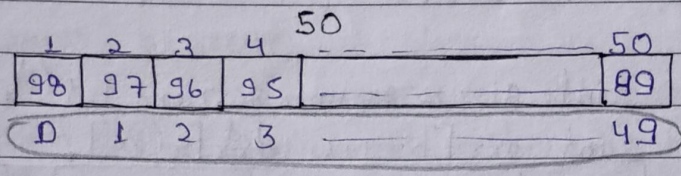
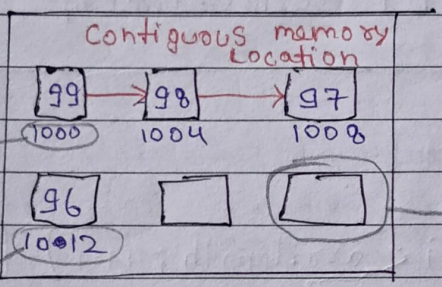
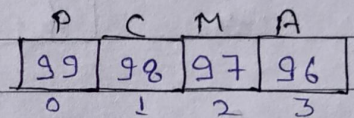
★ ARRAYS

* List of elements of the same data type placed in a contiguous memory location.



→ Zero based indexing (C++, Java)





→ Generally, these block are of 1 byte.

* There addresses are interval of there data type (4 bytes is of int).

• Operations on Arrays :-
1. Create
2. Input
3. Output
4. Update

1. Creating an array

⇒ datatype arrayName[] = new datatype [size];

⇒ datatype[] arrayName = new datatype [size]; (or)

Ex: int marks[] = new int [50];
int num[] = {1, 2, 3};
int morenum[] = {4, 5, 6};
~~int~~ String fruits[] = {"apple", "mango", "orange"};

* Array size is static i.e., size has to changed only at time of declaration.

2. Input of array

Direct Input : `int arr[] = {1, 2, 3};`

User Input :
```
int size = sc.nextInt();
int arr[] = new int[size];
for (int i=0; i<size, i++) {
    arr[i] = sc.nextInt();
}
```

3. Output of array

```
for (int i=0; i< arr.length ; i++) {
    Syso ("Element at index " +i+ " : " +arr[i]);
}
```

4. Update an array

* Direct assignment : `int arr[] = {1, 2, 3, 4, 5, 6};`
  `arr[2] = 10; //Updating by indexing.`

* Using Loops :
```
int [] arr = {1, 2, 3, 4, 5, 6};
for (int i=0 ; i < arr.Length ; i++) {
    if (arr[i] > 3) {
        arr[i] =0;
    }
}
```

1 2 3 0 0 0

Name of array

```
for (int num: arr) {
    Syso (num);
}
```

- Passing arrays as argument by reference.

- Linear Search

* Searching the element one by one i.e, traversing the all elements ~~before~~ (elements before desired element).

* find the index of element in a given array.

2 4 6 8 10 12 14 16

```
public static int linearSearch (int nums[], int key) {
    for (i=0 ; i< nums.length ; i++) {
        if (nums[i] == key) {
            return i;
        }
        return -1;
    }
}
psvm {
    int nums [] = {2, 4, 6, 8, 10, 12, 14, 16};
    int key = 10;
    int index = linearSearch (nums, key);
    if (index == -1) {
        Syso ("NOT FOUND !!! ");
    }
    else {
        syso ("Key is at index: " + index);
    }
}
```

# Time Complexity
O(n)

Key is at index: 5

$-\infty$ → Integer.MIN_VALUE

$+\infty$ → Integer.MAX_VALUE

* Find the largest element in a given array

-1 2 6 3 5

```
public static int largest_number (int nums[]) {
    int largest = Integer.MAIN_VALUE;
    int smallest = Integer.MAX_VALUE;
    for (i=0; i< nums.length; i++) {
        if (largest < nums[i]) {
            Largest = nums[i];
        }
        if (smallest > nums[i]) {
            smallest = nums[i];
        }
    }
    syso ("Smallest value is: " + smallest);
    return largest;
}
psvm {
    int numbers[] = {-1, 2, 6, 3, 5};
    syso ("Largest value is : " + largest_number (numbers));
}
```

