**Codility**

**Iterations**

BinaryGap

A *binary gap* within a positive integer N is any maximal sequence of consecutive zeros that is surrounded by ones at both ends in the binary representation of N.

For example, number 9 has binary representation 1001 and contains a binary gap of length 2. The number 529 has binary representation 1000010001 and contains two binary gaps: one of length 4 and one of length 3. The number 20 has binary representation 10100 and contains one binary gap of length 1. The number 15 has binary representation 1111 and has no binary gaps. The number 32 has binary representation 100000 and has no binary gaps.

Write a function:

class Solution { public int solution(int N); }

that, given a positive integer N, returns the length of its longest binary gap. The function should return 0 if N doesn't contain a binary gap.

For example, given N = 1041 the function should return 5, because N has binary representation 10000010001 and so its longest binary gap is of length 5. Given N = 32 the function should return 0, because N has binary representation '100000' and thus no binary gaps.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..2,147,483,647].

**static** **int** solution(**int** N) {

String[] binary = Integer.*toBinaryString*(N).split("");

**int** maxCount = 0;

**for** (**int** j = binary.length - 1; j >= 0; j--) {

**if** (binary[j].equals("0")) {

**int** ctr = 0;

**while** (binary[j].equals("0")) {

ctr++;

j--;

}

maxCount = Math.*max*(maxCount, ctr);

}

}

**return** maxCount;

}

**Arrays**

[**CyclicRotation**](https://app.codility.com/programmers/lessons/2-arrays/cyclic_rotation/)

An array A consisting of N integers is given. Rotation of the array means that each element is shifted right by one index, and the last element of the array is moved to the first place. For example, the rotation of array A = [3, 8, 9, 7, 6] is [6, 3, 8, 9, 7] (elements are shifted right by one index and 6 is moved to the first place).

The goal is to rotate array A K times; that is, each element of A will be shifted to the right K times.

Write a function:

class Solution { public int[] solution(int[] A, int K); }

that, given an array A consisting of N integers and an integer K, returns the array A rotated K times.

For example, given

A = [3, 8, 9, 7, 6] K = 3

the function should return [9, 7, 6, 3, 8]. Three rotations were made:

[3, 8, 9, 7, 6] -> [6, 3, 8, 9, 7] [6, 3, 8, 9, 7] -> [7, 6, 3, 8, 9] [7, 6, 3, 8, 9] -> [9, 7, 6, 3, 8]

For another example, given

A = [0, 0, 0] K = 1

the function should return [0, 0, 0]

Given

A = [1, 2, 3, 4] K = 4

the function should return [1, 2, 3, 4]

Assume that:

* N and K are integers within the range [0..100];
* each element of array A is an integer within the range [−1,000..1,000].

In your solution, focus on **correctness**. The performance of your solution will not be the focus of the assessment.

**static** **int**[] solution(**int**[] A, **int** K) {

**int** len=A.length;

**int**[] sol=**new** **int**[len];

**for** (**int** i = 0; i <len; i++) {

sol[(i+K)%len]=A[i];

}

**return** sol;

}

[OddOccurrencesInArray](https://app.codility.com/programmers/lessons/2-arrays/odd_occurrences_in_array/)

A non-empty array A consisting of N integers is given. The array contains an odd number of elements, and each element of the array can be paired with another element that has the same value, except for one element that is left unpaired.

For example, in array A such that:

A[0] = 9 A[1] = 3 A[2] = 9 A[3] = 3 A[4] = 9 A[5] = 7 A[6] = 9

* the elements at indexes 0 and 2 have value 9,
* the elements at indexes 1 and 3 have value 3,
* the elements at indexes 4 and 6 have value 9,
* the element at index 5 has value 7 and is unpaired.

Write a function:

class Solution { public int solution(int[] A); }

that, given an array A consisting of N integers fulfilling the above conditions, returns the value of the unpaired element.

For example, given array A such that:

A[0] = 9 A[1] = 3 A[2] = 9 A[3] = 3 A[4] = 9 A[5] = 7 A[6] = 9

the function should return 7, as explained in the example above.

Write an **efficient** algorithm for the following assumptions:

* N is an odd integer within the range [1..1,000,000];
* each element of array A is an integer within the range [1..1,000,000,000];
* all but one of the values in A occur an even number of times.

**static** **int** solution(**int**[] A) {

Map<Integer, Integer> map = **new** HashMap<Integer, Integer>();

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

**if** (!map.containsKey(A[i]))

map.put(A[i], 1);

**else** **if** (map.get(A[i]) + 1 == 2)

map.remove(A[i]);

**else**

map.put(A[i], map.get(A[i]) + 1);

}

Set<Integer> keySet = map.keySet();

**for** (Integer integer : keySet) {

**return** integer;

}

**return** 0;

}

**Time Complexity**

[FrogJmp](https://app.codility.com/programmers/lessons/3-time_complexity/frog_jmp/)

A small frog wants to get to the other side of the road. The frog is currently located at position X and wants to get to a position greater than or equal to Y. The small frog always jumps a fixed distance, D.

Count the minimal number of jumps that the small frog must perform to reach its target.

Write a function:

class Solution { public int solution(int X, int Y, int D); }

that, given three integers X, Y and D, returns the minimal number of jumps from position X to a position equal to or greater than Y.

For example, given:

X = 10 Y = 85 D = 30

the function should return 3, because the frog will be positioned as follows:

* after the first jump, at position 10 + 30 = 40
* after the second jump, at position 10 + 30 + 30 = 70
* after the third jump, at position 10 + 30 + 30 + 30 = 100

Write an **efficient** algorithm for the following assumptions:

* X, Y and D are integers within the range [1..1,000,000,000];
* X ≤ Y.

**static** **int** solution(**int** X, **int** Y, **int** D) {

**return** ((Y - X) % D == 0) ? (Y - X) / D : ((Y - X) / D + 1);

}

**Counting Elements**

[MissingInteger](https://app.codility.com/programmers/lessons/4-counting_elements/missing_integer/)

This is a demo task.

Write a function:

class Solution { public int solution(int[] A); }

that, given an array A of N integers, returns the smallest positive integer (greater than 0) that does not occur in A.

For example, given A = [1, 3, 6, 4, 1, 2], the function should return 5.

Given A = [1, 2, 3], the function should return 4.

Given A = [−1, −3], the function should return 1.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..100,000];
* each element of array A is an integer within the range [−1,000,000..1,000,000].

