# Problem 1:

**Minimum Element In A Sorted And Rotated Array**

Find the minimum element in an array that has been sorted and rotated by an unknown pivot.

**Example**

Input: [ 4, 5, 6, 7, 8, 1, 2, 3]

Output: 1

The array is sorted in the ascending order and right rotated by pivot 5. The minimum value 1 is at index 5.

Constraints:

* 1 <= number of array elements <= 10^5
* -10^9 <= any array element <= 10^9
* Array elements are unique.

# Solution

We provided three solutions.

**1) brute\_force\_solution.java**

We iterate over the given array and maintain the minimum value found.

**Time Complexity:**

O(*n*) where *n* is the number of elements in the array.

**Auxiliary Space Used:**

O(*1*).

We are not storing anything.

**Space Complexity:**

O(*n*).

**2) suboptimal\_solution.java**

**Time Complexity:**

Time complexity for the function find\_minimum: O(*log n*) where *n* is number of elements in array.

Time complexity for the complete program: O(*n*) where *n* is number of elements in array, because size of input is *n*.

In this approach we used recursive binary search. If we take some examples and look closely, we would observe some patterns:

If array was previously sorted in ascending order:

* The minimum element is the only element whose previous element is greater than it.
* If we found any subarray ( from low to high ) which is ascending sorted then minimum element will be element at low.
* Else minimum element lies in either left half or right half.
* If middle element is greater than element at low, then the minimum element lies in right half.
* Else minimum element lies in left half.

If array was previously sorted in descending order:

We use these patterns to make solution:

* The minimum element is the only element whose next element is greater than it.
* If we found any subarray ( from low to high ) which is descending sorted then minimum element will be element at high.
* Else minimum element lies in either left half or right half.
* If middle element is less than element at low, then the minimum element lies in right half.
* Else minimum element lies in left half.

As the time complexity of binary search will be

T(n) = T(n/2) + c ( Each iteration reducing array in half ).

The above function can be solved either using recurrence Tree method or Master method. It falls in case II of Master Method and solution of the function is O(*log n*) hence, complexity of our solution (find\_minimum function) is O(*log n*).

**Auxiliary Space Used:**

O(*log n*) where *n* is number of elements in array.

Similarly by above logic for time complexity, number of recursive calls will be O(*log n*) and hence size of function stack used will be O(*log n*).

**Space Complexity:**

O(*n*) where *n* is number of elements in array.

Input is O(*n*) because we are storing *n* elements of array and auxiliary space used is O(*1*). So, O(*n*) + O(*1*) -> O(*n*).

**3) optimal\_solution.java**

**Time Complexity:**

Time complexity for the function find\_minimum: O(*log n*) where *n* is number of elements in array.

Time complexity for the complete program: O(*n*) where *n* is number of elements in array, because size of input is *n*.

Here we are using an iterative approach of binary search. Explanation will be the same as mentioned above for suboptimal\_solution.

**Auxiliary Space Used:**

O(*1*).

As we are using only constant extra space.

**Space Complexity:**

O(*n*) where *n* is number of elements in array.

Input is O(*n*) because we are storing *n* elements of array and auxiliary space used is O(*1*). So, O(*n*) + O(*1*) -> O(*n*).

# Problem 2:

**Hamming Weight**

Calculate Hamming weight of an array of integers.

Hamming weight of an integer is defined as the number of set bits in its binary representation. Hamming weight of an array is a sum of hamming weights of all numbers in it.

**Example**

Input:[1, 2, 3]

Output: 4

Binary representation of 1 is “1”; one set bit.

Binary representation of 2 is “10”; one set bit.

Binary representation of 3 is “11”; two set bits.

1 + 1 + 2 = 4

**Notes**

Input Parameters: Function has one parameter: array of 64-bit integers.

Output: The function returns an integer.

Constraints:

* 1 <= *n* <= 10^5
* 0 <= *s[i]* < 2^32 where 0 <= i < *n*.

# Solution

We provided two solutions.

**1) brute\_force\_solution:**

Per the constraints, all the elements in the array can be stored in a 32-bit integer and hence, to calculate the number of set bits in an integer *x*, we can iterate on all these 32 bits of the corresponding integer and keep a count of the number of set bits.

We repeat the same process for all integers in the given input array *s* and keep the count of total set bits in all integers. To optimize the solution we can break traversal over the bits once we encounter the MSB(Most Significant Bit / Leftmost set bit) in the integer *x*.

**Time Complexity:**

*O(n \* 32)* where *n* is the number of elements in the given input array.

For every integer in the array, we iterate over its 32 bits (constant number of repetitions).