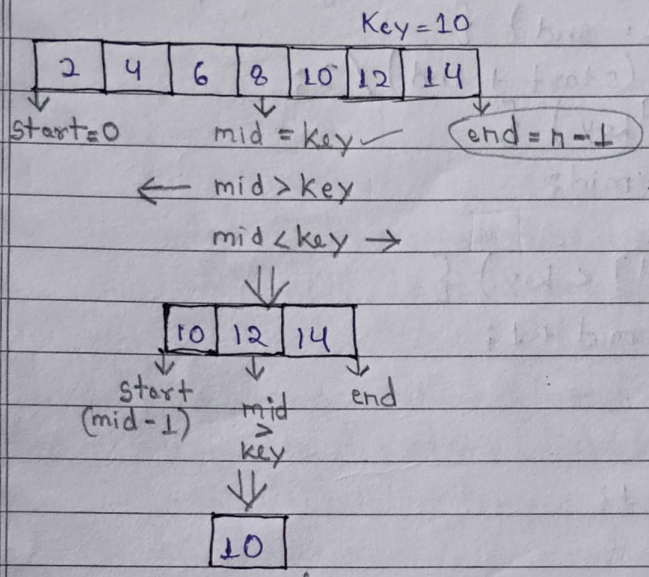
Smallest value is : -1

largest value is : 6

DRY RUN ⇒

Binary Search

It is necessary that array should be sorted.
(ascending or descending)



*Pseudo Code

```
start = 0, end = n-1
while (start <= end)
  find mid (start + end)/2.
  compare mid & key
    mid == key   FOUND
    mid > key    LEFT (1st half)
    mid < key    RIGHT (2nd half)
Element key != arr[]
    return -1;
```

* Linear Search

size = n = 8

O(8)

* Binary Search

8	n
4	n/2
2	n/2
1	1

O(4)

Time Complexity

Iteration-1	n	$n/2^0$
Iteration-2	$n/2$	$n/2^1$
Iteration-3	$n/4$	$n/2^2$
⋮	⋮	⋮
Iteration-n	1	$n/2^k$

$\therefore \frac{n}{2^k} = 1$

$n = 2^k$

$\log n = \log 2^k$

$\log n = k \log 2$

$\rightarrow k = \log_2 n$

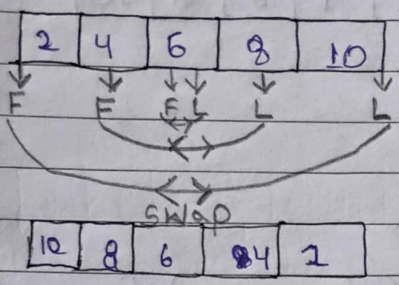
$\therefore O(\log n)$

* Binary Search Code

```
public static int binarySearch (int nums[], int key) {
    int start = 0, end = nums.length - 1; //end = n-1
    while (start <= end) {
        int mid = (start + end) / 2;
        if (nums[mid] == key) {
            return mid;
        }
        if (nums[mid] < key) {
            start = mid + 1;
        }
        else {
            end = mid - 1;
        }
    return -1;
    }
}
public static void main (String args[]) {
    int numbers[] = {2, 4, 6, 8, 10, 12, 14};
    int key = 10;
    Syso ("key is at index : " + binarySearch (numbers,
                                               key));
}
```

DRY RUN ⇒

* Reverse in Array



∵ Space Complexity = O(1)
∵ Time Complexity = O(n).

⇒
```
public static void reverse (int nums[]) {
    int ~~first~~ = 0; last = nums.length - 1;
    while (first < last) {
        int temp = nums[last];
        nums[last] = nums[first];
        nums[first] = temp;
        first ++;
        last ++;
    }
}
public static void main (String[] args) {
    Scanner sc = new Scanner (System.in);
    System.out.print("Enter size of arr : ");
    int size = sc.nextInt();  → int arr[] = new int[size];
    System.out.println ("Enter Elements : ");
    for (int i=0; i<size; i++) {
        arr[i] = sc.nextInt();
    }
    System.out.println ("Reverse of Array: ");
    reversing (arr);
    for (int i=0; i<arr.length ; i++) {
        System.out.print (arr[i] + " ");
    }
    syso();
}
```

// Universal Swap Code

* Pairs in an Array

2	4	6	8	10

(2,4) (2,6) (2,8) (2,10)
(4,6) (4,8) (4,10)
(6,8) (6,10)
(8,10)

⇒
```
public static void pairs_array (int nums[]) {
    int tp = 0; // tp = Total pairs = n(n-1)/2
    for (int i=0 ; i< nums.length ; i++) {
        int curr = nums[i];
        for (j=i+1 ; j < nums.length ; j++) {
            system.out.print ("(" +curr+ "," +nums[j]+ ")");
            tp++;
        }
        syso();
    }
    Syso ("Total Pairs: " + tp);
}
public static void main (Strings args[]) {
    Scanner sc = new Scanner (System.in);
    System.out.print ("Enter size of Array: ");
    int size = sc.nextInt();
    int arr[] = new int [size];
    Syso ("Enter Elements:");
    for (int i=0; i<size; i++) {
        arr[i] = sc.nextInt();
    }
    Syso ("Pairs of Given Array:");
    pairs_array (arr);
}
```

* Print Subarrays

2	4	6	8	10

a continuous part of Array.

