Codility Questions

**Random Number generator**

We can generate a pseudo-random number in the range from 0.0 to 32,767 using rand() function from <cstdlib> library. The maximum value is library generated but it is guaranteed to be at least 32767 on any standard implementation. We can check it from RAND\_MAX

<http://www.bogotobogo.com/cplusplus/RandomNumbers.php>

#include <iostream>

#include <cstdlib>

using namespace std;

int main()

{

int i = 0;

while(i++ < 10) {

int r = (rand() % 100) + 1;

cout << r << " ";

}

return 0;

}

But this will generate the same set of numbers always. We can set the seed using the srand method. The srand() method takes integer as argument as seed. If we initiliaze the srand with same number then it will generate the same random number always. To initialize the srand with different values we can pass the value of time(NULL) to srand. The time(NULL) produces the time value which is the number of seconds since epoch.

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

int main()

{

srand((int)time(0));

int i = 0;

while(i++ < 10) {

int r = (rand() % 100) + 1;

cout << r << " ";

}

return 0;

}

Codility: Find the sum of the area of two possible   overlapping rectangles.

*Codility test with two tasks to do in 100 minutes: a   battleship game and a queue manager for people waiting for an elevator.*

<http://stackoverflow.com/questions/12417383/programming-test-codility-dominator>

<http://stackoverflow.com/questions/4703047/review-of-a-codility-test-pair-sum-even-count>

Problem – You have to find a majority element in an [array](http://en.wikipedia.org/wiki/Array) . An element is majority element if it occurs more than half times the size of array.

Solution –

1. Take two variables ,major which will contain majority element at the end and another count variable.
2. Initialize major to first element of the array and set count to one.
3. Now traverse the rest of array from the 2nd element.
4. For each array element a[i] ,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | if ( major==a[i] )  count++;  else if ( count == 0 )  {  major=a[i];  count=1;  }  else  count--; |

1. At the end of traversing the array, major will consist of majority element.
2. [**Equi**](https://codility.com/demo/results/demoZQ2TSZ-7QY/)

Find an index in an array such that its prefix sum equals its suffix sum.

A zero-indexed array A consisting of N integers is given. An *equilibrium index* of this array is any integer P such that 0 ≤ P < N and the sum of elements of lower indices is equal to the sum of elements of higher indices, i.e.

A[0] + A[1] + ... + A[P−1] = A[P+1] + ... + A[N−2] + A[N−1].

Sum of zero elements is assumed to be equal to 0. This can happen if P = 0 or if P = N−1.

int solution(vector<int> &A) {

// write your code in C++14 (g++ 6.2.0)

int N = A.size();

vector<long> s(A.size(), false);

long sum = 0;

for (int i = N-1; i >=0; --i)

{

s[i] = A[i] + sum;

sum = sum + A[i];

}

long lsum = 0;

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

{

long rsum = s[i] - A[i];

// cout << "i=" << i << ", lsum=" << lsum << ", rsum="<< rsum << endl;

if (lsum == rsum)

{

return i;

}

lsum = lsum+A[i];

}

return -1;

}

1. [**BinaryGap**](https://codility.com/programmers/lessons/1-iterations/binary_gap/)

Find longest sequence of zeros in binary representation of an integer.

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.

Write a function:

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.

Assume that:

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

Complexity:

expected worst-case time complexity is O(log(N));

expected worst-case space complexity is O(1).

**Solution**

**int** **max**(**int** a, **int** b)

{

**if** (a>b)

**return** a;

**else**

**return** b;

}

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

// write your code in C++14 (g++ 6.2.0)

**int** num = N; //Input is always a +ve integer. So No need of unsigned int.

**bool** oneFound = **false**;

**int** count = 0;

**int** maxCount = 0;

**while**(num != 0)

{

**bool** isOne = (num &1);

**if** (isOne)

{

maxCount = max(count, maxCount);

count = 0;

oneFound = **true**;

}

**else** **if** (oneFound)

{

count++;

}

num = num>>1;

}

**return** maxCount;

}

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

Find value that occurs in odd number of elements.

A non-empty zero-indexed 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.

The technique is to use (XOR) operator A^A will be 0. Thus the same elements will cancel out and the remaining element is the number to be returned.

int solution(vector<int> &A) {

// write your code in C++14 (g++ 6.2.0)

int N = A.size();

int elem = A[0];

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

{

elem ^= A[i];

}

cout << elem;

return elem;

}

<https://codility.com/demo/results/training3UAAPR-E2K/>

1. **CyclicRotation**

Rotate an array to the right by a given number of steps.

A zero-indexed 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 also moved to the first place.

For example, the rotation of array A = [3, 8, 9, 7, 6] is [6, 3, 8, 9, 7]. The goal is to rotate array A K times; that is, each element of A will be shifted to the right by K indexes.

