HEAPS / PRIORITY QUEUES

★ PRIORITY QUEUE

It is special type of queue in which each element is associated with a priority.

* Elements are dequeued based on their priority, rather than their insertion order.
* Elements with the highest or lowest priority dequeued first.
* In Java, the PriorityQueue class provides an implementation of a priority queue that follows the min-heap property by default, meaning the smallest element is dequeued first.

★ PRIORITY QUEUES IN JCF

i) add() ⟶ O(logn)
ii) remove() ⟶ O(logn) — Because it will do regarding given priorities.
iii) peek() ⟶ O(1)

⇒
```
public static void main (String args[]) {
    PriorityQueue<Integer> pq = new PriorityQueue<>();
    pq.add(3);
    pq.add(1);
    pq.add(4);  //O(logn)
    pq.add(7);
    while (!pq.isEmpty()){
        Syso(pq.peek()); //O(1)
        pq.remove(); //O(logn)
    }
}
```

Output: 1
3
4
7

★ PRIORITY QUEUE FOR OBJECTS

⇒
```
static class Student implements Comparable<Student> {
    String name;
    int rank;
    public Student (String name, int rank)
        this.name = name;
        this.rank = rank;
    @Override
    public int compareTo (Student s2) {
        return this.rank - s2.rank;
    }
}
```

s1: "abc", 12 — s2: "xyz", 15

s1	s2	
"abc" 15	"xyz" 12	① 12 − 15 = −ve = −3 ⟶ obj1 < obj2 (s1 < s2)
"abc" 15	"xyz" 15	② 15 − 12 = +ve = +3 ⟶ obj1(this) > obj2 (s1 > s2)
		③ 15 − 15 = 0 ⟶ equal

```
public static void main (String args[]) {
    PriorityQueue<Student> pq = new PriorityQueue<>();
        for reversing → Comparator.reverseOrder()
    pq.add(new Student("A", 4));
    pq.add(new Student("B", 5));
    pq.add(new Student("C", 2));
    pq.add(new Student("D", 12));
    while (!pq.isEmpty()) {
        System.out.println(pq.peek().name + "->" + pq.peek().rank);
        pq.remove();
    }
}
```

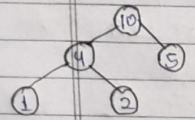
Output: C → 2
A → 4
B → 5
D → 12

Output: (for reverse order) D → 12
B → 5
A → 4
C → 2

★ HEAP

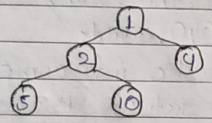
* Visualize as Binary Tree.
* Implementation as Array.

* It is a tree-based data structure that satisfies Heap property.
* Property dictates that the parent node is either greater than or less than its child nodes depending on whether it's a Max or Min Heap



Max Heap

* Max element ↑ priority.



Min Heap

* Min element ↑ priority.

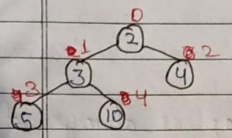
* Heap Property:

1. Heap is a Binary of most 2 children.
2. Heap is Complete Binary Tree.
   → CBT in which all the levels are completely filled except possibly the last one, which is filled from the left to right.
3. Heap Order Property:
   → Children >= Parent (minHeap)
   → Children <= Parent (maxHeap)

★ HEAP IS NOT IMPLEMENTED AS A CLASS (Node Class)

We can implement as class but we will not do usually because the insertion is too complex & the time complexity is more, which makes less efficient. — O(N)

★ HEAPS AS AN ARRAY / ARRAYLIST — O(1)



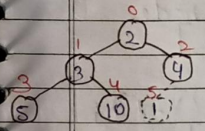
Array:

2	3	4	5	10
0	1	2	3	4

#NOTE: Property of Array for Heaps
(node) idx = i
left child = 2i+1
right child = 2i+2

* We Use ArrayList because we have to show parent-child relationship of a tree.

★ INSERT IN HEAP



MinHeap

→ Insert 1

S1& Arr -

2	3	4	5	10	1
0	1	2	3	4	5

S2: Added at ~~inde~~ last index.