**Auxiliary Space Used:**

*O(1).*

We don’t store any data.

**Space Complexity:**

*O(n)*.

**2) optimal\_solution:**

Our main aim of the solution is to calculate the hamming distance of an integer *x* in constant time with some precomputation.

As the integer size is 32 bit as per the input constraints. So, we can divide the 32-bit integer *x*, into two 16-bits integers.

[31th , 30th , ……. , 17th, 16th] [ 15th , 14th, ………. , 1th , 0th]

A B

Let’s call the first part i.e. from [31th bit to 16th bit] as A and second part from [15th bit to 0th] bit as B. Also, note that both these integers A and B are 16-bit integers. Now, the total set bits in the integer *x* is equal to number of set bits in integer A + number of set bits in integer B and this is the key idea for this solution. Now let *Sz* be the number of all possible 16-bit integers. So, we precompute the number of set bits for all *Sz* integers and store it in memory. We can compute the set bits for all *Sz* integers in linear time. Let’s say dp[i] denotes the number of set bits in integer i. So, we can compute dp[i] using the below state relation :

dp[i] = dp[i >> 1] + (i&1)

In the above relation (i&1) tells if the 0th bit is set in the binary representation of the integer i. To illustrate the above relation, consider the calculation of dp[5].

Now, (5)base10 = (101)base2

So, dp[5] = dp[5 >> 1] + 5&1

Here 5&1 is 1 as the 0th bit is set in binary representation of 5.

As now, we have taken 0th bit of 5 under consideration and hence, now we can right shift the binary representation of 5 by 1 to omit the 0th bit and then calculate the number of set bits in the resulting integer. Also, as right shifting 5 by 1 will result in an integer which is less than 5 and as we are iteratively computing dp states the resultant state dp[5>>1] would have already been computed. Hence, dp[5] = dp[2] + 1 i.e. last bit in 5 plus the number of set bits in 2.

Once, we have precomputed set bits individually for all 16-bit integers. We can answer calculate the number of set bits in a 32-bit integer by two lookups in the precomputed state values.

So, for an integer *x* we first divide it into A and B as explained above and then we do a lookup in our dp[A] and dp[B] to get the set bits in the integer *x*. We repeat the same process for all integers in the array *s* and keep count of the total number of set bits and hence, the hamming weight of the array.

Also, instead of dividing the array into 2 parts, we can divide it into 4 integers each of size 8 bits and proceed the same way as we did in the above explanation. This will reduce the space complexity significantly, but will require 4 lookups and hence will double the previous time complexity. Though, the asymptotics remain the same.

Bonus take away – this is called as the Space-Time trade off.

Kindly, refer to the solution for implementation details.

**Time Complexity:**

*O(n + Sz)* where *n* is number of elements in the given array *s* and *Sz* be the number of all possible 16-bit integers i.e. 2^16.

Precomputing dp state for all 16 bit integers take a linear time O(2^16) as explained above. To calculate the set bits in an integer *x* we are performing 2 iterations. Hence, for all *n* integers the time complexity become O(*2\*n*). Summing up the overall time complexity becomes *O( 2\*n + Sz )* →  *O(n + Sz).*

**Auxiliary Space Used:**

*O(Sz) where Sz* be the number of all possible 16-bit integers i.e. 2^16.

As, we are pre-computing set bits for all 16 bit integers and storing it. Hence the space complexity is *O(Sz).*

**Space Complexity:**

*O(n + Sz)* where *n* is number of elements in the given array *s* and *Sz* be the number of all possible 16-bit integers i.e. 2^16.

For storing input, it will take O(*n*) and as auxiliary space used is O(*Sz*) hence total complexity will be O(*n*) + O(*Sz*) → O(*n + Sz*).

# Problem 3:

**Run Length Encoder**

Compress a string of alphabetic characters with the basic encoding where you simply count the number of repeated characters.

**Example One**

Input: AAAAA

Output: 5A

Character “A” is repeated 5 times consecutively.

**Example Two**

Input: ABaaaBCC

Output: AB3aB2C

Character “a” is repeated 3 times consecutively, character “C” is repeated 2 times consecutively.

Constraints:

* Input string consists of alphabetic characters only
* 1 <= length of the string <= 10^5

# Solution

We provided one solution for this problem, it is optimal.

The problem asks to encode the input string in such a way so that its length remains the same or decreases. To do so, we counted repeated consecutive characters. If the count was more than one, we replaced the repeated portion by a number followed by the character.

**Time complexity:**

O(n) where n denotes the length of the input string.

**Auxiliary space:**

O(1) because no extra memory was used.

**Space complexity:**

O(n).

