
- invert arr[i] → −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 = res max (x~~
        maxEnding = max (maxEnding + arr[i], arr[i])
        res = max (res, maxEnding);
    }
    return res;
}.

int overallMax (int arr[], int n){
    int max_normal = FindMax (arr, n);
    if (max_normal < 0){
        return max_normal;
    }
    int sum = 0;
    for (int i=0; i<n; i++){
        sum = sum + arr[i];
        arr[i] = -arr[i];
    }
    int max_circular = FindMax (arr, n) + sum;
    res = max (max_normal, max_circular);
    return res;
}.
```

All elements are -ve, hence max sum will be the max element itself.

max_circular = sum + FindMax(arr, n)

Returns the negative of minimum sums. (Hence we will directly add it to find max_circular)

{8, -4, 3, -5, 4}
(3) -4 7 2 6

inverted

-8, 4, -3, 5, -4
-8 -4 -7 -2 -6
-8 4 1 (6)

arr_sum = (6)

max_circular → 6 + 6 = 12
max (8, 12) → 12

Majority Element

- Ate 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;
    }
    return -1;
}.
```

loop breaks when func. returns when it found such element.

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[res] == arr[i])
            count++;
        else
            count--;
        if (count == 0) {res= i; count = 1;}
    }.
```

et// cntd.

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

I/P → {8, 8, 6, 6, 6, 4, 6}

- 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

res = 3 count = 1

i = 4 count = 2
i = 5 count = 1
i = 6 count = 2

comparing every element with arr[res] → arr[3]

6 → 4 > [7/2]

O/P → 6

Time complexity → O(n)

I/P → {6, 8, 4, 8, 8}

count → 1 res = 0

i = 1 → 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 Minimum Flips to Make Same

→ allowed to do only consecutive flips in one go (minimum go required)

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 → 1 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 seq. 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 then it with no zeroes between the no of ~~zeo~~ zero grp → 0

diff → 1

1st case 111 --- (diff = 1)
2nd case 1110000 (diff = 0)
3rd case 1110001 diff = 1

As the difference is always gonna 1 or 0 so we make a rule to have a flip on second group. (Inner 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 1st element will always be the member of 1st group)

To check whether it is the starting of group we want to print or ending of group we compare it with 1st element ~~of 1st elem~~ if it's diff then it is starting

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

I/P → 0 0 1 1 0 0 1 1 0

i=0 i=1 arr[1] == arr[1-1]

i=2 arr[2] != arr[1]

↳ enter in condition to check whether it is starting of 2nd group or ending we comp it with arr[0]

To have starting element ≠ arr[0]

From 1 to

i=3 i=4 ——x——

arr[i] != arr[i-1]

& arr[i] == arr[0]

2 (newline)

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 → {1, 8, 30, -5, 20, 7}

k=3

39 33 45 23

O/P → 45

I/P → {5, -10, 6, 90, 3}.

k=2

O/P → 90

Naive Approach

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

```
int maxConsecutiveSum (int arr[], int n){
    int max_sum = INT_MIN;   ← used res inside code (res = max_sum)
    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)×k) ~ O(n

Effective Approach (Window Sliding Technique) O(n)

- compute the sum of 1st window (1st 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;}
```

Window Sliding Technique to print a subarray whose sum is given

- only applies on non-negative elements

I/P → {1, 4, 20, 3, 10, 5} Sum → 33

O/P → ~~{20, 3, 10}~~ true (20, 10, 3).

We start by 1st 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;
        if (e < n)
            curr_sum += arr[e];
    }.
    return (curr_sum == sum);
}
```

T.C → O(n)
A.S → O(1)

Exercise

② N- bonacci Numbers

↓

febonacci → 2-bonacci (every element is sum of previous 2)

3-bonacci → every element is sum of previous 3.

N-bonacci → every element is sum of previous n.

I/P. N = 3 M = 8 → No of elements to be printed

3-bonacci

- first n-1 elements is going to be zero & next element is always 1

O/P → 0 0 1 1 2 4 7 13 ...

Ques 2 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.

Prefix sum

Q/ Given a fixed array & multiple queries of following types on the array, how to efficiently perform the queries → getSum(0,2)
getsum(0,3)
getSum(2,6)

we have to find sum of elements between these indices & including these indices

arr[] → {2,8,3,9,6,5,4}
(0 1 2 3 4 5 6)

getsum(0,2) → 2+8+3
→ 13

getsum(2,6) → 3+9+6+5+4
→ 27

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

arr[] → {2,8,3,9,6,5,4}

pre-sum[] → {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.

getsum(l,r) → pref-sum[r] - pref-sum[l-1]

getsum(3,6) → {2,8,3,9,6,5,4} → 37 - 13 = 24
(sum)

- l=0 (case explicitly handled).

