Helvetic Coding Contest 2017
Solution Sketches

April 1, 2017
online mirror on CodeForces: May 28, 2017

A-C. Heidi and Library

Author: **Jakub Tarnawski**
This is known as the paging/caching problem. It is mostly considered in the field of online algorithms, but here we look at the offline version (where the entire request sequence is known in advance).

A. Easy

The optimum strategy is to, whenever we need to delete a book, delete the one which will be requested the latest in the future. Intuitively, this puts us in the best possible shape to handle further requests. A simple $O(n^3)$ implementation of this strategy is enough to get this version accepted.

B. Medium

Here we need to be much more efficient. There are many data structures that one can use. One idea is to store a queue of remaining requests for each possible book (initializing it at the beginning and removing requests from the front as they arrive) and an `std::set`-like structure (based on a balanced binary search tree; it is also possible to use a priority queue based on a heap) which stores books that are currently at the library, sorted by their next-request time.

C. Hard

Here we do not know of any greedy strategy that works optimally. We have to resort to a maximum flow algorithm – what's worse, it's going to be minimum cost maximum flow :)
We build a graph as follows. Let M be a huge number.

- We create two vertices $2i-1$ and $2i$ for each request $i = 1, 2, ..., n$, a source 0 and a sink $2n+1$.
- Every edge in the graph will have capacity 1.
- For each request $i = 1, ..., n$ we create three edges:
  - from the source to $2i-1$ with cost $c[a[i]]$,
  - from $2i-1$ to $2i$ with cost $-M$,
  - from $2i$ to the sink with cost 0.
- For each two requests $i < j$ we create an edge from $2i$ to $2j-1$ of cost 0 if $a[i] = a[j]$ or of cost $c[a[j]]$ otherwise.

We look for a minimum-cost flow of value exactly k in this graph. We leave the correctness of this as an exercise; to begin, notice that any optimum solution must necessarily take all the edges $(2i-1, 2i)$ because their cost is hugely negative, and thus the solution is a minimum-cost collection of k paths from the source to the sink that together use all of those edges. Now think that we have k memory slots (or places in the library in which to store books). The i-th of these k paths corresponds to the sequence of requests that are served by elements stored in the i-th memory slot. The cost of this path corresponds to the cost of buying the books that occupy this slot.

D-F. Marmots

Author: **Christian Kauth**

Can you remember the day you learned to distinguish shapes? Probably you can't, but you may remember May the 28th, 2017 as the day when you taught your computer to discern bells form rectangles and even estimate their parameterization!

D, F. Gaussian or Uniform?

Whatever your approach is, it would be convenient if it computes a metric that can be compared to a fixed threshold which separates the uniform from the Poisson distributions. There are myriads of approaches to do this.

Probably the most principled approach to solving the problem (all three versions of it, actually!) is what is called maximum-likelihood. For every possible distribution (of which there are 2000) the input samples may have come from, we compute the likelihood – the probability of these samples, given the fixed distribution. Because the samples were independent, this is just the product of their individual probabilities. However, implemented just like that, there would be a problem since this is too small – the result would round to zero for almost any distribution if we wanted to keep it in a `double` or similar variable. To cope with this, we instead compute the logarithm of the product, which is the sum of logarithms of the individual probabilities (called log-likelihood). We return the distribution with the highest likelihood.

And here are a few other (more or less ad-hoc) approaches based on various ideas:

What works

- The variance over mean ratio for Poisson distributions is smaller than for uniform ones. For Poisson, the mean and the variance both converge to the population P for a sufficiently large sample. On the other hand, the mean of a uniform distribution lies around its center, while its variance increases with the square of the mean (and significantly outweights the mean for $P \geq 10$). *#Easy*
- For uniform distributions, about 50% of the answers lie within $\left[\frac{1}{2}, \frac{3}{2}\right] \times P$. For Poisson distributions, that percentage is much higher. *#Easy*
- For Poisson, the maximum answer is relatively close to P, and the distribution's variance converges to P. So the maximum over variance ratio is bigger than 1. For the uniform distribution, on the other hand, the maximum is about $2P$, while the variance grows with P^2 and is much bigger for $P \geq 10$. Hence the ratio is smaller than one. *#Easy*
- Another approach consists of measuring the sample's distance to the most likely Poisson and the most likely uniform distribution (or all of them, if you tune your code a bit). For Poisson, the parameter can be estimated by the mean or variance, for the uniform by the maximum answer, augmented by the average gap between answers. A great choice of a distance metric is provided by the G-Test: $G = 2\Sigma_i \log\left(\frac{O_i}{E_i}\right)$, where O_i is the observed count in a cell and E_i is the expected count. *#Easy #Hard*