Write a function:

vector<int> solution(vector<int> &A, int K);

**vector**<**int**> solution(**vector**<**int**> &A, **int** K) {

// write your code in C++14 (g++ 6.2.0)

**int** N = A.size();

**vector**<**int**> r(N);

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

{

**int** np = (i+K)%N;

r[np] = A[i];

}

**return** r;

}

<https://codility.com/demo/results/training7628DE-KJX/>

1. [**PermMissingElem**](https://codility.com/programmers/lessons/3-time_complexity/perm_missing_elem/)

Find the missing element in a given permutation.

A zero-indexed array A consisting of N different integers is given. The array contains integers in the range [1..(N + 1)], which means that exactly one element is missing.

Your goal is to find that missing element.

Write a function:

class Solution {

public int solution(int[] A);

}

that, given a zero-indexed array A, returns the value of the missing element.

For example, given array A such that:

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

the function should return 4, as it is the missing element.

Assume that:

N is an integer within the range [0..100,000];

the elements of A are all distinct;

each element of array A is an integer within the range [1..(N + 1)].

Complexity:

expected worst-case time complexity is O(N);

expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

int solution(vector<int> &A) {

// write your code in C++14 (g++ 6.2.0)

int N = A.size();

long sum = 0;

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

{

sum += A[i];

}

// **BUG** N should be long else rSum will have overflow buffered result.

long rSum = (N+1) \* (N+2) / 2;

int diff = rSum - sum;

return diff;

}

**LEARNING**

**Above solution resulted in 60% correctness**

**Correct Solution**

int solution(vector<int> &A) {

// write your code in C++14 (g++ 6.2.0)

long N = A.size();

long sum = 0;

for (long i = 0; i < N; ++i)

{

sum += A[i];

}

long rSum = (N+1) \* (N+2) / 2;

int diff = rSum - sum;

return diff;

}

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

Count minimal number of jumps from position X to Y.

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:

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

Assume that:

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

Complexity:

* expected worst-case time complexity is O(1);
* expected worst-case space complexity is O(1).

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

// write your code in C++14 (g++ 6.2.0)

int count = (Y-X)/D;

int mod = (Y-X)%D;

if (mod != 0)

{

count++;

}

return count;

}

1. [**TapeEquilibrium**](https://codility.com/programmers/lessons/3-time_complexity/tape_equilibrium/)

Minimize the value |(A[0] + ... + A[P-1]) - (A[P] + ... + A[N-1])|.

A non-empty zero-indexed array A consisting of N integers is given. Array A represents numbers on a tape.

Any integer P, such that 0 < P < N, splits this tape into two non-empty parts: A[0], A[1], ..., A[P − 1] and A[P], A[P + 1], ..., A[N − 1].

The *difference* between the two parts is the value of: |(A[0] + A[1] + ... + A[P − 1]) − (A[P] + A[P + 1] + ... + A[N − 1])|

In other words, it is the absolute difference between the sum of the first part and the sum of the second part.

For example, consider array A such that:

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

We can split this tape in four places:

* P = 1, difference = |3 − 10| = 7
* P = 2, difference = |4 − 9| = 5
* P = 3, difference = |6 − 7| = 1
* P = 4, difference = |10 − 3| = 7

Write a function:

int solution(int A[], int N);

that, given a non-empty zero-indexed array A of N integers, returns the minimal difference that can be achieved.

For example, given:

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

the function should return 1, as explained above.

Assume that:

* N is an integer within the range [2..100,000];
* each element of array A is an integer within the range [−1,000..1,000].

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

#include <climits>

int min(int a, int b)

{

if (a<b)

return a;

else

return b;

}

int solution(vector<int> &A) {

// write your code in C++14 (g++ 6.2.0)

int N = A.size();

vector<int> s(N);

s[N-1] = A[N-1];

for (int i = N-2; i >=0; --i)

{

s[i] = A[i] + s[i+1];

}

int minDiff = INT\_MAX; // min should be INT\_MAX and max should be INT\_MIN

int lsum =A[0];

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

{

int diff = lsum - s[i];

if (diff < 0)

diff = diff \* -1;

minDiff = min(diff, minDiff);

lsum += A[i];

}

return minDiff;

}

1. **CountDIV**
2. **CountDIV**
3. **CountDIV**
4. **CountDIV**

**TBD**

1. **CountDIV**

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.

Assume that:

* 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.

Complexity:

* expected worst-case time complexity is O(1);
* expected worst-case space complexity is O(1).

**Solution**

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

// write your code in C++14 (g++ 6.2.0)

int offsetForLeftRange = 0;

if ( A % K == 0) { ++offsetForLeftRange; }

return (B/K) - (A /K) + offsetForLeftRange;

}

1. **Passing Cars**

A non-empty zero-indexed array A consisting of N integers is given. The consecutive elements of array A represent consecutive cars on a road.

Array A contains only 0s and/or 1s:

* 0 represents a car traveling east,
* 1 represents a car traveling west.

The goal is to count passing cars. We say that a pair of cars (P, Q), where 0 ≤ P < Q < N, is passing when P is traveling to the east and Q is traveling to the west.