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

Q// Find if an array has an equilibrium point.

(a) {3,4,8,-9,20,6} → Yes

3+4+8+(-9) ← ↑ → 6
= 6

(b) {4,2,-2} → Yes.

0 ← ↑ → 2+(-2)
= 0

(c) {4,2,2} → NO

NO equilibrium pt.

sum → total sum of array
P[] → prefix array

we can check if an element is equilibrium point if

sum - p[i] = p[i-1]

sum elements before it = sum of element after it

O(n) time & O(n) Auxiliary space →

- O(1) Auxiliary space

```
int sum=0
for (int i→n){
    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_sum = l_sum + arr[i]
    sum = sum - arr[i]} return false;
```

// Given n ranges, we have to find max. occ. element in these ranges.

L[] → {1, 2, 3}
R[] → {3, 5, 7}

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

vector<int> arr[1000];

0	1	2	3	4	5	6	7	8	9	10	11	12
	1	1	1									
				-1		-1		-1				

arr[pref] → 0 1 2 3 2 2 1 1 0 0 0 0
(0 1 2 3 4 5 6 7 8 9 10 11)

- By calculating the prefix sum, we keep track of frequency of certain element.
  ① if array falls 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;
}
```

Questions on Prefix sum

① Check if given array can be divided into three parts with equal sum
② Check if there is a subarray with 0 sum
③ Find the longest subarray with equal 0's & 1's

Searching

Binary Search

I/P → {10, 20, 30, 40, 50, 60}
x = 20
O/P → 1

I/P → {10, 10, 20}
x = 10
O/P → 0 or 1

I/P → {5, 15, 20}
x = 25
O/P → -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{
            return mid;
            break;
        }
    }
    return -1;
}
```

I/P → 10, 20, 30, 40, 50, 60 | x>25 | begin → 0, end → 5

mid = (0+5)/2 → 2

30 > 25

O/P → -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);
}
```

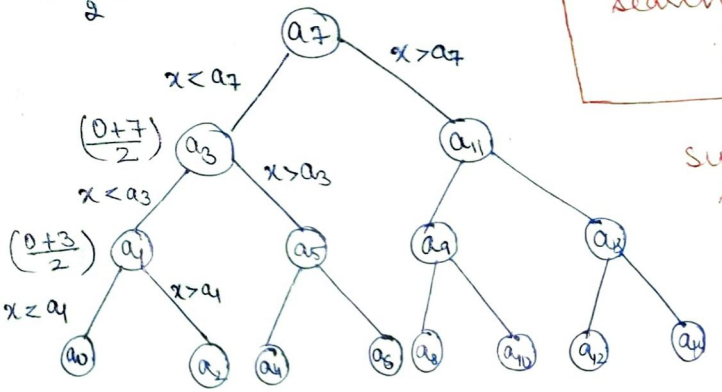
- **comparing iterative and recursive approach**

Time-complexity → same in both (O(logn))
Auxiliary space → iterative → O(1) space
recurve → O(logn) ← for storing recursion call stack.

Analysis of Binary search

a_0	a_1	---	a_6	a_7	a_8	---	a_{14}	a_{15}

mid = (0+15)/2 = 7



For unsuccessful search → $\log_2 n$ iterations

Successful search

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

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

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 got the element equal to arr[mid]

Here two ~~case~~ cases arise

① mid == 0 (First occurence 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 occurance →

Not the first occ. →

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 Occurence 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[mid]
```

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 f(arr, begin, mid-1, x);
  }
  else if (arr[mid] < x) {
  return f(arr, mid+1, end, x);
  }
  else{
    if (mid == n-1 ||
        arr[mid] != arr[mid+1])
    return mid;
    else {
      return f(arr, mid+1, end, x);
  }}
  return -1;
}
```

Iterative

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

Count occurences in a sorted Array

we can call first occurence and last occurence bsm and can find the no of occurences

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

check if element is not present

Count 1's in sorted Binary Array

Naive solution → Find index of first 1 and then substract it from size

{0, 0, 1, 1, 1, 1, 1} → 6 - 2 = (4)

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

* Calculate the first occurence using binary search
  time comp. → O(logn)

Calculate the square root

- If square root exists then return it otherwise take the ciel 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 ≤ 9	✓
i = 2	4 ≤ 9	✓
i = 3	9 ≤ 9	✓
i = 4	16 ≤ 9	✗

return 4-1 = 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;
```

storing temporarily ans & wanting for largest ans.