- The sigma (square-root of the variance, i.e., the standard deviation) of a uniform distribution converges to $\sqrt{3}$ of the distribution's maximum. Simulating many sets (millions) of 250 samples drawn from uniform distributions shows that the ratio of sigma to the maximum value does not exceed 2. One may approximate the Poisson distribution by a normal one, a reasonable assumption. For such a distribution, the largest element lies (with a 50% chance) at about 3 sigma, and the probability for it to be below 2 sigma is about $(95.45\%)^{250}$. That is about one chance in a million – so be brave and take the shot, this will pass! *#Easy #Hard*

What is a nice try, but is just not good enough

- Recenter the distribution around zero and fold the negative axis onto the positive one. Now let M be the maximum sample value. For uniform distributions, there are about as many samples within $\left[\frac{M}{2}, M\right]$ as within $\left[0, \frac{M}{2}\right]$. For Poisson distributions, the ratio of sample counts is much lower. The observation is nice and the tendency is observable, but randomness on samples as small as 250 has it that there exist exceptions to the trend, and no clear distinguishing cut exists for the distributions of the test-set.

- The distance from the smallest to the largest sample is about twice the mean value for uniform distributions. It is expected to be less for Poisson distributions. This fails for small populations, where the Poisson tails extend relatively far out and are non-negligible compared to a uniform distribution.

- Inspired by the G-Test curve-fitting approach, one may be tempted to use the χ^2 distance metric $\chi^2 = \Sigma_i \frac{(O_i - E_i)^2}{E_i}$. This metric however penalizes large errors too much and fails on our testset. Pearson introduced this metric as the second-order Taylor expansion of the G-Test at times when it was unduly laborious to calculate log-likelihood ratios.

- Using the $L1$-norm to compute the distance to a good Poisson and a good uniform distribution is not considered good enough either, sorry.

E. How many marmots?

Big math, intuition or try-and-see, but one thing is sure: Using an estimator with small bias and low variance puts you in good shape.

The Poisson distribution has flat tails, so the mean is a great estimator for its population. On the other hand, for the uniform distribution...

- The mean converges to the population P for a sufficiently large sample. However, the convergence is too slow and 250 answers come with too large a variance to fall into the acceptable tolerance interval. The fat tails of the uniform distribution are at the origin of this problem.

- So how about the median? Those who tried it out know that it does not make things any better!

- Half of the largest answer is the (biased) maximum likelihood estimator, and it passes! *#Medium*

- The upper tail's unlikeliness to be sampled equals the lower tail's unlikeliness. Hence the sum of the maximum and minimum sample values forms a good estimate of the mean. What misses at one end also misses at the other. This compensates for the previous approach's bias, but increases the variance. It also passes! *#Medium*

- The least variance estimator is the maximum element, augmented by the average gap between answers. This compensates the bias. It passes! *#Medium*

- Although the previous estimator is unbiased and has least variance, it is skewed: Underestimations are less likely than overestimations, but their aberration is all the more severe. So, to be centered even better on the symmetric bounds, we may multiply the previous estimator by 1.025. This introduces a bias, but enhances the worst case. It passes! *#Medium*

G-I. Fake News

Author: **Damian Straszak**

G. Easy

The most elegant solution, which works in $O(n)$, is to locate the first `h` in the string, then the first `e` located after that, then the first `i` located after that, then the first `d` located after that, and then the first `i` located after that. Other approaches are possible as well, as long as your algorithm is not $O(n^4)$ or $O(n^5)$, which is unfortunately not going to pass in time.

H. Medium

This proved to be a challenging constructive-algorithms task. The first natural idea is to have a pattern p like `abcd` and a text s like `aaabbccccccddd`, which gives $3 \cdot 2 \cdot 6 \cdot 3$ subsequence-occurrences, but unfortunately this approach seems unable to generate numbers which cannot be written as a product of small integers (large primes are particularly problematic).