For example, consider array A such that:

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

We have five pairs of passing cars: (0, 1), (0, 3), (0, 4), (2, 3), (2, 4).

Write a function:

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

that, given a non-empty zero-indexed array A of N integers, returns the number of pairs of passing cars.

The function should return −1 if the number of pairs of passing cars exceeds 1,000,000,000.

For example, given:

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

the function should return 5, as explained above.

Assume that:

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

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**Solution**

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**long inc =0;**

**long count = 0;**

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

**{**

**if (A[i] == 0)**

**{**

**inc++;**

**}**

**if (A[i] == 1)**

**{**

**count = count + inc;**

**}**

**if (count > 1000000000)**

**{**

**return -1;**

**}**

**}**

**return count;**

**}**

1. **Distinct**

Write a function

int solution(vector<int> &A);

that, given a zero-indexed array A consisting of N integers, returns the number of distinct values in array A.

Assume that:

* 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].

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.

Complexity:

* expected worst-case time complexity is O(N\*log(N));
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**Solution**

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**if (N==0)**

**return 0;**

**sort(A.begin(), A.end());**

**int count = 1;**

**for (int i =0; i < N-1; ++i)**

**{**

**if (A[i] != A[i+1])**

**count++;**

**}**

**return count;**

**}**

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

Maximize A[P] \* A[Q] \* A[R] for any triplet (P, Q, R).

A non-empty zero-indexed 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:

int solution(int A[], int N);

that, given a non-empty zero-indexed 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.

Assume that:

* 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].

Complexity:

* expected worst-case time complexity is O(N\*log(N));
* expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**sort(A.begin(), A.end());**

**return max(A[0]\*A[1]\*A[N-1], A[N-1]\*A[N-2]\*A[N-3]);**

**}**

1. [**Triangle**](https://codility.com/programmers/lessons/6-sorting/triangle/)

Determine whether a triangle can be built from a given set of edges.

A zero-indexed array A consisting of N integers is given. A triplet (P, Q, R) is *triangular* if 0 ≤ P < Q < R < N and:

* A[P] + A[Q] > A[R],
* A[Q] + A[R] > A[P],
* A[R] + A[P] > A[Q].

For example, consider array A such that:

A[0] = 10 A[1] = 2 A[2] = 5 A[3] = 1 A[4] = 8 A[5] = 20

Triplet (0, 2, 4) is triangular.

Write a function:

int solution(vector<int> &A);

that, given a zero-indexed array A consisting of N integers, returns 1 if there exists a triangular triplet for this array and returns 0 otherwise.

For example, given array A such that:

A[0] = 10 A[1] = 2 A[2] = 5 A[3] = 1 A[4] = 8 A[5] = 20

the function should return 1, as explained above. Given array A such that:

A[0] = 10 A[1] = 50 A[2] = 5 A[3] = 1

the function should return 0.

Assume that:

* 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].

Complexity:

* expected worst-case time complexity is O(N\*log(N));
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**sort(A.begin(), A.end());**

**if (A.size() <3)**

**return 0;**

**int N = A.size();**

**for (int i = 0; i < N-2; ++i)**

**{**

**long ai = A[i];**

**long ai1 = A[i+1];**

**long ai2 = A[i+2];**

**if ((ai + ai1) > ai2)**

**{**

**return 1;**

**}**

**}**

**return 0;**

**}**

**Explanation**

In a sorted array if this condition hold true

A[i] + A[i+1] > A[i+2]

Then the other two combinations will also hold true

A[i] + A[i+2] > A[i+1]

A[i+2] + A[i+1] > A[i]

This is because A[i] and A[i+1] are smaller than A[i+2] and if the number A[i] + A[i+1] are greater than A[i+2], then any sum with A[i+2] will be greater than

1. [**NumberOfDiscIntersections**](https://codility.com/programmers/lessons/6-sorting/number_of_disc_intersections/)

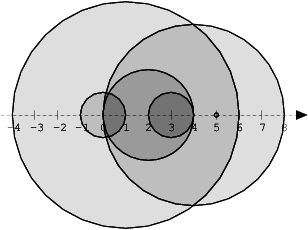
Compute the number of intersections in a sequence of discs.

We draw N discs on a plane. The discs are numbered from 0 to N − 1. A zero-indexed array A of N non-negative integers, specifying the radiuses of the discs, is given. The J-th disc is drawn with its center at (J, 0) and radius A[J].

We say that the J-th disc and K-th disc intersect if J ≠ K and the J-th and K-th discs have at least one common point (assuming that the discs contain their borders).

The figure below shows discs drawn for N = 6 and A as follows:

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



There are eleven (unordered) pairs of discs that intersect, namely:

* discs 1 and 4 intersect, and both intersect with all the other discs;
* disc 2 also intersects with discs 0 and 3.

Write a function:

int solution(int A[], int N);

that, given an array A describing N discs as explained above, returns the number of (unordered) pairs of intersecting discs. The function should return −1 if the number of intersecting pairs exceeds 10,000,000.