- check if x/2 is square root
  if x/2 then check x/4, then x/8 ...

Search in Infinite sized sorted Array

I/P → {1, 10, 15, 20, 40, 80, 90, 200, 120, 150 ---- } x = 100

O/P → 7

I/P → {20, 40, 100, 300, ··· } x = 50

O/P → -1

- Naive Solution

Time complexity → O(position)

① If element is present then simply O(position)

② If el. is not present then pos → where element should have present in sorted array

```
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;
}
```

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

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
```

```
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)
}
```

because element was greater than prev. i

element was smaller than i.

Time complexity → O(log(pos))

Popularly known as → Unbounded Binary Search

Search in Sorted & Rotated Array

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

I/P → {10, 20, 30, 40, 50, 8, 9}

x = 30

O/P → 2

I/P → {100, 200, 300, 10, 20}

x = 40

O/P → -1

- Here the 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)

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 >= a[low] && x < arr[mid]) {
                high = mid - 1; // Element present in left half
            } else { low = mid + 1}
        }
        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, 90, 67}

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) {
    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];
}
```

checking for corner cases

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

mid is peak element

left part contains peak

Right part contains peak.

Unsorted Array, we have to find ~~number~~ a pair with sum = ~~num~~ x

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

O/P → yes (9, 8)

I/P → {8, 4, 6} x = 11

O/P → 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.

Naive Solution

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

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

I/P → {2, 5, 8, 12, 30} x = 17 O/P → Yes (5, 12)

I/P → {3, 8, 13, 18} x = 14 O/P → NO

Two Pointer Approach

arr[] → {2, 5, 8, 12, 30}

sum → 32 > 17 move the right ptr

sum → 14 < 17 move the left pt

sum = 17 = 17 return true

arr[] → {3, 8, 13, 18}

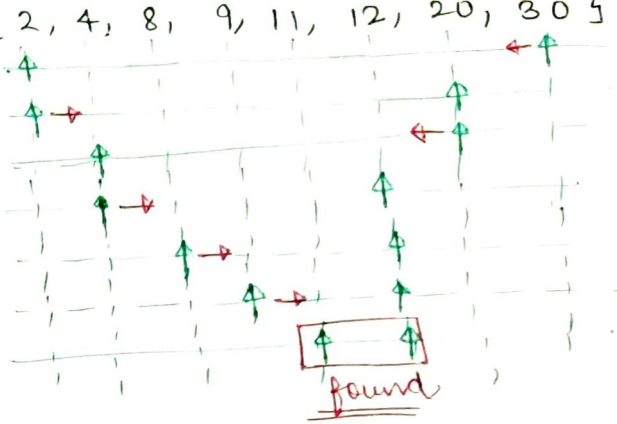
sum = 21 > 14

sum = 16 > 14

sum = 11 < 14

left and right are on same element return false

Ex {2, 4, 8, 9, 11, 12, 20, 30}



sum = 32

sum = 22

sum = 24

sum = 16

sum = 20

sum = 21

sum = 23 (Return true)

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.

I/P → {2, 3, 4, 8, 9, 20, 40} x = 32 O/P → Yes (4+8+20)

Naive Solution

Run three loops (O(n³))

Efficient

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

Median of two sorted Arrays

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

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

o/p → $\{5, 10, 15, 20, \underset{\uparrow}{25}, 30, 35, 40, 45, 50\}$

median → $\frac{25+30}{2} = 27.5$

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

i/p → $a_1[\,] = \{1, 2, 3, 4, 5, 6\}$.

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

o/p → $\{1, 2, 3, 4, 5, \underset{\uparrow}{6}, 10, 20, 30, 40, 50\}$

median

i/p → $a_1[\,] = \{10, 20, 30, 40, 50, 60\}$

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

o/p → $\{1, 2, 3, 4, 5, \underset{\uparrow}{10}, 20, 30, 40, 50, 60\}$

Naive Solution

copy elements of a_1 and a_2 to temp and then sort temp and if (n_1+n_2) → even median → mean of middle two

If (n_1+n_2) → odd then median → 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 half contain equal elements.

If elements are odd then left half contains one more elements

i_1 → starting of right set in a_1

i_2 → starting of right set in a_2.