**static** **int** solution(**int**[] A) {

Arrays.*sort*(A);

**if**(A[A.length-1]<=0) {

**return** 1;

}**else** {

**for** (**int** i = 0; i < A.length-1; i++) {

**if**(A[i]==A[i+1])**continue**;

**if**(A[i]+1==A[i+1])**continue**;

**else** **return** A[i]+1;

}

**return** A[A.length-1]+1;

}

}

[MinAvgTwoSlice](https://app.codility.com/programmers/lessons/5-prefix_sums/min_avg_two_slice/)

A non-empty array A consisting of N integers is given. A pair of integers (P, Q), such that 0 ≤ P < Q < N, is called a *slice* of array A (notice that the slice contains at least two elements). The *average* of a slice (P, Q) is the sum of A[P] + A[P + 1] + ... + A[Q] divided by the length of the slice. To be precise, the average equals (A[P] + A[P + 1] + ... + A[Q]) / (Q − P + 1).

For example, array A such that:

A[0] = 4 A[1] = 2 A[2] = 2 A[3] = 5 A[4] = 1 A[5] = 5 A[6] = 8

contains the following example slices:

* slice (1, 2), whose average is (2 + 2) / 2 = 2;
* slice (3, 4), whose average is (5 + 1) / 2 = 3;
* slice (1, 4), whose average is (2 + 2 + 5 + 1) / 4 = 2.5.

The goal is to find the starting position of a slice whose average is minimal.

Write a function:

class Solution { public int solution(int[] A); }

that, given a non-empty array A consisting of N integers, returns the starting position of the slice with the minimal average. If there is more than one slice with a minimal average, you should return the smallest starting position of such a slice.

For example, given array A such that:

A[0] = 4 A[1] = 2 A[2] = 2 A[3] = 5 A[4] = 1 A[5] = 5 A[6] = 8

the function should return 1, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [2..100,000];
* each element of array A is an integer within the range [−10,000..10,000].

**public** **static** **int** solution(**int**[] A) {

**int** result = 0;

**int** N = A.length;

**int**[] prefix = **new** **int**[N + 1];

**for** (**int** i = 1; i < prefix.length; i++) {

prefix[i] = prefix[i - 1] + A[i - 1];

}

**double** avg = Double.***MAX\_VALUE***;

**for** (**int** i = 1; i < N; i++) {

**for** (**int** j = i + 1; j <= N; j++) {

**double** temp = (**double**) (prefix[j] - prefix[i - 1]) / (**double**) (j - i + 1);

**if** (temp < avg) {

avg = temp;

result = i - 1;

}

}

}

**return** result;

}

[CountDiv](https://app.codility.com/programmers/lessons/5-prefix_sums/count_div/)

Write a function:

class Solution { public int solution(int A, int B, int K); }

that, given three integers A, B and K, returns the number of integers within the range [A..B] that are divisible by K, i.e.:

{ i : A ≤ i ≤ B, i **mod** K = 0 }

For example, for A = 6, B = 11 and K = 2, your function should return 3, because there are three numbers divisible by 2 within the range [6..11], namely 6, 8 and 10.

Write an **efficient** algorithm for the following assumptions:

* A and B are integers within the range [0..2,000,000,000];
* K is an integer within the range [1..2,000,000,000];
* A ≤ B.

**static** **int** solution(**int** A, **int** B, **int** K) {

**int** n=0;

**for** (**int** i = A; i <= B; i++) {

**if**(i%K==0)n++;

}

**return** n;

}

**Sorting**

[MaxProductOfThree](https://app.codility.com/programmers/lessons/6-sorting/max_product_of_three/)

A non-empty array A consisting of N integers is given. The *product* of triplet (P, Q, R) equates to A[P] \* A[Q] \* A[R] (0 ≤ P < Q < R < N).

For example, array A such that:

A[0] = -3 A[1] = 1 A[2] = 2 A[3] = -2 A[4] = 5 A[5] = 6

contains the following example triplets:

* (0, 1, 2), product is −3 \* 1 \* 2 = −6
* (1, 2, 4), product is 1 \* 2 \* 5 = 10
* (2, 4, 5), product is 2 \* 5 \* 6 = 60

Your goal is to find the maximal product of any triplet.

Write a function:

class Solution { public int solution(int[] A); }

that, given a non-empty array A, returns the value of the maximal product of any triplet.

For example, given array A such that:

A[0] = -3 A[1] = 1 A[2] = 2 A[3] = -2 A[4] = 5 A[5] = 6

the function should return 60, as the product of triplet (2, 4, 5) is maximal.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [3..100,000];
* each element of array A is an integer within the range [−1,000..1,000].

**static** **int** solution(**int**[] A) {

**int** max=0;

**for** (**int** i = 0; i <A.length-2; i++) {

**for** (**int** j = i+1; j < A.length-1; j++) {

**for** (**int** j2 = j+1; j2 < A.length; j2++)

max=Math.*max*(max, (i\*j\*j2));

}

}

**return** max;

}

[Distinct](https://app.codility.com/programmers/lessons/6-sorting/distinct/)

Write a function

class Solution { public int solution(int[] A); }

that, given an array A consisting of N integers, returns the number of distinct values in array A.

For example, given array A consisting of six elements such that:

A[0] = 2 A[1] = 1 A[2] = 1 A[3] = 2 A[4] = 3 A[5] = 1

the function should return 3, because there are 3 distinct values appearing in array A, namely 1, 2 and 3.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [0..100,000];
* each element of array A is an integer within the range [−1,000,000..1,000,000].

**static** **int** solution(**int**[] A) {

**int** len=A.length;

Set<Integer> set=**new** TreeSet<Integer>();

**for** (**int** i = 0; i <len; i++)set.add(A[i]);

**return** set.size();

}

**Stack and Queues**

[Brackets](https://app.codility.com/programmers/lessons/7-stacks_and_queues/brackets/)

A string S consisting of N characters is considered to be *properly nested* if any of the following conditions is true:

* S is empty;
* S has the form "(U)" or "[U]" or "{U}" where U is a properly nested string;
* S has the form "VW" where V and W are properly nested strings.

For example, the string "{[()()]}" is properly nested but "([)()]" is not.

Write a function:

class Solution { public int solution(String S); }

that, given a string S consisting of N characters, returns 1 if S is properly nested and 0 otherwise.