Given array A shown above, the function should return 11, as explained above.

Assume that:

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

Complexity:

* expected worst-case time complexity is O(N\*log(N));
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**Solutions**

**TBD**

1. [**Brackets**](https://codility.com/demo/results/trainingR48BCE-DJ2/)

Determine whether a given string of parentheses is properly nested.

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:

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.

Assume that:

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

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N) (not counting the storage required for input arguments).

**int solution(string &S) {**

**// write your code in C++14 (g++ 6.2.0)**

**stack<char> st;**

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

**{**

**char c = S[i];**

**if (c == '(' || c == '{' || c == '[')**

**{**

**//cout << "Push="<< c << endl;**

**st.push(S[i]);**

**}**

**else if (c == ')' || c == '}' || c == ']')**

**{**

**if (st.empty())**

**{**

**return 0;**

**}**

**char t = st.top();**

**if (c == ')' && t == '(')**

**{**

**//cout << "Pop="<< c << endl;**

**st.pop();**

**}**

**else if (c == ']' && t == '[')**

**{**

**//cout << "Pop="<< c << endl;**

**st.pop();**

**}**

**else if (c == '}' && t == '{')**

**{**

**//cout << "Pop="<< c << endl;**

**st.pop();**

**}**

**else**

**{**

**return 0;**

**}**

**}**

**}**

**if (!st.empty())**

**return 0;**

**return 1;**

**}**

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

N voracious fish are moving along a river. Calculate how many fish are alive.

You are given two non-empty zero-indexed 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:

int solution(int A[], int B[], int N);

that, given two non-empty zero-indexed 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.

Assume that:

* 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.

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**Solution**

**int solution(vector<int> &A, vector<int> &B) {**

**// write your code in C++14 (g++ 6.2.0)**

**int count = 0;**

**stack<int> previousFishes;**

**int N = A.size();**

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

**{**

**int currentFish = A[i];**

**int currentFlow = B[i];**

**// Make a note of fish going down**

**if (currentFlow == 1) {**

**previousFishes.push(currentFish);**

**}**

**// If fish going up and there are fishes going down, check what will be eaten**

**if (!previousFishes.empty() && currentFlow == 0)**

**{**

**while (!previousFishes.empty() && currentFish > previousFishes.top()) {**

**previousFishes.pop();**

**}**

**}**

**// If fish going up and no previous fishes going down, then increase the count**

**if (previousFishes.empty() && currentFlow == 0) {**

**count++;**

**}**

**}**

**return count + previousFishes.size();**

**}**

1. [**Nesting**](https://codility.com/programmers/lessons/7-stacks_and_queues/nesting/)

Determine whether given string of parentheses is properly nested.

A string S consisting of N characters is called *properly nested* if:

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

For example, string "(()(())())" is properly nested but string "())" isn't.

Write a function:

int solution(char \*S);

that, given a string S consisting of N characters, returns 1 if string 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.

Assume that:

* N is an integer within the range [0..1,000,000];
* string S consists only of the characters "(" and/or ")".

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(1) (not counting the storage required for input arguments).

**int solution(string &S) {**

**// write your code in C++14 (g++ 6.2.0)**

**int inCount=0;**

**int N = S.size();**

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

**{**

**if (S[i] == '(')**

**inCount++;**

**else**

**inCount--;**

**if (inCount < 0)**

**return 0;**

**}**

**if (inCount != 0)**

**return 0;**

**return 1;**

**}**

1. [**StoneWall**](https://codility.com/programmers/lessons/7-stacks_and_queues/stone_wall/)

Cover "Manhattan skyline" using the minimum number of rectangles.

You are going to build a stone wall. The wall should be straight and N meters long, and its thickness should be constant; however, it should have different heights in different places. The height of the wall is specified by a zero-indexed array H of N positive integers. H[I] is the height of the wall from I to I+1 meters to the right of its left end. In particular, H[0] is the height of the wall's left end and H[N−1] is the height of the wall's right end.

The wall should be built of cuboid stone blocks (that is, all sides of such blocks are rectangular). Your task is to compute the minimum number of blocks needed to build the wall.

Write a function:

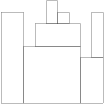
int solution(int H[], int N);

that, given a zero-indexed array H of N positive integers specifying the height of the wall, returns the minimum number of blocks needed to build it.

For example, given array H containing N = 9 integers:

H[0] = 8 H[1] = 8 H[2] = 5 H[3] = 7 H[4] = 9 H[5] = 8 H[6] = 7 H[7] = 4 H[8] = 8

the function should return 7. The figure shows one possible arrangement of seven blocks.