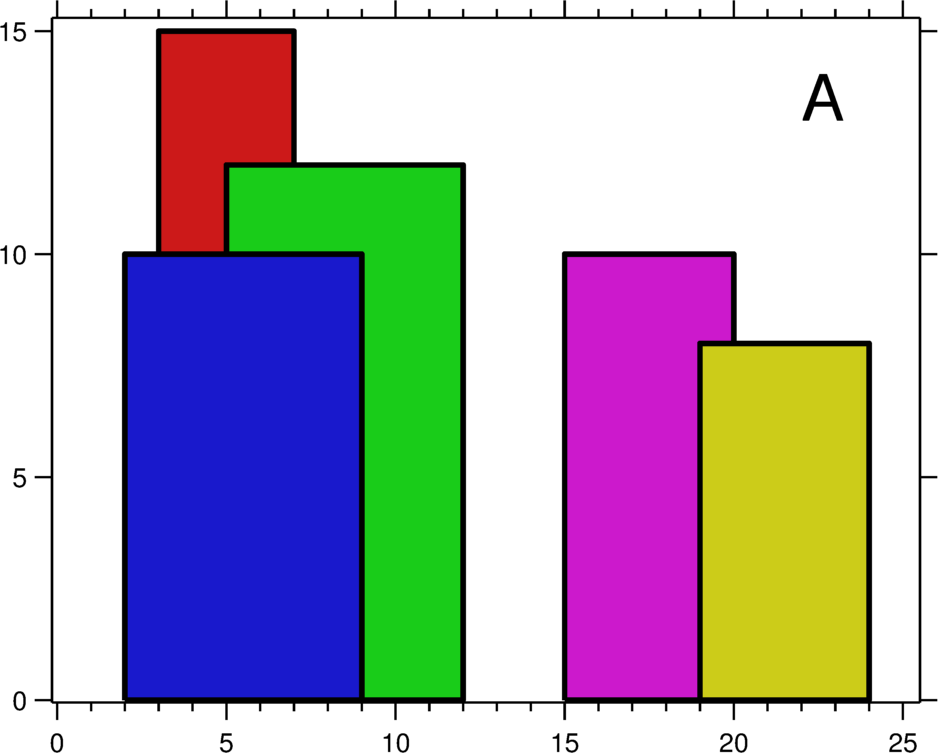
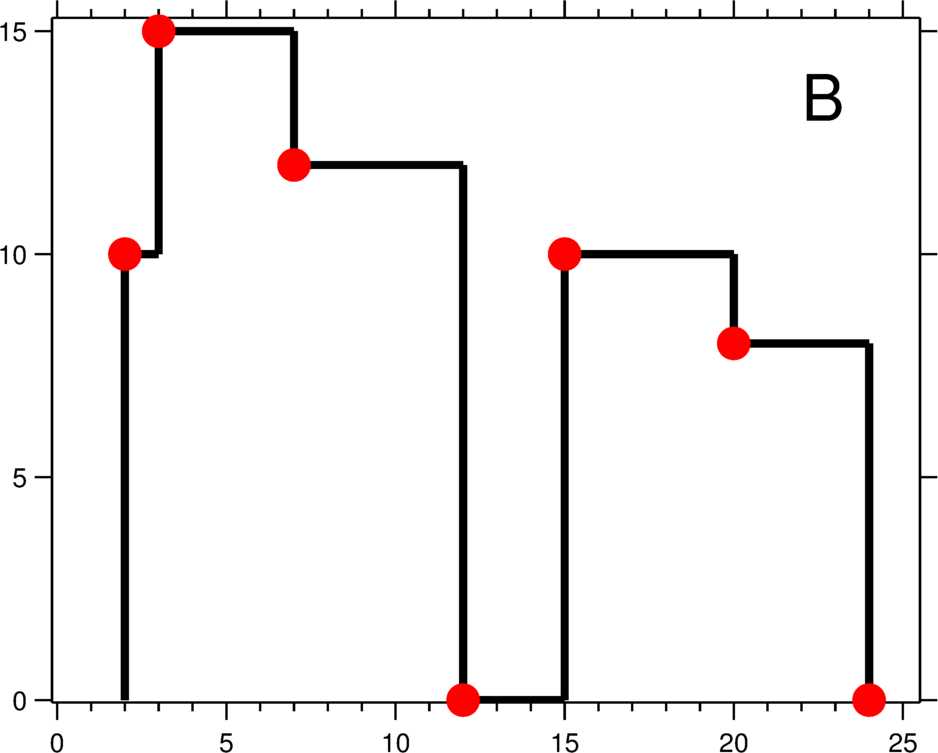
# Problem 4:

**Skyline**

Given n buildings on a two-dimensional plane, find the skyline of these buildings.