Nested Loops

Start : for (int i=0 to n) — L1
End : for (int j=i+1 to n) — L2
for (start to end) — L3

2	2,4	2,4,6	2,4,6,8	2,4,6,8,10
4	4,6	4,6,8	4,6,8,10	
6	6,8	6,8,10		
8	8,10			
10				

⇒
```
public static void subArrays (int nums[]) {
    int tsa = 0; //tsa = Total Sub-Arrays = n(n+1)/2
    for (int i=0; i < nums.length; i++) {
        int start = i;
        for (int j=i; j < num.length; j++) {
            int end = j;
            for (int k=start; k<=end; k++) {
                system.out.print (nums[k] + " ");
            }
            tsa++;
            Syso();
        }
        syso();
    }
    Syso ("Total SubArrays : " + tsa);
}
public static void main (Strings args[]) {
    int arr[] = {2, 4, 6, 8, 10};
    subArrays (arr);
}
```

* Max Subarray Sum - I

(Brute force) → $O(n^3)$

1	-2	6	-1	3

Max = 7

(1) → 1 ; (1,-2) → -1 ; (1,-2,6) → 5 ; (1,-2,6,-1) → 4 ; (1,-2,6,-1,3) → 7

& so on.....

```
public static void max_subarray_sum (int nums[]) {
    int currSum = 0;
    int maxSum = Integer.MIN_VALUE; // -∞
    for (int i=0; i< nums.length; i++) {
        int start = i;
        for (int j=i; j<nums.length; j++) {
            int end=j;
            currSum =0;
            for (int k=start; k<=end; k++) {
                currSum += nums [k];
            }
            Syso (currSum + " ");
            if (maxSum < currSum) {
                maxSum = currSum;
            }
        }
    }
    Syso ("Max Sum = " + maxSum);
}
```

Three Nested Loops

n1 ← for

n2 ← for

n3 ← for

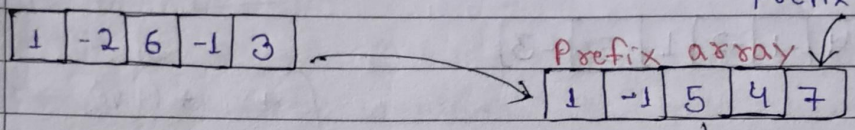
⇒ Time Complexity = $O(n^3)$

// Very Bad, we can write more efficient,

∴ We will use Prefix Sum.

* Max SubArray Sum (Prefix Sum) → O(n²)

prefix[i-1] + arr[i]



Prefix (end) - prefix (start-1)

⇒
```
public static void max_SubArray_Sum (int numbers[]) {
    int currSum=0;
    int maxSum=Integer.MIN_VALUE;
    int prefix[] = new int [nums.length];
    prefix[0] = numbers[0];
    for (int i=1; i<prefix.length; i++) {
        prefix[i] = prefix[i-1] + numbers[i];
    }
```

Two Nested Loops

n1 ← `for (int i=0; i<numbers.length; i++) {`

```
        int start = i;
```

n2 ← `for (int j=i; j<numbers.length; j++) {`

```
            int end = j;
            currSum = 0;
            currSum = start==0 ? prefix[end] : prefix[end] -
                                               prefix[start-1];
            System.out.print (currSum + " ");
            if (maxSum < currSum) {
                maxSum = currSum;
            }
        }
        Syso ("Max Sum = " + maxSum);
    }
}
```

∴ Time Complexity = O(n²).

* Max Subarray Sum (Kadane's Algorithm) → O(n)

-2	-3	4	-1	-2	1	5	-3

	0							
CurrSum	0	0	4	3	1	2	7	4
MaxSum (−∞)	0	0	4	4	4	4	7	7

∴ MaxSum = 7

+ve + (+ve) → +ve
+ve + (-ve) → +ve
+ve + (-ve) → -ve
→ For this, Kadane said make it zero, becoz it gives -ve.