Assume that:

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

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**int solution(vector<int> &H) {**

**// write your code in C++14 (g++ 6.2.0)**

**int stones=0;**

**stack<int> s;**

**int N = H.size();**

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

**{**

**while ( (!s.empty()) && (s.top() > H[i]) )**

**s.pop();**

**//if (!(!s.empty() && s.top() == H[i]))**

**if (s.empty() || s.top() != H[i] )**

**{**

**// We enter here if stack is empty or stack.top() < H[i]**

**stones +=1;**

**s.push(H[i]);**

**}**

**}**

**return stones;**

**}**

1. [**EquiLeader**](https://codility.com/programmers/lessons/8-leader/equi_leader/)

Find the index S such that the leaders of the sequences A[0], A[1], ..., A[S] and A[S + 1], A[S + 2], ..., A[N - 1] are the same.

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

The *leader* of this array is the value that occurs in more than half of the elements of A.

An *equi leader* is an index S such that 0 ≤ S < N − 1 and two sequences A[0], A[1], ..., A[S] and A[S + 1], A[S + 2], ..., A[N − 1] have leaders of the same value.

For example, given array A such that:

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

we can find two equi leaders:

* 0, because sequences: (4) and (3, 4, 4, 4, 2) have the same leader, whose value is 4.
* 2, because sequences: (4, 3, 4) and (4, 4, 2) have the same leader, whose value is 4.

The goal is to count the number of equi leaders.

Write a function:

int solution(int A[], int N);

that, given a non-empty zero-indexed array A consisting of N integers, returns the number of equi leaders.

For example, given:

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

the function should return 2, as explained above.

Assume that:

* 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].

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**Solution:**

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**int equi\_leaders = 0;**

**// first, compute the leader**

**int leader = A[0];**

**int ctr = 1;**

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

**if(A[i] == leader) ctr++;**

**else ctr--;**

**if(ctr == 0){**

**ctr = 1;**

**leader = A[i];**

**}**

**}**

**// check if it's a leader?**

**int total = 0;**

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

**if(A[i] == leader) total++;**

**}**

**if(total <= (N/2)) return 0; // impossible**

**int ldrCount = 0;**

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

**if(A[i] == leader) ldrCount++;**

**int leadersInRightPart = (total - ldrCount);**

**if(ldrCount > (i+1)/2 && leadersInRightPart > (N-i-1)/2){**

**equi\_leaders++;**

**}**

**}**

**return equi\_leaders;**

**}**

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

Find an index of an array such that its value occurs at more than half of indices in the array.

A zero-indexed 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

int solution(int A[], int N);

that, given a zero-indexed 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.

Assume that:

* 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].

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.

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**if (N ==0)**

**return -1;**

**// first, compute the leader**

**int leader = A[0];**

**int ctr = 1;**

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

**if(A[i] == leader) ctr++;**

**else ctr--;**

**if(ctr == 0){**

**ctr = 1;**

**leader = A[i];**

**}**

**}**

**// check if it's a leader?**

**int total = 0;**

**int index = 0;**

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

**if(A[i] == leader)**

**{**

**index = i;**

**total++;**

**}**

**}**

**if(total <= (N/2)) return -1; // impossible**

**return index;**

**}**

1. [**MaxProfit**](https://codility.com/programmers/lessons/9-maximum_slice_problem/max_profit/)

Given a log of stock prices compute the maximum possible earning.

A zero-indexed array A consisting of N integers is given. It contains daily prices of a stock share for a period of N consecutive days. If a single share was bought on day P and sold on day Q, where 0 ≤ P ≤ Q < N, then the *profit* of such transaction is equal to A[Q] − A[P], provided that A[Q] ≥ A[P]. Otherwise, the transaction brings *loss* of A[P] − A[Q].

For example, consider the following array A consisting of six elements such that:

A[0] = 23171 A[1] = 21011 A[2] = 21123 A[3] = 21366 A[4] = 21013 A[5] = 21367

If a share was bought on day 0 and sold on day 2, a loss of 2048 would occur because A[2] − A[0] = 21123 − 23171 = −2048. If a share was bought on day 4 and sold on day 5, a profit of 354 would occur because A[5] − A[4] = 21367 − 21013 = 354. Maximum possible profit was 356. It would occur if a share was bought on day 1 and sold on day 5.

Write a function,

int solution(int A[], int N);

that, given a zero-indexed array A consisting of N integers containing daily prices of a stock share for a period of N consecutive days, returns the maximum possible profit from one transaction during this period. The function should return 0 if it was impossible to gain any profit.

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

A[0] = 23171 A[1] = 21011 A[2] = 21123 A[3] = 21366 A[4] = 21013 A[5] = 21367

the function should return 356, as explained above.

Assume that:

* N is an integer within the range [0..400,000];
* each element of array A is an integer within the range [0..200,000].

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**int min(int a, int b)**

**{**

**if (a<b)**

**return a;**

**else**

**return b;**

**}**

**int max(int a, int b)**

**{**

**if (a>b)**

**return a;**

**else**

**return b;**

**}**

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int gmin = 200000;**

**int gmax = 0;**

**int N = A.size();**

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

**{**

**gmin = min(gmin, A[i]);**

**gmax = max(gmax,A[i] - gmin);**

**}**

**return gmax;**

**}**

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

Find a maximum sum of a compact subsequence of array elements.

A non-empty zero-indexed 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:

int solution(int A[], int N);

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).