S3: Fix Heap because tree is a minHeap

: child (x) → parent ((x-1)/2) — Universally true

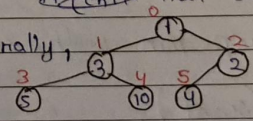
S4: fix Heap: parent = (x-1)/2 = (5-1)/2 = 2

while (child val < parent val)
→ if (child value < parent value)
swap (child, parent)

→ Again compare parent - child,
→ fix Heap: parent = (x-1)/2 = (2-1)/2 = 0

} ~~if (chil~~ then swap (child, parent)

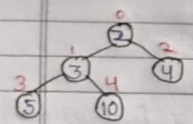
S5: Finally,



→ Arr =

1	3	2	5	10	4
0	1	2	3	4	5

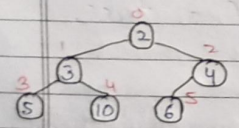
★ GET MIN IN HEAP



⇒ [2 | 3 | 4 | 5 | 10]
0 1 2 3 4

* Minimum in minHeap is always at index 0, so peek().
* peek() → arr.get(0);

★ DELETE IN HEAP



→ [2, 3, 4, 5, 10, 6]
0 1 2 3 4 5

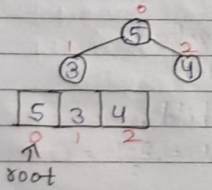
minHeap

S1: Swap first & last node.
→ [6, 3, 4, 5, 10, 2]

S2: Remove last index
→ arr.remove(arr.size()-1);
→ [6, 3, 4, 5, 10]

S3: Now, we have to fix my Heap
↓
Heapify

* Heapify (log n)



root

heapify
root = i
left = 2i+1 = 1
right = 2i+2 = 2
→ min (root, left, right)
↓
root

∴ Swap min with root.

S4: Call Heapify (root);

⇒
```
Static class Heap {
  Arraylist <Integer> arr = new Arraylist <>();
  public static void add (int data) { //O(log n)
    //add at last index
    arr.add (data);
    int x = arr.size ()-1; //x is child index
    int par = (x-1)/2; //parent index
    while (arr.get(x) < arr.get(par)) {
      //swapping
      int temp = arr.get (x);
      arr.set (x, arr.get (par));
      arr.set (par, temp);
      x = par;
      par = (x-1)/2;
    }
  }
  public int peek() {
    return arr.get (0); //Minimum element
  }
  private void heapify (int i) {
    int left = 2*i+1;
    int right = 2*i+2;
    int minIdx = i;
    if (left < arr.size() && arr.get(minIdx) > arr.get(left))
      minIdx = left;
    if (right < arr.size() && arr.get(minIdx) > arr.get(right))
      minIdx = right;
    if (minIdx != i)
      int temp = arr.get(i);
      arr.set (i, arr.get(minIdx)); //swapping
      arr.set (minIdx, temp);
    heapify (minIdx);
  }
```

```
public int remove() {
    int data = arr.get(0);
    //S1 - swap first & Last
    int temp = arr.get(0);
    arr.set(0, arr.get(arr.size()-1));
    arr.set(arr.size()-1, temp);
    //S2 - delete Last
    arr.remove(arr.size()-1);
    //S3 - Fix Heap - Heapify
    heapify(0);
    return data;
  }
  public boolean isEmpty() {
    return arr.size() == 0;
  }
}
public static void main(String args[]) {
    Heap pq = new Heap();
    pq.add(3);
    pq.add(4);
    pq.add(1);
    pq.add(5);
    while (!pq.isEmpty()) {
        Syso(pq.peek());
        pq.remove();
    }
}
```

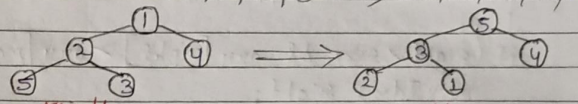
Output: 1
3
4
5

NOTE → For maxHeap, we have only change two signs or operators in Heapify() & one " " " in add().
→ Change minIdx to maxIdx.

★ HEAP SORT - O(n log n)

* If we have to sort in Ascending → make maxHeap.
* If we have to sort in Descending → make min Heap.

→ arr = 1, 2, 4, 5, 3 ⟹ 5, 3, 4, 2, 1



S1: Convert arr → Create maxHeap.
S2: non-leaf nodes = 2, 1
⇓
heapify()
(maxHeap concept)

S3: Largest element push at end.
→ swap (index:0, last)
→ heapify (index:0)
We're removing Largest element at last then removing from Heap to array.