⇒ If in an one-D array all elements are negative then KADANE's Algo will return 0 !!!!!

* Atleast 1 number should be positive.

Case 1: (atleast a +ve number.)

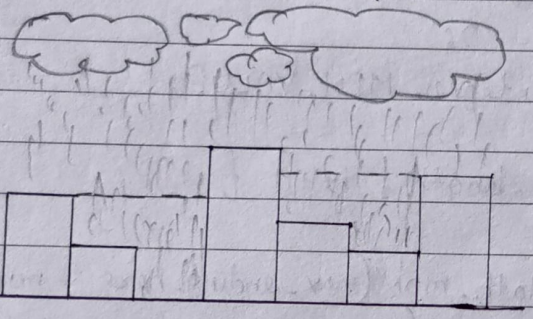
```
public static void ~~main~~ kadanes (int nums[]) {
    int maxSum = Integer.MIN_VALUE;
    int currSum = 0;
    for (int i=0; i < nums.length; i++) {
        currSum = currSum + nums[i];
        if (currSum < 0){
            currSum = 0;
        }
        maxSum = Math.max(currSum, maxSum);
    }
    System.out.print("Our Max Sub-Array Sum is: "+maxSum);
}
psvm (string args[]) {
    int numbers[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    kadanes (numbers);
}
```

Case 2: If all numbers are negative.

```
public static void max_SubArray_negative (int nums[]) {
    int max_so_far = Integer.MIN_VALUE;
    int max_ending_here = 0;
    int max_element = Integer.MIN_VALUE;

    for(int i=0; i<nums.length; i++) {

        max_ending_here = Math.max(max_ending_here + nums[i], 0);
        max_so_far = Math.max(max_ending_here, max_so_far);
        max_element = Math.max(max_element, nums[i]);

    }
    if (max_so_far == 0) {
        max_so_far = max_element;
    }
    System.out.println(max_so_far);
}
psvm (String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter Size: ");
    int size = sc.nextInt();
    int arr[] = new int[size];
    Syso("Enter Elements: ");
    for (int i=0; i< arr.length; i++) {
        arr[i] = sc.nextInt();
    }
    Syso("By Kadane's Algo.");
    Syso("Max Sub-Array Sum: ");
    Kadanes(arr);
}
```

#22 TRAPPING RAINWATER (Medium level)

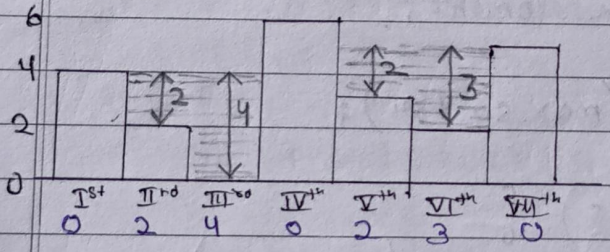
Ques: Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.



height = [4, 2, 0, 6, 3, 2, 5]

* In this, we will learn about Auxiliary Arrays.

⇒ (Waterlevel - barlevel / height) * width



*width = 1.

I st → (0-4) * 1 = -4 means 0.

II nd → (4-2) * 1 = 2.

III rd → (4-0) * 1 = 4

IV th → (0-6) * 1 = -6 means 0.

V th → (5-3) * 1 = 2

VI th → (5-2) * 1 = 3

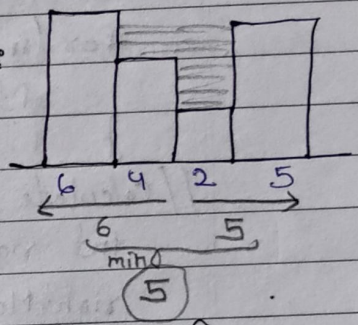
VII th → (0-5) * 1 = -5 means 0.

∴ Total rainwater trapped = 0+2+4+0+2+3+0
= 11.

* How we will calculate Waterlevel in code?

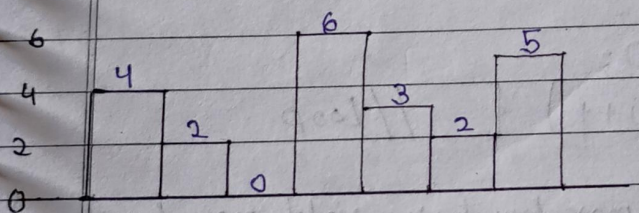
⇒ Minimum no. of bars > 2.
⇒ Ascending / Descending bars, no water is trapped.
⇒ Waterlevel = min (max left, max right)
⇒ Trapped Rainwater = (Waterlevel − barlevel) / width

Ex:



* How we will calculate max left & max right?

