CROATIAN OPEN COMPETITION IN INFORMATICS

1st ROUND

SOLUTIONS

Observe that a team can be formed if three conditions are satisfied: the number of girls is at least 2, the number of boys is at least 1, and **M**+**N** ≥ **K**+3 holds (since a team consists of three students, and **K** students need to go on an internship). We naturally arrive at a greedy algorithm - forming teams as long as the conditions are met. More precisely, the pseudocode is as follows:≥

```
while (M ≥ 2 and N ≥ 1 and M+N ≥ K+3) do
{
        result := result+1;        (increment the number of formed teams)
        M := M-2;                  (decrease the number of girls)
        N := N-1;                  (decrease the number of boys)
}
```

Alternative solution:

If there are at least twice as many girls as there are boys, we can say that they form a surplus regarding to team formation, otherwise the boys form a surplus. Thus, we can repeat **K** times: check if there is a surplus of girls; if so, decrement the number of girls (i.e. invite a girl to the internship), otherwise decrement the number of boys (i.e. invite a boy to the internship). In the end, we calculate the number of teams we can form from the remaining boys and girls.

Necessary skills:

Number comparison, *while* or *for* loop

Tags:

Ad-hoc, greedy algorithms

Observe that checking every possible interval and every possible grade is too slow for a sufficiently large **N**. Such a solution has a complexity of O($\mathbf{N}^3$) and is worth 70% of points.

To obtain a complete score, we need another approach.

If we assume the grade which the professor will award the students, we can, in a single pass, determine the longest continuous subsequence of desks such that each desk contains at least one student deserving the assumed grade. This can be implemented using a counter storing the current number of continuous desks, and updating it for each desk.

Finally, the solution consists of iterating the described algorithm for each grade from 1 through 5 and taking the maximum of the individual results.

Also observe that the problem can be solved with complexity O(**N**) even if grades are from the interval 1 through **N**. Solving this problem is left as an exercise for the reader.

Necessary skills:

for loop

Tags:

Ad-hoc, dynamic programming

COCI 2010/11	Task SRETAN
1st round, October 23rd, 2010	**Author:** Goran Gašić

A naïve solution would iterate over positive integers and check their luck until the **K**-th lucky integer is found. Such a solution is worth 20% of points.

A better solution is possible if a pattern is noticed in lucky numbers.

Specifically, if we substitute the digit 4 with 0, and 7 with 1, the lucky numbers correspond to binary number notation, with an additional quirk that leading zeros are allowed.

A solution iteratively generating the first **K** lucky numbers using this observation has a complexity of O(**K**) and is worth 50% of points.

Another speedup can be made observing the number of lucky numbers with a given length. There are $2^{\mathbf{N}}$ lucky numbers with length **N** (since each of **N** positions can contain one of two digits).

Knowing this, we can determine the length **L** of the required lucky number and the index **P** of that number among all lucky numbers of length **L**. Then, it is sufficient to output the number **P**-1 in binary with the appropriate number of leading zeros, substituting digits 0 and 1 with 4 and 7, respectively.

Necessary skills:

Pattern recognition, binary number system

Tags:

Ad-hoc

COCI 2010/11	Task LJUTNJA
1st round, October 23rd, 2010	**Author:** Adrian Satja Kurdija

Let **S** be the sum of all wishes. Imagine that each child was given as many candies as it wants, and now they have to be taken away, leaving **M** candies in total. That is, we have to take **S**-**M** candies away.

Since the sum of the numbers of candies that the children will be missing is constant (equal to **S**-**M**), and we wish to minimize the sum of squares of those numbers, a key observation is that the numbers should be "as equal as possible". More precisely, it is possible to prove mathematically (using the inequality of arithmetic and geometric means) that the sum of squares of positive numbers, with a fixed sum, is minimal when they are equal.

Let **K** be the remaining number of candies that need to be taken away (the starting value of **K** is **S**-**M**). Our goal is to, if possible, take away an equal number of candies from each child, specifically **K**/**N**. If this is not possible, the number of candies taken away should be as close as possible to **K**/**N**.