Assume that:

* 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].

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int gmax = A[0];**

**int cmax = A[0];**

**int N = A.size();**

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

**{**

**cmax = max(cmax+A[i], A[i]);**

**gmax = max(cmax, gmax);**

**}**

**return gmax;**

**}**

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

Find the maximal sum of any double slice.

A non-empty zero-indexed 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:

int solution(int A[], int N);

that, given a non-empty zero-indexed 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.

Assume that:

* 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].

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**TBD: I did not understand why is the max compared with 0(but not A[i]) in below code**

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

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

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**vector<int> k1(N);**

**vector<int> k2(N);**

**k1[0]=0;**

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

**{**

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

**}**

**k2[N-1]=0;**

**for (int i = N-2; i > 0; --i)**

**{**

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

**}**

**int maxSum = 0;**

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

**{**

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

**}**

**return maxSum;**

**}**

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

Count factors of given number n.

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:

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.

Assume that:

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

Complexity:

* expected worst-case time complexity is O(sqrt(N));
* expected worst-case space complexity is O(1).

**int solution(int N) {**

**// write your code in C++14 (g++ 6.2.0)**

**int i =1;**

**int result = 0;**

**while( i\*i < N)**

**{**

**if (N%i ==0)**

**result += 2;**

**++i;**

**}**

**if (i\*i == N)**

**++result;**

**return result;**

**}**

1. [**MinPerimeterRectangle**](https://codility.com/programmers/lessons/10-prime_and_composite_numbers/min_perimeter_rectangle/)

Find the minimal perimeter of any rectangle whose area equals N.

An integer N is given, representing the area of some rectangle.

The *area* of a rectangle whose sides are of length A and B is A \* B, and the *perimeter* is 2 \* (A + B).

The goal is to find the minimal perimeter of any rectangle whose area equals N. The sides of this rectangle should be only integers.

For example, given integer N = 30, rectangles of area 30 are:

* (1, 30), with a perimeter of 62,
* (2, 15), with a perimeter of 34,
* (3, 10), with a perimeter of 26,
* (5, 6), with a perimeter of 22.

Write a function:

int solution(int N);

that, given an integer N, returns the minimal perimeter of any rectangle whose area is exactly equal to N.

For example, given an integer N = 30, the function should return 22, as explained above.

Assume that:

* N is an integer within the range [1..1,000,000,000].

Complexity:

* expected worst-case time complexity is O(sqrt(N));
* expected worst-case space complexity is O(1).

**int solution(int N) {**

**// write your code in C++14 (g++ 6.2.0)**

**int i =1;**

**int minPeri = INT\_MAX;**

**while( i\*i <= N)**

**{**

**if (N%i ==0)**

**{**

**int div = N/i;**

**int peri = 2\*(div+i);**

**if (peri<minPeri)**

**minPeri = peri;**

**}**

**++i;**

**}**

**return minPeri;**

**}**

1. [**Peaks**](https://codility.com/programmers/lessons/10-prime_and_composite_numbers/peaks/)

Divide an array into the maximum number of same-sized blocks, each of which should contain an index P such that A[P - 1] < A[P] > A[P + 1].

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

A *peak* is an array element which is larger than its neighbors. More precisely, it is an index P such that 0 < P < N − 1,  A[P − 1] < A[P] and A[P] > A[P + 1].

For example, the following array A:

A[0] = 1 A[1] = 2 A[2] = 3 A[3] = 4 A[4] = 3 A[5] = 4 A[6] = 1 A[7] = 2 A[8] = 3 A[9] = 4 A[10] = 6 A[11] = 2

has exactly three peaks: 3, 5, 10.

We want to divide this array into blocks containing the same number of elements. More precisely, we want to choose a number K that will yield the following blocks:

* A[0], A[1], ..., A[K − 1],
* A[K], A[K + 1], ..., A[2K − 1],  
  ...
* A[N − K], A[N − K + 1], ..., A[N − 1].

What's more, every block should contain at least one peak. Notice that extreme elements of the blocks (for example A[K − 1] or A[K]) can also be peaks, but only if they have both neighbors (including one in an adjacent blocks).

The goal is to find the maximum number of blocks into which the array A can be divided.

Array A can be divided into blocks as follows:

* one block (1, 2, 3, 4, 3, 4, 1, 2, 3, 4, 6, 2). This block contains three peaks.
* two blocks (1, 2, 3, 4, 3, 4) and (1, 2, 3, 4, 6, 2). Every block has a peak.
* three blocks (1, 2, 3, 4), (3, 4, 1, 2), (3, 4, 6, 2). Every block has a peak. Notice in particular that the first block (1, 2, 3, 4) has a peak at A[3], because A[2] < A[3] > A[4], even though A[4] is in the adjacent block.

However, array A cannot be divided into four blocks, (1, 2, 3), (4, 3, 4), (1, 2, 3) and (4, 6, 2), because the (1, 2, 3) blocks do not contain a peak. Notice in particular that the (4, 3, 4) block contains two peaks: A[3] and A[5].