For example, given S = "{[()()]}", the function should return 1 and given S = "([)()]", the function should return 0, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [0..200,000];
* string S consists only of the following characters: "(", "{", "[", "]", "}" and/or ")".

**static** **int** areParanthesisBalanced(String expr) {

**char** ar[] = expr.toCharArray();

Stack<Character> st = **new** Stack();

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

**if** (ar[i] == '{' || ar[i] == '[' || ar[i] == '(') {

st.push(ar[i]);

**continue**;

}

**if** (st.isEmpty()) **return** **0**;

**switch** (ar[i]) {

**case** ')':

**if** (st.peek() == '(') {

st.pop();

**break**;

} **else** {

**return** **0**;

}

**case** ']':

**if** (st.peek() == '[') {

st.pop();

**break**;

} **else** {

**return** **0**;

}

**case** '}':

**if** (st.peek() == '{') {

st.pop();

**break**;

} **else** {

**return** **0**;

}

**default**:

**continue**;

}

}

**return** st.isEmpty()?1:0;

}

[Fish](https://app.codility.com/programmers/lessons/7-stacks_and_queues/fish/)

You are given two non-empty arrays A and B consisting of N integers. Arrays A and B represent N voracious fish in a river, ordered downstream along the flow of the river.

The fish are numbered from 0 to N − 1. If P and Q are two fish and P < Q, then fish P is initially upstream of fish Q. Initially, each fish has a unique position.

Fish number P is represented by A[P] and B[P]. Array A contains the sizes of the fish. All its elements are unique. Array B contains the directions of the fish. It contains only 0s and/or 1s, where:

* 0 represents a fish flowing upstream,
* 1 represents a fish flowing downstream.

If two fish move in opposite directions and there are no other (living) fish between them, they will eventually meet each other. Then only one fish can stay alive − the larger fish eats the smaller one. More precisely, we say that two fish P and Q meet each other when P < Q, B[P] = 1 and B[Q] = 0, and there are no living fish between them. After they meet:

* If A[P] > A[Q] then P eats Q, and P will still be flowing downstream,
* If A[Q] > A[P] then Q eats P, and Q will still be flowing upstream.

We assume that all the fish are flowing at the same speed. That is, fish moving in the same direction never meet. The goal is to calculate the number of fish that will stay alive.

For example, consider arrays A and B such that:

A[0] = 4 B[0] = 0 A[1] = 3 B[1] = 1 A[2] = 2 B[2] = 0 A[3] = 1 B[3] = 0 A[4] = 5 B[4] = 0

Initially all the fish are alive and all except fish number 1 are moving upstream. Fish number 1 meets fish number 2 and eats it, then it meets fish number 3 and eats it too. Finally, it meets fish number 4 and is eaten by it. The remaining two fish, number 0 and 4, never meet and therefore stay alive.

Write a function:

class Solution { public int solution(int[] A, int[] B); }

that, given two non-empty arrays A and B consisting of N integers, returns the number of fish that will stay alive.

For example, given the arrays shown above, the function should return 2, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..100,000];
* each element of array A is an integer within the range [0..1,000,000,000];
* each element of array B is an integer that can have one of the following values: 0, 1;
* the elements of A are all distinct.

**public** **static** **int** solution(**int**[] A, **int**[] B) {

Stack<Integer> s = **new** Stack<Integer>();

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

**int** size = A[i], dir = B[i];

**if** (s.empty()) s.push(i);

**else** {

**while** (!s.empty() && dir - B[s.peek()] == -1 && A[s.peek()] < size) {

s.pop();

}

**if** (!s.empty()) {

**if** (dir - B[s.peek()] != -1)

s.push(i);

} **else**

s.push(i);

}

}

**return** s.size();

}

**Leader**

[Dominator](https://app.codility.com/programmers/lessons/8-leader/dominator/)

An array A consisting of N integers is given. The *dominator* of array A is the value that occurs in more than half of the elements of A.

For example, consider array A such that

A[0] = 3 A[1] = 4 A[2] = 3 A[3] = 2 A[4] = 3 A[5] = -1 A[6] = 3 A[7] = 3

The dominator of A is 3 because it occurs in 5 out of 8 elements of A (namely in those with indices 0, 2, 4, 6 and 7) and 5 is more than a half of 8.

Write a function

class Solution { public int solution(int[] A); }

that, given an array A consisting of N integers, returns index of any element of array A in which the dominator of A occurs. The function should return −1 if array A does not have a dominator.

For example, given array A such that

A[0] = 3 A[1] = 4 A[2] = 3 A[3] = 2 A[4] = 3 A[5] = -1 A[6] = 3 A[7] = 3

the function may return 0, 2, 4, 6 or 7, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [0..100,000];
* each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].

**public** **static** **int** solution(**int**[] A) {

Map<Integer, Integer> D = **new** HashMap<Integer, Integer>();

**for** (**int** a : A) {

**if** (D.containsKey(a)) {

D.put(a, D.get(a) + 1);

} **else** {

D.put(a, 1);

}

}

**int** n = A.length;

**for** (Map.Entry<Integer, Integer> d : D.entrySet()) {

**if** (d.getValue() > n / 2) {

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

**if** (A[i] == d.getKey()) {

**return** i;

}

}

}

}

**return** -1;

}

**Maximum Slice Problem(Kadane’s Algorithm)**

[MaxDoubleSliceSum](https://app.codility.com/programmers/lessons/9-maximum_slice_problem/max_double_slice_sum/)

A non-empty array A consisting of N integers is given.

A triplet (X, Y, Z), such that 0 ≤ X < Y < Z < N, is called a *double slice*.

The *sum* of double slice (X, Y, Z) is the total of A[X + 1] + A[X + 2] + ... + A[Y − 1] + A[Y + 1] + A[Y + 2] + ... + A[Z − 1].

For example, array A such that:

A[0] = 3 A[1] = 2 A[2] = 6 A[3] = -1 A[4] = 4 A[5] = 5 A[6] = -1 A[7] = 2

contains the following example double slices:

* double slice (0, 3, 6), sum is 2 + 6 + 4 + 5 = 17,
* double slice (0, 3, 7), sum is 2 + 6 + 4 + 5 − 1 = 16,
* double slice (3, 4, 5), sum is 0.

The goal is to find the maximal sum of any double slice.

Write a function:

class Solution { public int solution(int[] A); }

that, given a non-empty array A consisting of N integers, returns the maximal sum of any double slice.