→ 5, 3, 4, 2, 1
→ 1, 3, 4, 2, 5
→ 1, 3, 4, 2 → 1, 3, 2, 4
→ 1, 3, 2 → 1, 2, 3
→ 1, 2 → 1

S4: At Last, → 1, 2, 3, 4, 5
sorted

// NEXT PAGE

⇒
```
public static void heapify (int arr[], int i, int size){
    int left = 2*i + 1;
    int right = 2*i + 2;
    int maxIdx = i;
    if (left < size && arr[left] > arr[maxIdx]) {
        maxIdx = left;
    }
    if (right < size && arr[right] > arr[maxIdx]) {
        maxIdx = right;
    }
    if (maxIdx != i) {
        //swap
        int temp = arr[i];
        arr[i] = arr[maxIdx];
        arr[maxIdx] = temp;
        heapify (arr, maxIdx, size);
    }
}
public static void heapSort (int arr[])
    //S1: Build maxHeap
    int n = arr.length;
    for (int i = n/2; i>=0; i--)
        heapify (arr, i, n);
    //S2: Push largest element at end
    for (int i=n-1; i>0; i--)
        //Swap (Largest element - first with last)
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify (arr, 0, i);
}
```

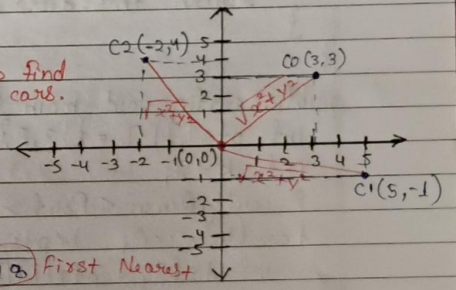


```
public static void main (String args[]) {
    int arr[] = {1, 2, 4, 5, 3};
    heapSort (arr);
    //print
    for (int i=0; i<arr.length; i++){
        System.out.print (arr[i] + " ");
    }
    Sysо ();
}
```

★ NEARBY CARS – We are given N points in a 2D plane which are locations of N cars. If we are at the origin, print the nearest K cars.

→ C0(3,3)
→ C1(5,-1)
→ C2(-2,4)

*K=2
we have to find 2 nearest cars.



⇒ Ans : C0 & C2

* For C0, $\sqrt{3^2+3^2} = \sqrt{18}$ First Nearest
* For C1, $\sqrt{5^2+(-1)^2} = \sqrt{26}$
* For C2, $\sqrt{(-2)^2+4^2} = \sqrt{20}$ Second Nearest

* We will use Priority Queue, ⇒ Dist² ↓ priority ↑

(C0 18) (C1 26) (C2 20)

[C0,18 | C2-20 | C1-26]
PQ

⇒
```
static class Point implements Comparable<Point>{
    int x;
    int y;
    int distSq;
    int idx;
    public Point(int x, int y, int distSq, int idx){
        this.x = x;
        this.y = y;
        this.distSq = distSq;
        this.idx = idx;
    }
    @Override
    public int compareTo(Point p2){
        return this.distSq - p2.distSq;
    }
}
public static void main(String args[]){
    int pts[][] = {{3,3}, {5,-1}, {-2,4}};
    int k = 2;
    PriorityQueue<Point> pq = new PriorityQueue<>();
    for(int i=0; i<pts.length; i++){
        int distSq = pts[i][0] * pts[i][0] + pts[i][1] *
                     pts[i][1];
        pq.add(new Point(pts[i][0], pts[i][1], distSq,
                         i));
    } //nearest K cars
    for(int i=0; i<k; i++){
        Sys o("C" + pq.remove().idx);
    }
}
```


★ CONNECT N ROPES – Given are N ropes of different lengths, the task is to connect these ropes into one rope with minimum cost, such that the cost to connect two ropes is equal to the sum of their lengths.

→ ropes = {4, 3, 2, 6} ⇒ Ans : 29

Ex⇒ connect 2 & 3 → cost = 5
connect 5 & 4 → cost = 9
connect 9 & 6 → cost = 15
→ Total cost = 29.

* Approach

→ ropes = {4, 3, 2, 6}

2 + 3 → cost = 5

5 + 4 → cost = 9 → 5 + 4 (5 = 2 + 3)

9 + 6 → cost = 15 → 9 + 6 (9 = 5 + 4, 5 = 2 + 3)

∴ minimum length → ropes add.