In general, in tasks like this, we like to be able to perform two incremental operations: to get from a pair giving k occurrences to a pair giving $2k$ occurrences, and to get from a pair giving k occurrences to a pair giving $k+1$ occurrences. Then, by performing $O(\log n)$ such operations we would be able to generate any number. Going $k \to 2k$ is rather easy, provided we haven't used up all the letters yet: add a new character to the end of the pattern and add two occurences of this character to the end of the text. But, again, we don't know of any easy way to go $k \to k+1$...

It is, however, possible to go $k \to 2k+1$ and $k \to 2k+2$ if we use pattern-text pairs of a special form. Namely, we will always guarantee that the text s begins with the pattern p, and the pattern p will be of the form $p =$ `abcd...`

For $k = 1$, we can just have $s = p =$ `a`. Then:

- To go $k \to 2k+1$, suppose that for k we have a pair (s, p) with $s = pu$. Then generate $p' = p$`x`, where `x` is a new letter, and $s' = p$`x`u`xx`.
- To go $k \to 2k+2$, generate the same p' and take $s' = p$`xx`u`xx`.

We leave it to the reader to see that this construction is correct. (For example look at $k \to 2k+1$. The first `x` in p`x`u`xx` is only useful for matching the first occurence of p`x` at the beginning; if this `x` is not used, then we are reduced to having pu`xx`, which clearly has $2k$ occurences of p`x` since pu has k occurences of p.)

There are many other possible solutions. For example, some contestants have s always equal to `aaaaaaaab` and p being a carefully chosen mixture of `a`'s and `b`'s.

I. Hard

This is a task about suffix structures, and you can use your favorite one: suffix array, suffix tree, or suffix automaton.

We first describe a solution using the suffix array (and the LCP structure) and subsequently sketch the idea on how to apply suffix trees.

Solution using suffix arrays. We first build the suffix array together with the LCP structure (it is simplest to use the $O(n \log^2 n)$ algorithm).

Let us consider drawing all the suffixes sorted lexicographically (as in the suffix array). Look at a single letter (more precisely, letter occurrence) in that drawing (there are $n(n+1)/2$ of them in total, where n is the length of the string). For example look at the last `c` here:

```
...
aaaa
```

```
abcde
abcdf
abcx
...
```

If we give this letter a charge of k units, where k is the number of strings above which agree with the string we are looking at up to at least the position of our letter (in this case, that would be 3, since this is the third suffix beginning with `abc`; we are counting this suffix as well), then the sum of all charges (over each letter-occurrence) is almost what we want. Namely, we are going to return $2 \cdot (\text{that sum}) + n(n+1)/2$. It may require some thought to see that this is the right answer.

Therefore we can take a pass over the suffix array (from top to bottom), while maintaining a list of charges of all the letters in the last suffix. This list needs to be a data structure which supports the following operations:

- get the sum of all entries,
- for a given k, set all entries with indices larger than k to 0, and increment all entries with indices at most k.

This structure is not hard to implement using a queue (plus a single counter), so that both operations work in $O(1)$ amortized time. Now we go from top to bottom and, for each row of the suffix array, invoke the second operation with k being the length of the longest common prefix of the two current suffixes, and then invoke the first operation and add the result to our sum.

Solution using suffix trees. The solution using suffix trees is conceptually much simpler, though it requires implementing and dealing with suffix trees, which is often considered not as straightforward as suffix arrays.

Let us start (as usual) by describing what we would need to do given a structure called suffix **trie** – a trie built of all suffixes of the input string. Note that every node in this trie corresponds to exactly one substring of the input string. Moreover, for every node v, the number of occurences of the corresponding string is equal to the number of leaves in the subtree rooted at v. This suggests a straightforward solution. Traverse the trie using DFS and precompute the number of leaves in all rooted subtrees. Then, simply output the sum of squares of all these numbers!

The only "detail" which makes this solution not quite efficient is that the suffix trie can have size quadratic in the size of the input string. This is the reason why one uses suffix **trees** in general – the long paths are compressed into single edges, which allows us to run essentially the same algorithm and make it efficient – linear in the number of nodes of the suffix tree and hence linear in length of the input string!

J-K. Send The Fool Further!

Author: **Slobodan Mitrović**

J. Easy ($O(n)$ by DFS)

Heidi starts at vertex 0, and visits some friends (i.e., some vertices), so that no vertex is visited more than once. Now, if Heidi is currently visiting friend v, and v has a neighbor u that is not yet visited, can it happen that the worst case for Heidi is not to visit u and instead end the travel at v? Since the travel costs are strictly positive, it is always more costly if Heidi is sent to u compared to stopping her travel at v.