The maximum number of blocks that array A can be divided into is three.

Write a function:

int solution(int A[], int N);

that, given a non-empty zero-indexed array A consisting of N integers, returns the maximum number of blocks into which A can be divided.

If A cannot be divided into some number of blocks, the function should return 0.

For example, given:

A[0] = 1 A[1] = 2 A[2] = 3 A[3] = 4 A[4] = 3 A[5] = 4 A[6] = 1 A[7] = 2 A[8] = 3 A[9] = 4 A[10] = 6 A[11] = 2

the function should return 3, as explained above.

Assume that:

* 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].

Complexity:

* expected worst-case time complexity is O(N\*log(log(N)));
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**if (N < 3)**

**return 0;**

**// Find all the peaks**

**vector<int> peaks;**

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

**if(A[i] > A[i-1] && A[i] > A[i+1]) peaks.push\_back(i);**

**}**

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

**if(N % size != 0) continue; // skip if non-divisible**

**int find = 0;**

**int groups = N/size;**

**bool ok = true;**

**// Find whether every group has a peak**

**for(int peakIdx : peaks){**

**//cout << "peaks.size()=" << peaks.size() << endl;**

**if(peakIdx/size > find){**

**ok = false;**

**break;**

**}**

**if(peakIdx/size == find) find++;**

**}**

**if(find != groups) ok = false;**

**if(ok)**

**{**

**//cout << "find="<<find<<", groups="<<groups<<", size="<<size <<endl;**

**return groups;**

**}**

**}**

**return 0;**

**}**

1. [**Flags**](https://codility.com/programmers/lessons/10-prime_and_composite_numbers/flags/)

Find the maximum number of flags that can be set on mountain peaks.

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

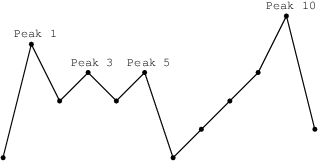
A *peak* is an array element which is larger than its neighbours. More precisely, it is an index P such that 0 < P < N − 1 and A[P − 1] < A[P] > A[P + 1].

For example, the following array A:

A[0] = 1 A[1] = 5 A[2] = 3 A[3] = 4 A[4] = 3 A[5] = 4 A[6] = 1 A[7] = 2 A[8] = 3 A[9] = 4 A[10] = 6 A[11] = 2

has exactly four peaks: elements 1, 3, 5 and 10.

You are going on a trip to a range of mountains whose relative heights are represented by array A, as shown in a figure below. You have to choose how many flags you should take with you. The goal is to set the maximum number of flags on the peaks, according to certain rules.



Flags can only be set on peaks. What's more, if you take K flags, then the distance between any two flags should be greater than or equal to K. The distance between indices P and Q is the absolute value |P − Q|.

For example, given the mountain range represented by array A, above, with N = 12, if you take:

* two flags, you can set them on peaks 1 and 5;
* three flags, you can set them on peaks 1, 5 and 10;
* four flags, you can set only three flags, on peaks 1, 5 and 10.

You can therefore set a maximum of three flags in this case.

Write a function:

int solution(int A[], int N);

that, given a non-empty zero-indexed array A of N integers, returns the maximum number of flags that can be set on the peaks of the array.

For example, the following array A:

A[0] = 1 A[1] = 5 A[2] = 3 A[3] = 4 A[4] = 3 A[5] = 4 A[6] = 1 A[7] = 2 A[8] = 3 A[9] = 4 A[10] = 6 A[11] = 2

the function should return 3, as explained above.

Assume that:

* N is an integer within the range [1..400,000];
* each element of array A is an integer within the range [0..1,000,000,000].

Complexity:

* expected worst-case time complexity is O(N);
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

<https://codility.com/media/train/solution-flags.pdf>

**vector<bool> createPeaks(vector<int>& A)**

**{**

**int N = A.size();**

**vector<bool> vb(N, false);**

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

**{**

**if (A[i-1] < A[i] && A[i] > A[i+1])**

**vb[i] = true;**

**}**

**return vb;**

**}**

**vector<int> nextPeak(vector<int>& A)**

**{**

**vector<bool> vb = createPeaks(A);**

**int N = A.size();**

**vector<int> next(N);**

**next[N-1] = -1;**

**for (int i = N-2; i > -1; --i)**

**{**

**if (vb[i] == true)**

**next[i] = i;**

**else**

**next[i] = next[i+1];**

**}**

**return next;**

**}**

**int max(int a, int b)**

**{**

**if (a>b)**

**return a;**

**else**

**return b;**

**}**

**int solution(vector<int> &A) {**

**// write your code in C++14 (g++ 6.2.0)**

**int N = A.size();**

**if (N < 3)**

**return 0;**

**vector<int> next = nextPeak(A);**

**int i = 1;**

**int result = 0;**

**while( (i-1)\*i <= N)**

**{**

**int pos = 0;**

**int num = 0;**

**while(pos < N && num < i)**

**{**

**pos = next[pos];**

**if (pos == -1)**

**break;**

**num += 1;**

**pos += i; // Increment pos by i**

**}**

**result = max(result, num);**

**i += 1;**

**}**

**return result;**

**}**

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

Count the number of different ways of climbing to the top of a 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:

vector<int> solution(vector<int> &A, vector<int> &B);

that, given two non-empty zero-indexed 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.