Each building on the two-dimensional plane has a start and end x-coordinates, and a y-coordinate height. Skyline is defined as a unique representation of rectangular strips of different heights which are created after the overlap of multiple buildings on a two-dimensional plane.

The following picture demonstrates some buildings on the left side and their skyline on the right.

****

**Example**

Input:

5

2 9 10

3 7 15

5 12 12

15 20 10

19 24 8

**Output:**

2 10

3 15

7 12

12 0

15 10

20 8

24 0

From the image referenced above, we see the blue building at the start and the corresponding red dot in the right image at (2,10). The next change in skyline occurs at an x coordinate of 3 with the red building coming up at the height of 15, so in the output, the next line is printed as 3 15. Similarly, all the buildings are traversed to find the output as given in the sample output section.

Constraints:

* 1 <= *n* <= 10^5
* 1 <= *x, y, height* <= 2 \* 10^9

# Solution

**1) optimal\_solution.java**

A *BuildingIndex* class is defined in the solution consisting of three members:

* *index: it is the value of x-coordinate of the potential skyline point.*
* *startEnd: it is a character that denotes if this particular index is a start point or an end point of the building.*
* *height: it is the height of the building.*

A priority queue named *priorityQ* is created. All the start and end points of all the buildings as objects of *BuildingIndex* class are added in this queue with the following insertion constraints:

* A lower value of *index* gets priority.
* If the *index* values are the same, *'start'* point gets priority.
* If two buildings are starting at the same *index*, then *BuildingIndex* with higher *height* gets priority.
* If two buildings are ending at the same *index*, then *BuildingIndex* with smaller *height* gets priority.

A binary search tree named *heightCountQ* is created which stores the count of heights ordered by the heights of buildings.

Also, a variable named maxHeight is initialized with 0 which stores the current max height of the skyline.

Now pop elements from the *priorityQ*, one by one, till it is empty.

* If the popped element is the starting index, then check if it is greater than *maxHeight*, if so add the start index and height pair to the *ans* list.
* If the popped element is the end index, then check if it was the *maxHeight* till now, if yes, then update the *ans* list with the current index and height from the next tallest building from the *heightCountQ.* And decrease the count in heightCountQ, if the count is 1, then the node is removed from the tree.

**Time Complexity:**

Sorting all the buildings as per the given constraints in the description would take O(2*n*\*log(2*n*)), where *n* is the number of buildings in the given array and 2\**n* is the *BuildingIndex*. Also maintaining the count of heights take O(*n*\*log(*n*)), since each insertion and deletion is O(log(*n*)) in a BST.

So, the total time complexity is O(*n*\*log(*n*)).

**Auxiliary Space Used:**

The *priorityQ* and *heightCount* both require O(*n*) auxiliary space to store *n* buildings. So, total auxiliary space complexity is O(*n*).

**Space Complexity:**

Input and output arrays both require 3\**n* and 2\**n* amount of space respectively. Total space complexity including auxiliary space comes out to be O(*n*).

# Problem 5:

**Find The Next Palindromic Number**

Given an integer *n*, find the smallest palindromic number that’s greater than *n*.

**Example One**

Input: 5

Output: 6

6 is a palindromic number, and it is greater than 5. There is no palindromic number lesser than 6 and greater than 5.

**Example Two**

Input: 10

Output: 11

Constraints:

* 0 <= *n* <= 2147483647

# Solution

If you are getting the wrong answer, make sure to use the appropriate data type. 32-bit integer will overflow for the given constraints.

We provided two solutions, optimal\_solution1.cpp and optimal\_solution2.cpp. We recommend looking at them in this order.

In optimal\_solution1.cpp we can use an integer array instead of strings to simplify some of the steps, but it will take more memory. Integer takes is 4 bytes while char takes 1.

optimal\_solution2.cpp is different from most solutions you will find online. It might take some time to understand initially, but after that in interviews it will be less error prone to write the solution.

For both solutions:

**Time Complexity:**

O(n).

**Auxiliary Space Used**

O(n).

**Space Complexity:**

O(n).

# Problem 6:

**Alternating Positives and Negatives**

Given an *array* of positive and negative integers, rearrange the elements so that the positive and negative numbers alternate. Order of the positive elements should be preserved, same with the negative ones.

Consider zero a positive number for this exercise.

Start output with a positive integer if one exists in the input.

Number of positive and negative integers may not be equal and extra positives or negatives have to appear at the end of the output.