Ex⇒ ropes = {2, 3, 3, 4, 6}

PQ: 2 | 3 | 3 | 4 | 6

* Now, we will take first & second minimum from PQ,

2 + 3 = 5

PQ: 3 | 4 | 5 | 6

⇒ 3 + 4 = 7

5 | 6 | 7

⇒ 5 + 6 = 11

7 | 11

⇒ 7 + 11 = 17

∴ ⇒ 5 + 7 + 11 + 17 = 41

```
⇒ public static void main (String args[]) {
    int ropes[] = {2, 3, 3, 4, 6};
    PriorityQueue<Integer> pq = new PriorityQueue();
    for (int i=0; i<ropes.length; i++) {
        pq.add (ropes[i]);
    }
    int cost = 0;
    while (pq.size() >= 1) {
        int min = pq.remove();
        int min2 = pq.remove();
        cost += min + min2;
        pq.add (min + min2);
    }
    Syso ("Total cost connecting n ropes =" + cost);
}
```

★ WEAKEST SOLDIER — We are given an $m * n$ binary matrix of 1's (soldiers) & 0's (civilians). The soldiers are positioned in front of the civilians. That is, all the 1's will appear to the left of all the 0's in each row.

* A row i is weaker than a row j if one of the following is true:
→ number of soldiers in a row is less than the number of soldiers in row j.
→ Both rows have the same number of soldiers & $i < j$.

∴ Find the K weakest row.

Ex→ m=4, n=4, k=2

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

∴ Ans = row 0 & row 2

* Approach
m=4, n=4, k=2

1 0 0 0 → i=0
1 1 1 1 → i=1
1 0 0 0 → i=2
1 0 0 0 → i=3

* min (weakest) {
soldiers ↓ → weak
soldiers = equal ⇒ i < j
}

Now, Row number [soldiers_count, index]

→ R0 [1, 0]
→ R1 [4, 1]
→ R2 [1, 2]
→ R3 [1, 3]

→ | R0 | R2 | R3 | R1 |
PQ — In ascending order

```
⇒ static class Row implements Comparable<Row> {
    int soldiers; int idx;
    public Row (int soldiers, int idx)
        this.soldiers = soldiers;
        this.idx = idx;
    @Override
    public int compareTo (Row r2) {
        if (this.soldiers == r2.soldiers)
            return this.idx - r2.idx;
        else
            return this.soldiers - r2.soldiers;
    }
}
```

```
public static void main (String args[]) {
    int army[][] = { {1, 0, 0, 0},
                     {1, 1, 1, 1},
                     {1, 0, 0, 0},
                     {1, 0, 0, 0}};

    int k = 2;
    PriorityQueue<Row> pq = new PriorityQueue<>();
    for (int i = 0; i < army.length; i++) {
        int count = 0;
        for (int j = 0; j < army[0].length; j++) {
            count += army[i][j] == 1 ? 1 : 0;
        }
        pq.add (new Row (count, i));
    }
    for (int i = 0; i < k; i++) {
        Syso ("R" + pq.remove().idx);
    }
}
```

★ SLIDING WINDOW MAXIMUM

— Maximum of all subarrays of size k : we have an array arr[] of size N & an integer k. Find the maximum for each & every contiguous subarray of size K.

* We have done this problem using Deque → O(n)

* Now, we will do using Priority Queue → O(n log k)

Ex → 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 , K = 3

K size की window बनानी है।

Return max in pair of 3 windows

∴ Ans = 3, 4, 5, 6, 7, 8, 9, 10

* Approach

n = 8

1, 3, -1, -3, 5, 3, 6, 7 ; k = 3

→ res[] = | 3 | | | | | | (n-k+1)

q = 1 2 3 4 5

S1: add k nums to PQ (1st window)

| 3 | 1 | -1 |

PQ at max K. MaxHeap.

S2: (i) PQ.peek() → window[0]

(ii) while (pq.peek().idx <= (i-k))
pq.remove()

S3: pq.add(curr)
window[] = pq.peek

* For all these we will step by step pair of k inside the pq & remove step by step.