For example, given:

A[0] = 3 A[1] = 2 A[2] = 6 A[3] = -1 A[4] = 4 A[5] = 5 A[6] = -1 A[7] = 2

the function should return 17, because no double slice of array A has a sum of greater than 17.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [3..100,000];
* each element of array A is an integer within the range [−10,000..10,000].

**public** **static** **int** solution (**int**[] A) {

**int**[] k1 = **new** **int**[A.length];

**int**[] k2 = **new** **int**[A.length];

**int** max = 0, arlen = A.length - 1;

**for** (**int** i = 1; i < arlen; i++) {

k1[i] = Math.*max*(k1[i - 1] + A[i], 0);

}

**for** (**int** i = arlen - 1; i > 0; i--) {

k2[i] = Math.*max*(k2[i + 1] + A[i], 0);

}

**for** (**int** i = 1; i < k1.length - 1; i++) {

max = Math.*max*(k1[i - 1] + k2[i + 1], max);

}

**return** max;

}

[MaxSliceSum](https://app.codility.com/programmers/lessons/9-maximum_slice_problem/max_slice_sum/)

A non-empty array A consisting of N integers is given. A pair of integers (P, Q), such that 0 ≤ P ≤ Q < N, is called a *slice* of array A. The *sum* of a slice (P, Q) is the total of A[P] + A[P+1] + ... + A[Q].

Write a function:

class Solution { public int solution(int[] A); }

that, given an array A consisting of N integers, returns the maximum sum of any slice of A.

For example, given array A such that:

A[0] = 3 A[1] = 2 A[2] = -6 A[3] = 4 A[4] = 0

the function should return 5 because:

* (3, 4) is a slice of A that has sum 4,
* (2, 2) is a slice of A that has sum −6,
* (0, 1) is a slice of A that has sum 5,
* no other slice of A has sum greater than (0, 1).

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..1,000,000];
* each element of array A is an integer within the range [−1,000,000..1,000,000];
* the result will be an integer within the range [−2,147,483,648..2,147,483,647].

**public** **static** **int** maxSliceAlgoSingle(**int**[] A) {

**int** max = A[0], maxSoFar = A[0];

**for** (**int** i = 1; i < A.length; i++) {

maxSoFar = Math.*max*(A[i], A[i] + maxSoFar);

max = Math.*max*(max, maxSoFar);

}

**return** max;

}

**Prime and Composite Factors**

[CountFactors](https://app.codility.com/programmers/lessons/10-prime_and_composite_numbers/count_factors/)

A positive integer D is a *factor* of a positive integer N if there exists an integer M such that N = D \* M.

For example, 6 is a factor of 24, because M = 4 satisfies the above condition (24 = 6 \* 4).

Write a function:

class Solution { public int solution(int N); }

that, given a positive integer N, returns the number of its factors.

For example, given N = 24, the function should return 8, because 24 has 8 factors, namely 1, 2, 3, 4, 6, 8, 12, 24. There are no other factors of 24.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..2,147,483,647].

**public** **static** **int** solution(**int** N) {

**int** cnt=0;

**for** (**int** i = 1; i < N+1; i++) **if**(N%i==0) cnt++;

**return** cnt;

}

**SE Alogorithm**

[CountNonDivisible](https://app.codility.com/programmers/lessons/11-sieve_of_eratosthenes/count_non_divisible/)

You are given an array A consisting of N integers.

For each number A[i] such that 0 ≤ i < N, we want to count the number of elements of the array that are not the divisors of A[i]. We say that these elements are non-divisors.

For example, consider integer N = 5 and array A such that:

A[0] = 3 A[1] = 1 A[2] = 2 A[3] = 3 A[4] = 6

For the following elements:

* A[0] = 3, the non-divisors are: 2, 6,
* A[1] = 1, the non-divisors are: 3, 2, 3, 6,
* A[2] = 2, the non-divisors are: 3, 3, 6,
* A[3] = 3, the non-divisors are: 2, 6,
* A[4] = 6, there aren't any non-divisors.

Write a function:

class Solution { public int[] solution(int[] A); }

that, given an array A consisting of N integers, returns a sequence of integers representing the amount of non-divisors.

Result array should be returned as an array of integers.

For example, given:

A[0] = 3 A[1] = 1 A[2] = 2 A[3] = 3 A[4] = 6

the function should return [2, 4, 3, 2, 0], as explained above.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..50,000];
* each element of array A is an integer within the range [1..2 \* N].

**public** **int**[] solution(**int**[] A) {

List list = **new** LinkedList();

**int** count = 0;

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

**for** (**int** j = 0; j < A.length; j++)

**if** (A[i] % A[j] != 0) {

count++;

}

list.add(count);

count = 0;

}

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

A[i] = (**int**) list.get(i);

}

**return** A;

}

**Euclidean Algorithm**

[ChocolatesByNumbers](https://app.codility.com/programmers/lessons/12-euclidean_algorithm/chocolates_by_numbers/)

Two positive integers N and M are given. Integer N represents the number of chocolates arranged in a circle, numbered from 0 to N − 1.

You start to eat the chocolates. After eating a chocolate you leave only a wrapper.

You begin with eating chocolate number 0. Then you omit the next M − 1 chocolates or wrappers on the circle, and eat the following one.

More precisely, if you ate chocolate number X, then you will next eat the chocolate with number (X + M) modulo N (remainder of division).

You stop eating when you encounter an empty wrapper.

For example, given integers N = 10 and M = 4. You will eat the following chocolates: 0, 4, 8, 2, 6.

The goal is to count the number of chocolates that you will eat, following the above rules.

Write a function:

class Solution { public int solution(int N, int M); }

that, given two positive integers N and M, returns the number of chocolates that you will eat.

For example, given integers N = 10 and M = 4. the function should return 5, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N and M are integers within the range [1..1,000,000,000].

**public** **static** **int** solution(**int** N, **int** M) {

**int** counter = 1;

**int** start = 0;

**int** value;

**while** ((start + M) % N != 0) {

value = (start + M) % N;

start = value;

counter++;

}

**return** counter;

}

[CommonPrimeDivisors](https://app.codility.com/programmers/lessons/12-euclidean_algorithm/common_prime_divisors/)

A *prime* is a positive integer X that has exactly two distinct divisors: 1 and X. The first few prime integers are 2, 3, 5, 7, 11 and 13.