If the child that wants the smallest amount of candies has at least **K**/**N** candies, we will be able to fulfill the goal by taking **K**/**N** candies away from it, thus reducing the problem to the remaining **N**-1 children (by decrementing **N**, calculating the new value of **K**, and processing the child with the next smallest wish). On the other hand, if the observed child has less than **K**/**N** candies, preventing us from taking that many from it, we will take as many as we can, i.e. all the candies it was given, and proceed to process the remaining children.

In the end, we can just output the sum of squared numbers of candies taken away from the children. The complexity of the described algorithm is linear, but it requires first sorting children's wishes into a nondecreasing sequence, resulting in the total complexity of O(**N** log **N**).

It is also possible to solve this problem using binary search, however this solution is left as an exercise for the reader.

Necessary skills:
Sorting in O(**N** log **N**), discerning arithmetic relationships, greedy algorithms

Tags:
Greedy algorithms, binary search

Given any solution, we can decide to first do all tab additions and then all deletions, since the order of operations is irrelevant. Using this convention, we can ignore the rule stating that no line should have less than zero tabs during algorithm execution.

The task can be interpreted as follows: we will define a sequence of numbers in which the **i**-th number equals $\mathbf{K_i}$-$\mathbf{P_i}$. We will also define two queries: increment or decrement (by one) all numbers from a particular range. Those ranges will be referred to as positive and negative ranges, respectively. The goal is then to transform the initial sequence to a sequence of all zeros by executing a minimal number of queries.

Solution:

Let us consider an optimal solution which consists of some number of positive and negative ranges. We choose a positive and a negative one which overlap, if such exist.

There are four distinct cases of overlapping ranges:

1. Both ranges are equal. This means that the solution found is not optimal, since the two queries will cancel out.

2. Both ranges have equal upper or lower bound. This also means that the solution found is not optimal, since the same effect can be achieved by a single positive or negative range equal to their difference.

3. The ranges share no equal bounds but one range is contained within another. We observe that the overlapping part of the ranges cancels out, but since two ranges are necessary in either case, both solutions are equally good.

4. The ranges share no equal bounds but they partially overlap. The overlapping part again cancels out, and the effect is equal as if we had two non-overlapping ranges. The number of ranges (two) remains unchanged, making both solutions equally good.

Therefore, whenever we encounter case 1 or 2, we can replace two ranges with a single one, as described above. By applying consecutive replacements for those two

cases, it is obvious that the number of ranges is decreasing. Therefore, at some point, we will reach the minimum number of ranges, which means an optimal solution is found. By also substituting ranges in cases 3 and 4, we can always obtain an optimal solution in which no two ranges overlap.

We can divide the sequence into contiguous subsequences of only nonnegative or only nonpositive numbers. This alternating division can be found with linear complexity. Each of them can be solved separately using only negative or only positive intervals, respectively. Since the cases are mutually analogous, in the remainder of the discussion we will consider solving the case with nonnegative numbers using negative intervals.

If there is a zero value in the subsequence, we can divide it into the part left of that zero and the part right of it and solve each of these parts separately. It can always be done since there will never be an interval overlapping a zero in the optimal solution. If there is no zero value, we can use a single interval to solve the subsequence and repeat the same procedure (which has possibly created some zeros). This process will eventually result in an optimal solution. Proving this fact is not difficult, so it is left as an exercise for the reader.

We can represent the (nonnegative) subsequence as a histogram, where numbers represent column heights. One possible way of obtaining a solution for a (sub)sequence is using recursion. We can find the smallest number B using an interval tree. After that, we use a negative interval on the subsequence B times, zeroing the smallest number. Then we can divide the subsequence around the newly created zero and solve the left and right parts recursively. The complexity of this algorithm is O(**N** log **N**).

This recursive algorithm actually divides the histogram to a binary tree, branching to two smaller disjoint cases in each step. Instead of recursion, we can use a well-known algorithm with complexity O(**N**) using a stack.