We will use an Auxiliary or Helper array.



i.e., Left max boundary : 4 4 4 6 6 6 6
Right max boundary : 6 6 6 6 5 5 5

Now, Index wise Value एक-एक लेकर min value निकालनी

* Auxiliary Array : It is a temporary data structure used to help solve a specific problem or algorithm. It is not directly solving problem, but rather providing support in implementing a solution. Auxiliary Array may no longer be needed.

O(n)

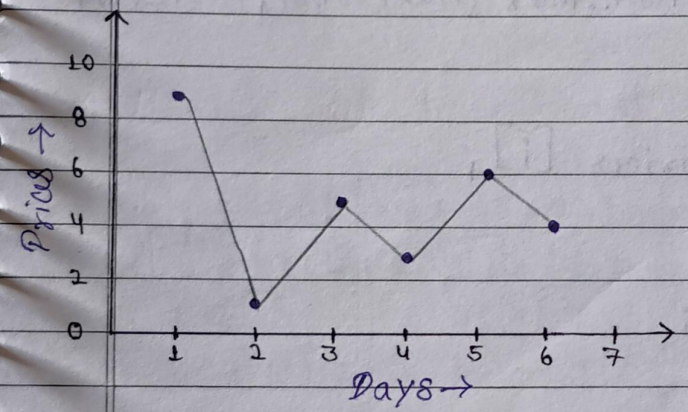
TRAPPING RAINWATER - CODE

```
public static int trapped_rainwater (int height[]) {
    int n = height.length;
//Calculate left-max-boundary using auxiliary array
    int leftMax [] = new int [n];
    leftMax [0] = height [0];
    for (int i=1; i<n; i++) {
        leftMax [i] = Math.max (height [i], leftMax [i-1]);
    }
//Calculate right-max-boundary using auxiliary array
    int rightMax [] = new int [n];
    rightMax [n-1] = height [n-1];
    for (int i=n-2; i>=0; i--) {
        rightMax [i] = Math.max (height [i], rightMax [i+1]);
    }
    int trappedWater = 0;
    for (int i=0; i<n; i++) {    //Loop

// Waterlevel = min (left-max-boundary, right-max-boundary)
        int waterlevel = Math.min (leftMax [i], rightMax [i]);

// trapped water = (waterlevel - height [i]) * width
// width = 1.
        trappedWater += waterlevel - height [i];
    }
    return trappedWater;
}
psvm (String [] args) {
    int height[] = {4, 2, 0, 6, 3, 2, 5};
    Syso (trapped_rainwater (height));
}
```

Buy & Sell Stocks.

You are given an array prices where prices[i] is the price of a given stock on the i^{th} day. You want to maximize your profit by choosing a single day to buy one stock & choosing a different day in the future to sell that stock. Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0. prices = [7, 1, 5, 3, 6, 4]

* profit : Buy ↓ Sell ↑ = Sell − Buy.



* Profit = SellingPrice − buyPrice
↑max ↓min

* Buy Price = ~~7~~, 1
(min. price)

Day 1	Day 2	Day 3	Day 4	Day 5	Day 6
BP = 0	BP = 7	BP = 1	BP = 1	BP = 1	BP = 1
	SP = 1	SP = 5	SP = 3	SP = 6	SP = 4
	P = 1 − 7 = −6	P = 5 − 1 = 4	P = 3 − 1 = 2	P = 6 − 1 = 5	P = 4 − 1 = 3
	No profit	Profit	Profit	Profit (MAX)	Profit.

⇒
```
for (i=0 to n) {
   (BP < SP) {
      P = SP − BP
   }  └→ max?
  (BP > SP) {
     BP = curr Price
  }
}
```

CODE

```
public static int buy_sell_stocks (int prices[]) {

    int buyPrices = Integer.MAX_VALUE;
    int maxProfit = 0;

    for (int i=0; i< Prices.length ; i++) {

        if (buyPrices < prices [i]) {
            int profit = prices [i] - buyPrices;
            maxProfit = Math.max (maxProfit, profit);
        }
        else {
            buyPrices = prices [i];
        }
    }
    return maxProfit;
}

public static void main (String[] args) {

    int prices [] = {7, 1, 5, 3, 6, 4};
    Syso ("Maximum Profit : " + buy_sell_stocks
            (prices));
}
```

Time Complexity, $O(n)$.