A prime D is called a *prime divisor* of a positive integer P if there exists a positive integer K such that D \* K = P. For example, 2 and 5 are prime divisors of 20.

You are given two positive integers N and M. The goal is to check whether the sets of prime divisors of integers N and M are exactly the same.

For example, given:

* N = 15 and M = 75, the prime divisors are the same: {3, 5};
* N = 10 and M = 30, the prime divisors aren't the same: {2, 5} is not equal to {2, 3, 5};
* N = 9 and M = 5, the prime divisors aren't the same: {3} is not equal to {5}.

Write a function:

class Solution { public int solution(int[] A, int[] B); }

that, given two non-empty arrays A and B of Z integers, returns the number of positions K for which the prime divisors of A[K] and B[K] are exactly the same.

For example, given:

A[0] = 15 B[0] = 75 A[1] = 10 B[1] = 30 A[2] = 3 B[2] = 5

the function should return 1, because only one pair (15, 75) has the same set of prime divisors.

Write an **efficient** algorithm for the following assumptions:

* Z is an integer within the range [1..6,000];
* each element of arrays A, B is an integer within the range [1..2,147,483,647].

**public** **int** solution(**int**[] A, **int**[] B) {

**int** count = 0;

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

**if** (hasSamePrimeDivisors(A[i], B[i])) {

count++;

}

}

**return** count;

}

**public** **int** gcd(**int** a, **int** b) {

**if** (a % b == 0) **return** b;

**return** gcd(b, a % b);

}

**public** **boolean** hasSamePrimeDivisors(**int** a, **int** b) {

**int** gcdValue = gcd(a, b);

**int** gcdA, gcdB;

**while** (a != 1) {

gcdA = gcd(a, gcdValue);

**if** (gcdA == 1) **break**;

a = a / gcdA;

}

**if** (a != 1) **return** **false**;

**while** (b != 1) {

gcdB = gcd(b, gcdValue);

**if** (gcdB == 1) {

**break**;

}

b = b / gcdB;

}

**return** b == 1;

}

**Fibonacci Problem**

[Ladder](https://app.codility.com/programmers/lessons/13-fibonacci_numbers/ladder/)

You have to climb up a ladder. The ladder has exactly N rungs, numbered from 1 to N. With each step, you can ascend by one or two rungs. More precisely:

* with your first step you can stand on rung 1 or 2,
* if you are on rung K, you can move to rungs K + 1 or K + 2,
* finally you have to stand on rung N.

Your task is to count the number of different ways of climbing to the top of the ladder.

For example, given N = 4, you have five different ways of climbing, ascending by:

* 1, 1, 1 and 1 rung,
* 1, 1 and 2 rungs,
* 1, 2 and 1 rung,
* 2, 1 and 1 rungs, and
* 2 and 2 rungs.

Given N = 5, you have eight different ways of climbing, ascending by:

* 1, 1, 1, 1 and 1 rung,
* 1, 1, 1 and 2 rungs,
* 1, 1, 2 and 1 rung,
* 1, 2, 1 and 1 rung,
* 1, 2 and 2 rungs,
* 2, 1, 1 and 1 rungs,
* 2, 1 and 2 rungs, and
* 2, 2 and 1 rung.

The number of different ways can be very large, so it is sufficient to return the result modulo 2P, for a given integer P.

Write a function:

class Solution { public int[] solution(int[] A, int[] B); }

that, given two non-empty arrays A and B of L integers, returns an array consisting of L integers specifying the consecutive answers; position I should contain the number of different ways of climbing the ladder with A[I] rungs modulo 2B[I].

For example, given L = 5 and:

A[0] = 4 B[0] = 3 A[1] = 4 B[1] = 2 A[2] = 5 B[2] = 4 A[3] = 5 B[3] = 3 A[4] = 1 B[4] = 1

the function should return the sequence [5, 1, 8, 0, 1], as explained above.

Write an **efficient** algorithm for the following assumptions:

* L is an integer within the range [1..50,000];
* each element of array A is an integer within the range [1..L];
* each element of array B is an integer within the range [1..30].

**public** **static** **int**[] solution1(**int**[] A, **int**[] B) {

**int** len = A.length;

**int**[] ladder = **new** **int**[len];

**if** (A != **null** && B != **null** && (A.length == B.length)) {

**int** fib[] = **new** **int**[len + 1];

fib[0] = 1; fib[1] = 1;

**for** (**int** i = 2; i < len + 1; ++i) fib[i] = (fib[i - 1] + fib[i - 2]) % (1 << 30);

**for** (**int** i = 0; i < len; ++i) ladder[i] = fib[A[i]] % (1 << B[i]);

} **return** ladder; }

**Binary Search Algorithm**

[MinMaxDivision](https://app.codility.com/programmers/lessons/14-binary_search_algorithm/min_max_division/)

You are given integers K, M and a non-empty array A consisting of N integers. Every element of the array is not greater than M.

You should divide this array into K blocks of consecutive elements. The size of the block is any integer between 0 and N. Every element of the array should belong to some block.

The sum of the block from X to Y equals A[X] + A[X + 1] + ... + A[Y]. The sum of empty block equals 0.

The *large sum* is the maximal sum of any block.

For example, you are given integers K = 3, M = 5 and array A such that:

A[0] = 2 A[1] = 1 A[2] = 5 A[3] = 1 A[4] = 2 A[5] = 2 A[6] = 2

The array can be divided, for example, into the following blocks:

* [2, 1, 5, 1, 2, 2, 2], [], [] with a large sum of 15;
* [2], [1, 5, 1, 2], [2, 2] with a large sum of 9;
* [2, 1, 5], [], [1, 2, 2, 2] with a large sum of 8;
* [2, 1], [5, 1], [2, 2, 2] with a large sum of 6.

The goal is to minimize the large sum. In the above example, 6 is the minimal large sum.

Write a function:

class Solution { public int solution(int K, int M, int[] A); }

that, given integers K, M and a non-empty array A consisting of N integers, returns the minimal large sum.

For example, given K = 3, M = 5 and array A such that:

A[0] = 2 A[1] = 1 A[2] = 5 A[3] = 1 A[4] = 2 A[5] = 2 A[6] = 2

the function should return 6, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N and K are integers within the range [1..100,000];
* M is an integer within the range [0..10,000];
* each element of array A is an integer within the range [0..M].