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

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,

  1st set → some elements from left of a_1 + some elements from left of a_2

  2nd 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 there corresponding $i_2 = \left\lfloor \frac{n_1+n_2+1}{2} \right\rfloor - i_1$

- 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 compared i_1, i_1-1, i_2, i_2-1 to check the base case

i_1 → beginning of ~~left~~ right side a_1 (min 1)

i_1-1 → end of left side a_1 (max 1)

i_2 → beginning of right side of a_2 (min 2)

i_2-1 → 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;

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

corner cases when whole array falls in one set

```
        if(max1 <= min2 && max2 <= min1){
            if((n1+n2)%2 == 0){
                return ((double)(max(max1, max2) +
                        min(min1, min2))/2;
```

if elements are even.

```
            else
                return (double) max(max1, max2);
```

```
        else if(max1 > min2)
            end = i1 - 1;
        else begin = i1 + 1;
    }
```

two cases, whether i_1 is to be shifted right or left

Time complexity → $O(\log n_1)$.

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 increament the count for every element. and return the ~~max~~ index if count > $n/2$.

→ $O(n^2)$

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 intialise the first element itself as majority and run a loop for $i \to 1 \to n$, 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;
        }
    }
```

Phase 1 — Calculating the expected Majority if exists.

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

Phase-2 for checking if the candidate is majority or not.

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

output = 6 (majority). index returned = 3

Time complexity → O(n)

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 ≥ 2
- All elements are present exactly once except one element with repeats (any numbers of times)
- 0 ≤ max(arr) ≤ n-2

I/P → {0, 2, 1, 3, 2, 2}
O/P → 2
I/P → {1, 2, 3, 0, 3, 4, 5}
O/P → 3
I/P → {0, 0}
O/P → 0

Super Naive solution

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

- O(n²) time complexity
- O(1) time complexity

Naive solution

- Sort the array, and check if adjacent element are same.
  if(arr[i] == arr[i+1]) → Return arr[i].
  - O(nlogn) 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

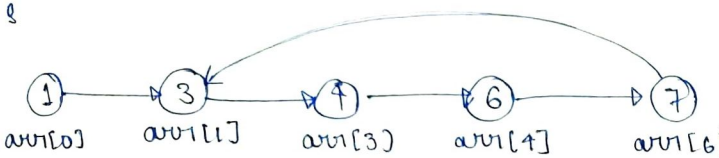
```
visited[] = {F, F, F, ....};
for(int i=0; i<n; i++){
    if(visited[arr[i]]) return a[i]; → a[i] is present already
    visited[arr[i]] = true; }
```

- $O(n)$ time complexity
- $O(n)$ space complexity

Efficient solution (we are calculating when smallest element is 1)

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

arr[] → {1, 3, 2, 4, 6, 5, 7, 3}.
0 1 2 3 4 5 6 7



It shows that two occurances have same value, thats why we have a loop on '3'

To find the loop we

① check if loop is present. } Phase-1
slow → move by 1
fast → move by 2

② calculate the value of starting of loop } Phase-2

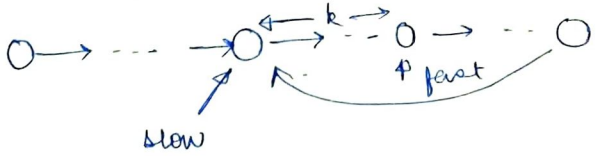
```
int findRepeating (int arr[], int n){
    int slow = arr[0], fast = arr[0];
    do {
        slow = arr[slow];
        fast = arr[arr[fast]];
    } while (slow != fast);

    slow = arr[0];
    while (slow != fast){
        slow = arr[slow];
        fast = arr[fast];
    }
    return fast;
}
```

Phase-1

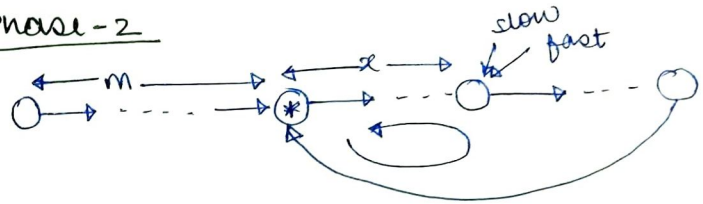
Phase-2

Proof of 1st phase



- fast will enter the loop first or at the same time if (loop starts at first element)
- in each iteration the gap between two increases by 1 and will definitely become m (the no of nodes in the cycle).

Phase-2



- move slow to the front & move both of them at same speed.
- Before first meet

Fast distance = 2 (slow distance)

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

i → loop covered by fast
j → loop covered by slow

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

$\boxed{m + x = c(i - 2j)}$

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

now if fast start to move only by one it then where will it reach after one iteration → at the same point right?. Now if we subtract x from it (i.e mean if it move) then it will be at x distance before or at * 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
    slow = arr[0] + 1;
    while (slow != fast) {
        fast = arr[fast] + 1;
        slow = arr[slow] + 1;
    }
    return slow - 1;
}
```