Also, observe that if u if a leaf, then u has no unvisited neighbor it can send Heidi to. Hence, in u is a leaf, once Heidi goes to u from v her journey stops. So, in the worst case, Heidi's journey would start at the root and end at some leaf.

This observation gives rise to the following approach: find the cost of a most expensive root-leaf path. This can be obtained in $O(n)$ using, say, DFS.

K. Medium ($O(n$ log $n)$ by DP)

Recall that we are given a rooted tree. By children of a vertex we understand its neighbors away from the root.

Let v be a vertex, and p_v be the first vertex on the v-root path. Clearly, the first time Heidi visits v it must be through p_v. Let $u_1, u_2, \ldots, u_t$ be the children of v that Heidi visits. Assume that after visiting v Heidi visits p_v again, i.e. she goes back to p_v from v at some moment. Then, there is a worst case scenario in which she goes to p_v from v only **after** having visited all the t neighbors $u_1, \ldots, v_t$. It is not hard to see this – as, no matter how many time she crosses an edge, she pays the cost only once.

This implies that in the worst-case scenario, after Heidi visits v along with some of its children and goes back to p_v, she will never come back to v again.

Now, there are two cases of what can happen when Heidi comes to v from p_v: (1) she goes back to p_v at some point; or, (2) she finishes her walk somewhere in the subtree rooted at v. In the first case, Heidi can visit at most $k-1$ of the children of v. In the second case, Heidi can visit up to k children of v.

Let us define $DP[v][true]$ to be the maximum cost Heidi has to pay for visiting the subtree rooted at v (while visiting every vertex at most k times) assuming she has to go back to p_v. So, this covers the first case. Similarly, we define $DP[v][false]$ to be the maximum cost if she does *not* go back to p_v.

Again, let $u_1, \ldots, u_t$ be the children Heidi visits from v (assuming we already know them). Then

$$DP[v][true] = \sum_{i=1}^{t} DP[u_i][true],$$

and

$$DP[v][false] = \max_{j=1\ldots t} \left(DP[u_j][false] + \sum_{i=1\ldots t,\text{ and } i\neq j} DP[u_i][true] \right).$$

We can rewrite the last expression as

$$DP[v][false] = \max_{j=1\ldots t} \left(DP[u_j][false] - DP[u_j][true] + \sum_{i=1}^{t} DP[u_i][true] \right).$$

But how to obtain $u_1, \ldots, u_t$? To compute $DP[v][true]$, the vertices $u_1, \ldots, u_t$ are simply the t children having largest $DP[u_i][true]$ values. To compute $DP[v][false]$ we iterate over all the children of v as a candidate for u_j. Once we fix u_j, we want to maximize the rest under the max. One way to maximize the summation under max is to choose the $t-1$ children of v having largest $DP[u_i][true]$. A naive implementation would result in time $\Omega(d_v^2)$, where d_v is the number of children of v. Such a running time would be too slow, as a vertex may have degree $\Omega(n)$. However, there are only two cases that we should consider:

- for a given u_j, u_j is not among the $t-1$ children of v having lagest $DP[u_i][true]$, in which case we can simply precompute and use the sum of those $t-1$ largest $DP[u_i][true]$ values to maximize the expression for fixed u_j;
- u_j is among the first $t-1$ children of v having largest $DP[u_i][true]$, in which case we precompute the sum S of t largest $DP[u_i][true]$ values, and get our answer as $S - DP[u_j][true]$.

Overall, we first sort $DP[u_i][true]$ values, preprocess/precompute them as described by the two cases, and after that for every u_j compute the expression under the max in $O(d_v)$ time. Sorting takes $O(d_v \log d_v)$ time.

Finally, we return $DP[0][false]$, which can be computed in total $O(n \log n)$ time.

L. Hard

Let E_v be the expected cost Heidi has to pay if she starts from vertex v. By the defintion, if v is a leaf, then $E_v = 0$. If v is not a leaf, then we have

$$E_v = \frac{1}{d_v} \sum_{\text{neighbor } u \text{ of } v} (E_u + c_{u,v}).$$

And we are interested in E_0.

The set of equations above is a system of linear equations.