**public** **static** **int** solution(**int** K, **int** M, **int**[] A) {

**int** min = 0;

**int** max = 0;

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

max += A[i];

min = Math.*max*(min, A[i]);

}

**int** result = max;

**while** (min <= max) {

**int** mid = (min + max) / 2;

**if** (*division*(A, mid, K - 1)) {

max = mid - 1;

result = mid;

} **else** {

min = mid + 1;

}

}

**return** result;

}

**private** **static** **boolean** division(**int**[] A, **int** mid, **int** k) {

**int** sum = 0;

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

sum += A[i];

**if** (sum > mid) {

sum = A[i];

k--;

}

**if** (k < 0) {

**return** **false**;

}

}

**return** **true**;

}

**CaterPiller Method**

[AbsDistinct](https://app.codility.com/programmers/lessons/15-caterpillar_method/abs_distinct/)

A non-empty array A consisting of N numbers is given. The array is sorted in non-decreasing order. The *absolute distinct count* of this array is the number of distinct absolute values among the elements of the array.

For example, consider array A such that:

A[0] = -5 A[1] = -3 A[2] = -1 A[3] = 0 A[4] = 3 A[5] = 6

The absolute distinct count of this array is 5, because there are 5 distinct absolute values among the elements of this array, namely 0, 1, 3, 5 and 6.

Write a function:

class Solution { public int solution(int[] A); }

that, given a non-empty array A consisting of N numbers, returns absolute distinct count of array A.

For example, given array A such that:

A[0] = -5 A[1] = -3 A[2] = -1 A[3] = 0 A[4] = 3 A[5] = 6

the function should return 5, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..100,000];
* each element of array A is an integer within the range [−2,147,483,648..2,147,483,647];
* array A is sorted in non-decreasing order.

**public** **static** **int** solution(**int** A[], **int** N) {

Set<Integer> set = **new** HashSet<>();

**for** (**int** i = 0; i < N; i++) {

set.add(Math.*abs*(A[i]));

}

**return** set.size();

}

[MinAbsSumOfTwo](https://app.codility.com/programmers/lessons/15-caterpillar_method/min_abs_sum_of_two/)

Let A be a non-empty array consisting of N integers.

The *abs sum of two* for a pair of indices (P, Q) is the absolute value |A[P] + A[Q]|, for 0 ≤ P ≤ Q < N.

For example, the following array A:

A[0] = 1 A[1] = 4 A[2] = -3

has pairs of indices (0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2).   
The abs sum of two for the pair (0, 0) is A[0] + A[0] = |1 + 1| = 2.   
The abs sum of two for the pair (0, 1) is A[0] + A[1] = |1 + 4| = 5.   
The abs sum of two for the pair (0, 2) is A[0] + A[2] = |1 + (−3)| = 2.   
The abs sum of two for the pair (1, 1) is A[1] + A[1] = |4 + 4| = 8.   
The abs sum of two for the pair (1, 2) is A[1] + A[2] = |4 + (−3)| = 1.   
The abs sum of two for the pair (2, 2) is A[2] + A[2] = |(−3) + (−3)| = 6.

Write a function:

class Solution { public int solution(int[] A); }

that, given a non-empty array A consisting of N integers, returns the minimal abs sum of two for any pair of indices in this array.

For example, given the following array A:

A[0] = 1 A[1] = 4 A[2] = -3

the function should return 1, as explained above.

Given array A:

A[0] = -8 A[1] = 4 A[2] = 5 A[3] =-10 A[4] = 3

the function should return |(−8) + 5| = 3.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..100,000];
* each element of array A is an integer within the range [−1,000,000,000..1,000,000,000].

**static** **int** solution(**int**[] a) {

**int** num = Math.*abs*(a[0] + a[0]);

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

**for** (**int** j = i; j < a.length; j++) {

**if** (num > (Math.*abs*(a[i] + a[j]))) {

num = Math.*abs*(a[i] + a[j]);

}

}

}

**return** num;

}

**Greedy Algorithm**

[TieRopes](https://app.codility.com/programmers/lessons/16-greedy_algorithms/tie_ropes/)

There are N ropes numbered from 0 to N − 1, whose lengths are given in an array A, lying on the floor in a line. For each I (0 ≤ I < N), the length of rope I on the line is A[I].

We say that two ropes I and I + 1 are *adjacent*. Two adjacent ropes can be tied together with a knot, and the length of the tied rope is the sum of lengths of both ropes. The resulting new rope can then be tied again.

For a given integer K, the goal is to tie the ropes in such a way that the number of ropes whose length is greater than or equal to K is maximal.

For example, consider K = 4 and array A such that:

A[0] = 1 A[1] = 2 A[2] = 3 A[3] = 4 A[4] = 1 A[5] = 1 A[6] = 3

The ropes are shown in the figure below.

We can tie:

* rope 1 with rope 2 to produce a rope of length A[1] + A[2] = 5;
* rope 4 with rope 5 with rope 6 to produce a rope of length A[4] + A[5] + A[6] = 5.

After that, there will be three ropes whose lengths are greater than or equal to K = 4. It is not possible to produce four such ropes.

Write a function:

class Solution { public int solution(int K, int[] A); }

that, given an integer K and a non-empty array A of N integers, returns the maximum number of ropes of length greater than or equal to K that can be created.

For example, given K = 4 and array A such that:

A[0] = 1 A[1] = 2 A[2] = 3 A[3] = 4 A[4] = 1 A[5] = 1 A[6] = 3

the function should return 3, as explained above.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [1..100,000];
* K is an integer within the range [1..1,000,000,000];
* each element of array A is an integer within the range [1..1,000,000,000].

**public** **static** **int** solution(**int** K, **int**[] A) {

**int** result = 0;

**int** previousValue = 0;

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

**if** (A[i] >= K) {

result++;

} **else** {

previousValue = previousValue + A[i];

**if** (previousValue >= K) {

result++;

previousValue = 0;

}

}

}

**return** result;

}

**Dynamic Programming**

[MinAbsSum](https://app.codility.com/programmers/lessons/17-dynamic_programming/min_abs_sum/)

For a given array A of N integers and a sequence S of N integers from the set {−1, 1}, we define val(A, S) as follows:

val(A, S) = |**sum**{ A[i]\*S[i] for i = 0..N−1 }|

(Assume that the sum of zero elements equals zero.)

For a given array A, we are looking for such a sequence S that minimizes val(A,S).