Assume that:

* L is an integer within the range [1..30,000];
* 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].

Complexity:

* expected worst-case time complexity is O(L);
* expected worst-case space complexity is O(L), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**#include <math.h>**

**// you can write to stdout for debugging purposes, e.g.**

**// cout << "this is a debug message" << endl;**

**int getMax(vector<int> v)**

**{**

**int N = v.size();**

**int max = v[0];**

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

**{**

**if(v[i] > max)**

**max = v[i];**

**}**

**return max;**

**}**

**vector<int> buildCache(int aM, int bM)**

**{**

**vector<int> cache(aM+1);**

**cache[1] = 1; // Although not needed**

**cache[2] = 2;**

**for (int i = 3; i <= aM; ++i)**

**{**

**cache[i] = (cache[i-1] + cache[i-2]) % ((int)pow((int)2,(int)bM));**

**}**

**return cache;**

**}**

**vector<int> solution(vector<int> &A, vector<int> &B) {**

**// write your code in C++14 (g++ 6.2.0)**

**int aMax = getMax(A);**

**int bMax = getMax(B);**

**vector<int> cache = buildCache(aMax, bMax);**

**int N = A.size();**

**vector<int> result(N);**

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

**{**

**result[i] = cache[A[i]] % (int)pow((int)2, (int)B[i]);**

**}**

**return result;**

**}**

1. [**FibFrog**](https://codility.com/programmers/lessons/13-fibonacci_numbers/fib_frog/)

Count the minimum number of jumps required for a frog to get to the other side of a river.

The Fibonacci sequence is defined using the following recursive formula:

F(0) = 0 F(1) = 1 F(M) = F(M - 1) + F(M - 2) if M >= 2

A small frog wants to get to the other side of a river. The frog is initially located at one bank of the river (position −1) and wants to get to the other bank (position N). The frog can jump over any distance F(K), where F(K) is the K-th Fibonacci number. Luckily, there are many leaves on the river, and the frog can jump between the leaves, but only in the direction of the bank at position N.

The leaves on the river are represented in a zero-indexed array A consisting of N integers. Consecutive elements of array A represent consecutive positions from 0 to N − 1 on the river. Array A contains only 0s and/or 1s:

* 0 represents a position without a leaf;
* 1 represents a position containing a leaf.

The goal is to count the minimum number of jumps in which the frog can get to the other side of the river (from position −1 to position N). The frog can jump between positions −1 and N (the banks of the river) and every position containing a leaf.

For example, consider array A such that:

A[0] = 0 A[1] = 0 A[2] = 0 A[3] = 1 A[4] = 1 A[5] = 0 A[6] = 1 A[7] = 0 A[8] = 0 A[9] = 0 A[10] = 0

The frog can make three jumps of length F(5) = 5, F(3) = 2 and F(5) = 5.

Write a function:

int solution(int A[], int N);

that, given a zero-indexed array A consisting of N integers, returns the minimum number of jumps by which the frog can get to the other side of the river. If the frog cannot reach the other side of the river, the function should return −1.

For example, given:

A[0] = 0 A[1] = 0 A[2] = 0 A[3] = 1 A[4] = 1 A[5] = 0 A[6] = 1 A[7] = 0 A[8] = 0 A[9] = 0 A[10] = 0

the function should return 3, as explained above.

Assume that:

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

Complexity:

* expected worst-case time complexity is O(N\*log(N));
* expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).

Elements of input arrays can be modified.

**TBD: I don’t understand the question.**

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

There are N chocolates in a circle. Count the number of chocolates you will eat.

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:

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.

Assume that:

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

Complexity:

* expected worst-case time complexity is O(log(N+M));
* expected worst-case space complexity is O(log(N+M)).

**This has very good explanation:**

<https://www.martinkysel.com/codility-chocolatesbynumbers-solution/>

N and M meet at their least common multiply. Dividing this LCM by M gets the number of steps(chocolates) that can be eaten.

**This has better code**

<https://codesays.com/2014/solution-to-chocolates-by-numbers-by-codility/>

**long gcd(long a, long b)**

**{**

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

**return b;**

**else**

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

**}**

**long mysolution(long N, long M) {**

**// write your code in C++14 (g++ 6.2.0)**

**long lcm = M\*N/gcd(N,M);**

**return lcm/M;**

**}**

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

**return mysolution(N,M);**

**}**

**Find number of possible triangles**

**http://www.geeksforgeeks.org/find-number-of-triangles-possible/**

* Min Heap data structures stl
* Shared ptr implementation
* Thread safe queue implementation
* Fibonacci with recursion