**Example**

Input: [2 3 -4 -9 -1 -7 1 -5 -6]

Output: [2 -4 3 -9 1 -1 -7 -5 -6]

The order of positives in the input: 2, 3, 1.

The order of negatives in the input: -4, -9, -1, -7, -5, -6.

We start with the first positive number, alternate until we run out of (in this case) positives, and dump the remaining negatives at the end of the output.

Output: Return an integer array with alternate positive and negative numbers with order maintained.

Constraints:

* 1 <= *n* <= 500000
* -2 \* 10^9 <= *array[i]* <= 2 \* 10^9

# Solution

We provided two solutions.

**1) optimal\_solution.cpp**

We keep two pointers, one pointing to all positive numbers in the original array and another pointing to all negative numbers in the original array. Let us call them positive\_pointer and negative\_pointer respectively. Then similar to merge sort, we will merge them the numbers pointed by these two pointers in a new array keeping positive and negative numbers alternatively placed to each other in the final array.

Example:

Let the input array be:

5 0 1 -3 4 -6 -8 3 2 -9

We initialise both positive\_pointer and negative\_pointer to 0 (index at the start of the array).

We will run a loop to fill each index of the output array.

We move the pointers forward unless we find the next positive element for the positive\_pointer and next negative element for the negative\_pointer. For our example, the first positive integer is 5 which is at index 0 and the first negative number is -3 which is at index 3.

Now, at the first iteration, since the index of iteration in the loop is 0, and we are starting with the positive element first, we will place the next positive element there which is 5 and move the positive\_pointer ahead by 1 place and go to the next iteration.

Now we repeat the same by finding the next positive and negative elements and continue the process ahead.

Once the value of positive\_pointer or negative\_pointer becomes 10 (size of array), it means that no more positive or negative elements respectively are left. We will keep adding other elements available.

**Time Complexity:**

O(n)

positive\_pointer and negative\_pointer iterate over each element in the array once. O(n) + O(n) -> O(n)

**Auxiliary Space Used:**

O(n)

As we are using an array to store all the resultant elements.

**Space Complexity:**

O(n)

As input is O(n) and auxiliary space used is O(n). O(n) + O(n) -> O(n)

**2) other\_solution.cpp – Didn’t understood this solution**

In this solution, wrong\_index points to the element that shouldn’t be there, either a positive element is at an odd index or a negative element is at an even index. If a wrong\_index is set, it is replaced with an eligible current index whenever possible.

Example:

Let the input array be:

5 0 1 -3 4 -6 -8 3 2 -9

Initially the value of wrong\_index is -1. Then we run a loop for each element in the array.

For each index, we check if the wrong\_index is set and if it is set, we replace it with the next eligible element and push the rest of the elements forward upto the index of the current element and if wrong\_index is not set, we just check if the current element is at wrong\_index.

For index 0, wrong\_index is not set (as wrong\_index equals -1), we check if the element at index 0 is at a wrong position. Since, array[0] which is 5 is positive and expected at even index, we do nothing and simply move forward to analyse the rest of the array.

For index 1, at an odd index we have 0, which is considered positive for this problem, we mark wrong\_index as 1.

Moving forward to index 2, since wrong\_index is set (not equal to -1),  we check if the current element (at index 2) should be at wrong\_index. The current element is positive and we are looking for a negative element for wrong\_index (which is currently 1 and odd).

Now at index 3, wrong\_index is still set and array[3] is -3 which is a negative number. It is the one supposed to be at the wrong index. So we right rotate the subarray from wrong\_index (which is at 1) to the current index in the loop which is at 3.

The rotated array becomes:

5 -3 0 1 4 -6 -8 3 2 -9

After rotation, the element next to wrong\_index would be at the correct position given the fact that it should be of the opposite sign already. So, the next candidate for wrong\_index is 2 steps from its current value. That value is at wrong\_index if the current index in the loop is more than 2 steps away as they all were of the same sign and couldn’t replace the element at wrong\_index. If the gap is not more than 2, we just move ahead, unset wrong\_index and then check if the element at current\_index is at the wrong index and keep repeating this entire procedure for all the indexes in the array.

**Time Complexity:**

O(n^2)

wrong\_index continuously keeps moving forward linearly. For each pair of wrong\_index and right\_index, the rotation takes time proportional to the elements between the pair of indices. n such pairs and n time to rotate each pair in worst case.

**Auxiliary Space Used:**

O(1)

As we are using constant space to rotate the array and store the variables and no extra space is used throughout. To solve this problem with constant auxiliary space, a minimum time complexity of O(n^2) is required.

**Space Complexity:**

O(n)

As input is O(n) and auxiliary space used is O(1). O(n) + O(1) -> O(n)