Alternative solution:

There is a completely different solution, using dynamic programming, that does not depend on the facts proven above. The state is determined by three numbers:

1. the number of opened positive intervals
2. the number of opened negative intervals
3. the index of the current position

In each step, we can:

1. open a new interval beginning at the current index

2. close a previously opened interval (ending it with the previous index)
3. if the number of positive and negative intervals matches the required number of tabs, we can move on to the next index

This algorithm can be implemented with complexity O($\mathbf{N}*\mathbf{M}^2$), where **M** is the largest number of tabs that can appear in a row.

Necessary skills:

Greedy algorithms (correct usage and proof of correctness), converting a histogram into a binary tree using a RMQ structure or a stack, dynamic programming

Tags:

Greedy, dynamic programming

We observe that the frog numbered **N**-1 reaches its initial position in a single jump. Similarly, the frog numbered **N** overshoots its initial position by one (effectively moving one position forward). The algorithm is as follows: we arrange frogs numbered 1 through **N**-1 in such a way that their relative order is preserved in the resulting arrangement. In the end, we just let the **N**-th frog leap until it reaches the correct position.

This leaves frogs numbered 1 through **N**-1, so we need to arrange them. Assume that frogs numbered **N**-1, **N**-2, **N**-3, ..., **X**+1 are arranged into a correct relative ordering. Now, we try to add the **X**-th frog into the relative ordering without violating it. Among the frogs which have already been arranged, we find the frog which should immediately precede the frog numbered **X**. We denote by **D** the number of spaces the frog **X** needs to be moved to be correctly placed.

Let **G** be the greatest common divisor of **X** and **N**-1. Let us see how the frog numbered **X** leaps if the Frog Regent proclaims its name. After (**N**-1)/**G** proclamations, the frog will return to its initial position. After a proclamation, the value **D** mod **G** remains unchanged. Thus, if **D** mod **G** equals zero, we can put the frog to the desired position by repeatedly proclaiming **X**.

In case **D** mod **G** does not equal zero, we should change that number so that **D** mod **G** equals zero by gradually decreasing it as follows: we position the **N**-th frog immediately in front of **X** and then proclaim **X** once. Once **D** mod **G** reaches zero, we can put the frog to the desired position using the aforementioned procedure.

Alternative solution:

Let us assume that there exists a function which would decipher the proclamations needed to get the resulting arrangement of 1, 2, 3, ..., **N**. By changing the order of those proclamations, it is possible to get the reverse ordering, as well. This can be achieved by reversing the sequence of proclamations and replacing each proclamation with a series of same proclamations (left to the reader to calculate the exact number) which results in the reverse movement of the frog being moved.

The above function is yet to be realized. We will put the frog numbered **N**-2 immediately after the frog numbered **N**-1, then the frog numbered **N**-3 immediately after the frog numbered **N**-2 etc., with the frog numbered 1 being finally put

immediately after the frog numbered 2. Then, the frog numbered **N** can trivially be put immediately after the first frog.
Assume that we wish to place the frog **X** immediately after **X**+1. We put the frog numbered 1 after the frog **X**+1, the frog 2 after **X**+1 or 1, the frog 3 after **X**+1 or 1 or 2 etc. We repeat this with all frogs up to and including **X**.
At that moment, we have a circle which comprises two parts: the first one is formed by frogs numbered 1 to **X**, and the second one by frogs numbered **X**+1 to **N**-1 (the position of the frog numbered **N** can be safely ignored).
Now we repeat the procedure, but this time without the frog numbered **X**. We put the frog numbered 1 after the frog **X**, the frog 2 after **X** or 1 etc. The final step is to put the frog numbered **X**-1 immediately after the frog **X**, 1, 2, 3, ..., or **X**-2. This results in the frog numbered **X** being put immediately after the frog numbered **X**+1, which was our initial intent.

Necessary skills:

Number theory

Tags:

Ad-hoc