⇓

* At last in res[], answer will be stored :

⇒ res[] : | 3 | 3 | 5 | 5 | 6 | 7 | (n-k+1)

0 1 2 3 4 5

⇒
```
static class Pair implements Comparable<Pair> {
  int val;
  int idx;
  public Pair (int val, int idx) {
    this.val = val;
    this.idx = idx;
  }
  @Override
  public int compareTo (Pair p2) {
    //for ascending
    /* return this.val - p2.val; */
    //for descending
    return p2.val - this.val;
  }
}
public static void main (String args []) {
  int arr [] = {1, 3, -1, -3, 5, 3, 6, 7};
  int k = 3; //window size
  int res [] = new int [arr.length - k + 1]; //n-k+1
  PriorityQueue <Pair> pq = new PriorityQueue<>();
  //1st Window
  for (int i=0; i<k; i++)
    pq.add (new Pair (arr[i], i));
  res [0] = pq.peek().val;
  for (int i=k; i<arr.length; i++)
    while (pq.size() >0 && pq.peek().idx <= (i-k))
      pq.remove();
    pq.add (new Pair (arr [i], i));
    res [i-k+1] = pq.peek().val;
  //print result
  for (int i=0; i<res.length; i++)
    System.out.print (res [i] + " ");
} Syso();
```


QUE. K'th Largest Element in a stream – We have an infinite stream of integers, find the K th largest element at any point of time.

Input: stream [] = {10, 20, 11, 70, 50, 40, 100, 5, ...}, k=3

Output: {_, _, 10, 11, 20, 40, 50, 50, ...} 50

⇒
```
static PriorityQueue <Integer> min;
static int k;
static List <Integer> getAllKthNumber (int arr[]) {
  List <Integer> list = new ArrayList <>();
  for (int val : arr) {
    if (min.size() < k) {          //Time Complexity - O(log k)
      min.add (val);               //Space Complexity - O(1)
    } else {
      if (val > min.peek()) {
        min.poll();
        min.add (val);
      }
    }
    if (min.size() >= k)
      list.add (min.peek());
    else
      list.add (-1);
  }
  return list;
}
public static void main (String[] args) {
  min = new PriorityQueue <>();
  k=4;
  int arr [] = {1, 2, 3, 4, 5, 6};
  List <Integer> res = getAllKthNumber (arr);
  for (int x : res)
    System.out.print (x + " ");
}
```

QUE 2. Minimum time required to fill given N slots –
We have an integer N which denotes the numbers of slots, & an array arr[] consisting of K integers in the range [1, N] supreand. Each element of the array are in the range [1, N] which represents the indices of the filled slots. At each unit of time, the index with filled slot fills the adjacent empty slots. The task is to find the minimum time taken to fill ~~the~~ all the N slots.

Ex → N = 6 (meaning there are 6 slots)
arr[] = [2, 5] (slots 2 & 5 are intially filled)

* At time 0:
  → Initial filled slots : [2, 5]
  → [_ , 1 , _ , _ , 1 , _] (where 1 means filled & _ means empty)
    ① ② ③ ④ ⑤ ⑥

* At time 1:
  → Slot 1 gets filled becoz it's next to slot 2.
  → Slot 3 gets filled becoz it's next to slot 2.
  ⇒ [1 , 1 , 1 , _ , 1 , _]
    ① ② ③ ④ ⑤ ⑥

* At time 2:
  → Slot 4 gets filled becoz it's next to slot 5.
  → Slot 6 gets filled becoz it's next to slot 5.
  ⇒ [1, 1, 1, 1, 1, 1]

∴ So, after 2 units of time, all slots filled.

⇒ Output : 2.


```
⇒ public static void minTime(int arr[], int N, int K){
    Queue<Integer> q = new LinkedList<>();
    boolean vis[] = new boolean[N+1];
    int time = 0;
    for(int i=0; i<K; i++){
      q.add(arr[i]);
      vis[arr[i]] = true;
    }
    while(q.size()>0){
      for(int i=0; i<q.size(); i++){
        int curr = q.poll();
        if(curr-1 >= 1 && !vis[curr-1]){
          vis[curr-1] = true;
          q.add(curr-1);
        }
        if(curr+1 <= N && !vis[curr+1]){
          vis[curr+1] = true;
          q.add(curr+1);
        }
      }
      time++;
    }
    Sys o(time-1);
  }
  public static void main(String args[]){
    int N=6;
    int arr[] = {2,6};
    int K = arr.length;          // Time Complexity - O(N)
    minTime(arr, N, K);          // Space Complexity - O(N)
  }
```

QUE 3: Path with minimum Effort – We have a 2D grid, each cell of which contains an integer cost which represents a cost to traverse through that cell, we need to find a path from the top left cell to the bottom right cell by which the total cost incurred is minimum.