Write a function:

class Solution { public int solution(int[] A); }

that, given an array A of N integers, computes the minimum value of val(A,S) from all possible values of val(A,S) for all possible sequences S of N integers from the set {−1, 1}.

For example, given array:

A[0] = 1 A[1] = 5 A[2] = 2 A[3] = -2

your function should return 0, since for S = [−1, 1, −1, 1], val(A, S) = 0, which is the minimum possible value.

Write an **efficient** algorithm for the following assumptions:

* N is an integer within the range [0..20,000];
* each element of array A is an integer within the range [−100..100].

**Singly Linked List**

**Condensed List**

import java.util.Scanner;

class Node {

public int data;

public Node next;

public void displayNodeData() {

System.out.println("{ " + data + " } ");

}

}

class SinglyLinkedList {

private Node head;

public boolean isEmpty() {

return (head == null);

}

public boolean checkList(int data) {

boolean flag = true;

System.out.println("checking list element");

Node firstnode = head;

while (firstnode != null) {

int d = firstnode.data;

firstnode = firstnode.next;

if (d == data) {

flag = false;

}

}

return flag;

}

public void insertLast(int data) {

Node current = head;

if (head == null) {

Node newNode = new Node();

newNode.data = data;

newNode.next = null;

head = newNode;

} else {

while (current.next != null) {

current = current.next; // we'll loop until current.next is null

}

if (checkList(data)) {

Node newNode = new Node();

newNode.data = data;

current.next = newNode;

}

}

}

// For printing Linked List

public void printLinkedList() {

System.out.println("Printing LinkedList (head --> last) ");

Node current = head;

while (current != null) {

current.displayNodeData();

current = current.next;

}

// System.out.println();

}

}

public class SinglyLinkedlIstDemo {

public static void main(String args[]) {

SinglyLinkedList myLinkedlist = new SinglyLinkedList();

Scanner sc = new Scanner(System.in);

int size = sc.nextInt();

for (int i = 0; i < size; i++) {

myLinkedlist.insertLast(sc.nextInt());

}

myLinkedlist.printLinkedList();

// myLinkedlist.insertLast(2);

// myLinkedlist.insertLast(4);

// myLinkedlist.printLinkedList();

}

}

**Insert a Node (Before,After),Delete a Node**

class Node {

int data;

Node next;

void displayData()

{

System.out.println("[" +data +"]");

}

}

class SinglyLinkedList1 {

private Node head;

public void insertAfter(int data)// 7

{

Node current=head;//5,null

while(current.next!=null)

{

current=current.next;

}

Node newNode = new Node();

newNode.data = data;//6

current.next = newNode;

}

public void insertFirst(int data)

{

Node newNode = new Node();

newNode.data = data;//7

newNode.next = head;

head = newNode;

}

public void deleteNode(Node after)

{

Node temp =head;

while(temp.next!=null && temp.data!=after.data)

{

temp=temp.next;

}

if(temp.next!=null)

{

temp.next=temp.next.next;

}

}

public void printList()

{

System.out.println("Priniting List Elements");

Node firstnode =head;

while(firstnode!=null)

{

firstnode.displayData();

firstnode=firstnode.next;

}

}

}

class LinkedListDemo {

public static void main(String args[]) {

SinglyLinkedList1 sll = new SinglyLinkedList1();

sll.insertFirst(5);

sll.printList();

sll.insertAfter(6);

sll.printList();

sll.insertAfter(7);

sll.printList();

Node node=new Node();

node.data=7;

sll.deleteNode(node);

sll.printList();

}

}

**Hackerrank**

Picking Tickets

**Example**

*tickets = [8, 5, 4, 8, 4]*

Valid subsequences, sorted, are *{4, 4, 5}* and *{8, 8}.* These subsequences have *m* values of *3* and *2*, respectively. Return *3*.

**static** **int** takingTicket(**int**[] a) {

Arrays.*sort*(a);

**int** curCount = 0, max = 0;

**for** (**int** i = 0; i < a.length - 1; i++) {

**if** (Math.*abs*(a[i + 1] - a[i])<= 1) {

curCount++;

max = Math.*max*(max, curCount);

} **else**

curCount = 0;

}

**return** max + 1;

}

Word Compression

**Example**

*word = "abbcccb"*

*k = 3*

* Remove *k = 3* characters  '*c',* now *word = "abbb"*.
* Remove 3 characters '*b'*, so the final word is*"a"*.

**public** **static** String compressWord(String word, **int** k) {

StringBuffer s = **new** StringBuffer(word);

**char** prev;

Map<Character, Integer> m = **new** HashMap();

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

**if** (i == 0) {

prev = s.charAt(i);

**continue**;

} **else** {

m.put(s.charAt(i), m.get(s.charAt(i)) == **null** ? 1 : m.get(s.charAt(i)) + 1);

**if** (m.get(s.charAt(i)) == k) {

s.deleteCharAt(i);

s.deleteCharAt(i - 1);

s.deleteCharAt(i - 2);

**return** *compressWord*(s.toString(), k);

}

}

}

**return** s.toString();

}

How many flips

**Example:**

*target = 01011*

Start with a string of *5* zeros, the same length string as the*target.*

*Initial String -> 00000*

*Flip the 3rd digit -> 00111*

*Flip the 2nd digit -> 01000*

*Flip the 4th digit -> 01011*

*3 flips are required to reach the target.* The return value is*3.*

**private** **static** **int** minFlips(String target) {

**char** curr='1';

**int** count=0;

StringBuilder sb=**new** StringBuilder("");

**for**(**int** i=0;i<target.length();i++)

{

**if**(target.charAt(i)==curr)

{

count++;

curr=(**char**)(48 +(curr +1)%2);

sb.append(curr);

System.***out***.println(sb.toString());

}

}

**return** count;

}

 Missing Words

**Example**

*s = 'I like cheese'*

*t = 'like'*

Then 'like' is the subsequence, and ['I', 'cheese'] is the list of missing words, in order

**public** **static** LinkedList getMissingString(String str1, String str2) {

LinkedList<String> list1 = **new** LinkedList<String>(Arrays.*asList*(str1.split(" ")));

LinkedList<String> list2 = **new** LinkedList<String>(Arrays.*asList*(str2.split(" ")));

**if**(str1.contains(str2)) **return** **new** LinkedList<String>(Arrays.*asList*(str1.replace(str2, "").trim().split(" ")));

**for** (String s : list2) {

**if** (list1.contains(s)) {

list1.remove(s);

}

}

**return** list1;

}

Valid Email Address

Valid HackerRank emails are of the form user@hackerrank.com, and the characteristics of *user* are:

* It starts with *1* to *6* lowercase English letters denoted by the character class *[a-z]*.
* The lowercase letter(s) are followed by an *optional* underscore, i.e. zero or one occurrence of the *underscore '\_'* character.
* The optional underscore is followed by *0* to *4* *optional* digits denoted by the character class *[0-9]*.