Ex:

1	3	1
2	1	5
4	2	1

Start of top-left corner
End at bottom-right corner.

∴ In this grid, the path that gives you the minimum cost could be:

⇒ 1 + 2 + 1 + 2 + 1 = 7

⇒
```
// a utility class to represent the coordinates of each cell
static class Cell {
    int x, y, cost;
    Cell(int x, int y, int cost) {
        this.x = x;
        this.y = y;
        this.cost = cost;
    }
}
public static int minEffortPath(int grid[][]) {
    int rows = grid.length;
    int cols = grid[0].length;
    // Directions for moving up, down, left, right
    int[] rowDir = {-1, 1, 0, 0};
    int[] colDir = {0, 0, -1, 1};
    Stack<Cell> stack = new Stack<>();
```



```
    stack.push(new Cell(0, 0, grid[0][0]));
    // array to track minimum cost to reach each cell
    int[][] minCost = new int[rows][cols];
    for (int i=0; i<rows; i++) {
        for (int j=0; j<cols; j++) {
            minCost[i][j] = Integer.MAX_VALUE;
        }
    }
    minCost[0][0] = grid[0][0];
    while (!stack.isEmpty()) {
        Cell current = stack.pop();
    }
    // Traverse all possible directions
    for (int i=0; i<4; i++) {
        int newX = current.x + rowDir[i];
        int newY = current.y + colDir[i];
        // check for valid bounds
        if (newX >=0 && newX < rows && newY >=0 && newY < cols) {
            int newCost = current.cost + grid[newX][newY];
            // if a better cost found, update min cost & push to stack
            if (newCost < minCost[newX][newY]) {
                minCost[newX][newY] = newCost;
                stack.push(new Cell(newX, newY, newCost));
            }
        }
    }
    // return min cost to reach bottom-right cell
    return minCost[rows-1][cols-1];
}
    // Time Complexity - O(rows * columns)
    // Space Complexity - O(rows * columns)
```

```
public static void main (String[] args) {
    int[][] grid = { {1, 3, 1},
                     {1, 5, 1},
                     {4, 2, 1} };
    int result = minEffortPath (grid);
    Syso ("The minimum cost to reach the
          bottom-right cell is : " + result);
}
```

QUE 4. Minimum Operations to Halve Array Sum
– We have an array Arr[], the task is to find out the minimum number of operations to make the sum of array elements lesser or equal to half of its intial value. In one such operation, it is allowed to half the value of any array element.

Example: Let's take an array : → Arr[] = [10, 20, 17]

S1: Initial Sum: 10 + 20 + 7 = 37
Half of Sum : 37 / 2 = 18

S2: Operations:
1- halve the largest element (20):
new arr → [10, 10, 7]
new sum → 10 + 10 + 7 = 27
2- Halve the largest element (10):
new arr[] → [5, 10, 7]
new sum → 5 + 10 + 7 = 22
3- Halve the largest element (10):
new arr[] → [5, 5, 7]
new sum → 5 + 5 + 7 = 17

So, the minimum no. of operations is 3.

⇒
```
static int minops (ArrayList<Integer> nums) {
    int sum = 0;
    for (int i=0; i<nums.size(); i++) {
        sum += nums.get(i);
    }
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
    for (int i=0; i<nums.size(); i++) {
        pq.add (-nums.get(i));
    }
    double temp = sum;
    int cnt = 0;
    while (temp > sum/2) {
        int x = -pq.peek();
        pq.remove();
        temp -= Math.ceil (x * 1.0 / 2);
        pq.add (x/2);
        cnt++;
    }                          //Time Complexity - O(nlogn)
    return cnt;                //Space Complexity - O(n)
}
public static void main (String args[]) {
    ArrayList<Integer> nums = new ArrayList<Integer>
        (List.of(4, 6, 3, 9, 10, 2));
    int count = minops (nums);
    Syso (count);
}
```