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter a emailId: ");

String input = sc.next();

String domain = "hackerrank.com";

System.***out***.println(Pattern.*matches*("^[a-z]{1,6}\_?\\d{0,4}@hackerrank.com", input));

}

Counting Bits

For example, the diagram below depicts this information for the value *n = 37*:

In the binary representation of *37*, there are three *1*-bits located at the respective *1st*, *4th*, and *6th* positions.

**static** **void** getOneBits(**int** N) {

String[] binary = Integer.*toBinaryString*(N).split("");

Set<Integer> set = **new** LinkedHashSet<Integer>();

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

**if** (binary[i].equals("1"))

set.add(i+1);

}

System.***out***.println(set.size());

System.***out***.println(set);

}

Grouping Bits

**Example**

*arr=[0,1,0,1]*

With *1* move, switching elements *1* and *2*, yields *[0,0,1,1],*a sorted array.

**public** **static** **int** solution(**int**[] A) {

**int** one = 0, zero = 0, first = 0, second = 0;

**for** (Integer integer : A) {

**if** (integer == 1) {

one++;

first += zero;

} **else** {

zero++;

second += one;

}

}

**return** Math.*min*(first, second);

}

Cardinality Sorting

**Example**

*n = 4*

*nums = [1, 2, 3, 4]*

The binary cardinality of each element of the array nums:

* (1)10 → (1)2, so *1*'s binary cardinality is *1*.
* (2)10 → (10)2, so *2*'s binary cardinality is *1*.
* (3)10 → (11)2, so *3*'s binary cardinality is *2*.
* (4)10 → (100)2, so *4*'s binary cardinality is *1*.

**public** **static** List<Integer> cardinalitySort(List<Integer> nums) {

Collections.*sort*(nums, **new** Comparator<Integer>() {

@Override

**public** **int** compare(Integer num1, Integer num2) {

**if** (Integer.*bitCount*(num1) < Integer.*bitCount*(num2))

**return** -1;

**else** **if** (Integer.*bitCount*(num1) > Integer.*bitCount*(num2))

**return** 1;

**else** **if** (Integer.*bitCount*(num1) == Integer.*bitCount*(num2))

**return** (num1 < num2) ? -1 : 1;

**else**

**return** 0;

}

});

**return** nums;

}

Rod Cutting

**Example**

*n = 4*

*lengths = [1, 1, 3, 4]*

* The shortest rods are *1* unit long, so discard them and record their length.
* Remove their length, *1* unit, from the longer rods and discard the offcuts.
* Now, there are *2* rods, *lengths = [2, 3]*. Discard the rod of length *2*.
* Cut *2* from the rod length *3,* and discard the offcut.
* Now there is only one rod of length *1*. It is the shortest, so discard it.

Return an array with the number of rods at the start of each turn: *[4, 2, 1]*.

**public** **static** List rodOffcut(List lengths) {

Collections.sort(lengths);

List list = **new** LinkedList();

**while** (lengths.size() != 0) {

list.add(lengths.size());

lengths = *cutLength*(lengths);

}

**return** list;

}

**public** **static** List cutLength(List lengths) {

**int** small = lengths.get(0);

List list = **new** LinkedList();

**for** (**int** i = 0; i < lengths.size(); i++) {

**if** (lengths.get(i) - small > 0)

list.add(lengths.get(i) - small);

}

**return** list;

}

Distance Metric

**Example:**

*n = 6*

*arr = [1, 2, 1, 1, 2, 3].*

The element *arr[0] = 1.*Similar elements are at indices*2*and*3.*

The distance metric for *arr[0] = |0-2| + |0-3| = 5*

Similar logic follows:

The distance metric for *arr[1] = |1-4| = 3*

The distance metric for *arr[2] = |2-0| + |2-3| = 3*

The distance metric for *arr[3] = |3-0| + |3-2| = 4*

The distance metric for *arr[4] = |4-1| = 3*

The distance metric for *arr[5] = 0*

Thus, distance metrics*= [5, 3, 3, 4, 3, 0]*

**public** **static** List<Long> getDistanceMetrics(**int**[] arr) {

List<Long> list = **new** ArrayList();

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

**long** sum = 0;

**for** (**int** j = 0; j < arr.length; j++) {

**if** (arr[i] == arr[j]) {

sum = sum + Math.*abs*(i - j);

}

}

list.add(sum);

}

**return** list;

}

Power Company

**Example:**

*n = 14*

*model = [3, 4, 6, 11, 9, 9, 9, 9, 8, 8, 8, 8, 8, 8].*

There are *14* generators, and the ceiling of *n/2 = 14/2 = 7.*At least *7* generators must be deactivated. One of the optimal solutions is deactivating two types of generators, models *9* and *8.* The number of models *9* and *8* is *4 + 6 = 10*, which is *≥ 7*. The answer is *2*.

**public** **static** **int** reduceCapacity1(List<Integer> model) {

**float** halfSize = model.size() / 2;

**if** (model.size() % 2 != 0) {

halfSize++;

}

Collections.*sort*(model);

HashMap<Integer, Integer> map = **new** HashMap<Integer, Integer>();

**for** (**int** i = 0; i < model.size(); i++) {

Integer intg = map.get(model.get(i));

**if** (intg == **null**) {

map.put(model.get(i), 1);

} **else** {

map.put(model.get(i), intg + 1);

}

}

map = map.entrySet().stream().sorted(Map.Entry.*comparingByValue*(Comparator.*reverseOrder*()))

.collect(Collectors.*toMap*(e -> e.getKey(), e -> e.getValue(), (e1, e2) -> **null**, // or throw an exception

() -> **new** LinkedHashMap<Integer, Integer>()));

**int** result = 0;

**int** resultSum = 0;

System.***out***.println("" + map);

**for** (Map.Entry<Integer, Integer> val : map.entrySet()) {

**if** (halfSize > resultSum) {

resultSum = resultSum + val.getValue();

result++;

}

}

**return** result;

}