If we grab some off-the-shelf general solver, e.g. Gaussian elimination, it might not work in time. In particular, applying standard Gaussian elimination naively will work in $O(n^3)$ time. However, if we apply Gaussian elimination very carefully in this case, by eliminating variables that correspond to leaves, we obtain the running time of $O(n \log(10^9 + 7))$, where the log factor comes from obtaining number inverses. More precisely, we eliminate a leaf, and prune the graph by removing the eliminated leaf. Note that this might result in some new vertex becoming a leaf. Then we take another leaf, and repeat the process as long as the graph contains a vertex.

What was the weird determinant comment about? Well, as we run Gaussian elimination, we need to perform division (which we do by computing number inverses modulo $10^9 + 7$). One could worry that at some point we would need to invert a number which is not really zero, but zero modulo $10^9 + 7$. However, the guarantee that the determinant of the matrix that defines the linear system of equations is nonzero implies that this will not happen (regardless of the order in which we eliminate vertices / variables). So we can just perform all arithmetic operations in the field modulo $10^9 + 7$.

M-O. April Fools' Problem

Author: **Jakub Tarnawski (easy), Jakub Pachocki (hard)**

M. Easy

Return the sum of k smallest numbers from the input.

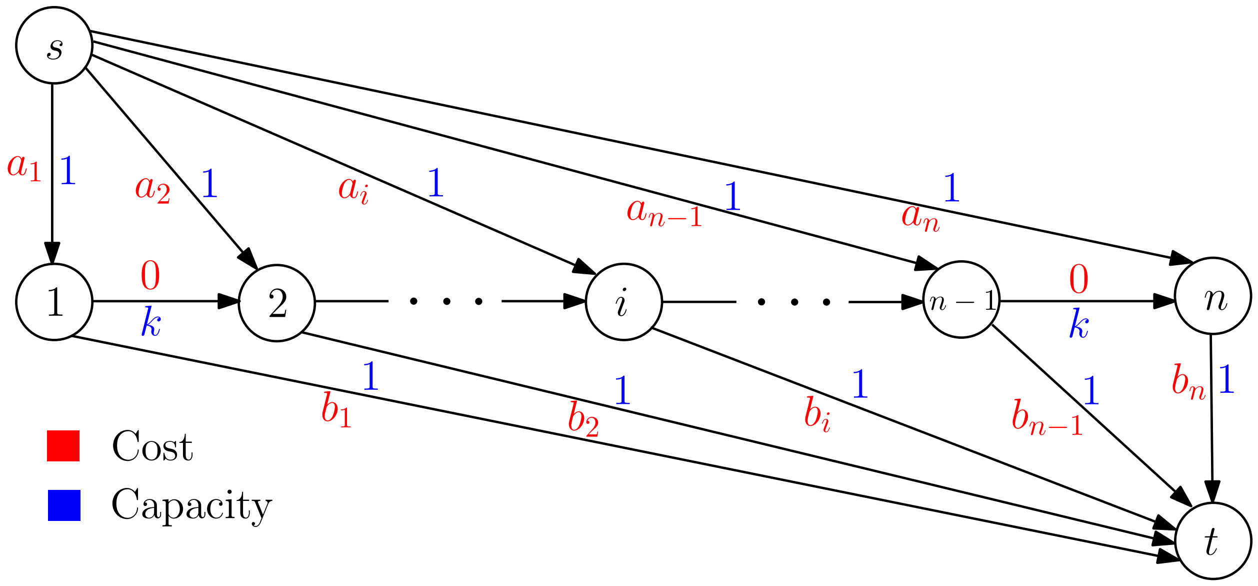
N. Medium

The problem can be solved in polynomial time using dynamic programming. However, it seems hard to do it in $O(n^2)$...

One way to see the problem is, again, as a minimum-cost maximum-flow instance. Consider the following graph:

- The graph has $n + 2$ vertices: one per each day, plus a source s and a sink t.
- For every day $i = 1, ..., n$ add three edges:
  - from the source to i with capacity 1 and cost a_i,
  - from i to the sink with capacity 1 and cost b_i,
  - from i to $i + 1$ (unless $i = n$) with capacity k and cost 0.

See below for an illustration of the network. It is not hard to see that a minimum-cost flow of value exactly k in this network corresponds to a solution to our problem. An off-the-shelf min cost max flow algorithm will take time $O(nm)$ (plus possibly some log factors) and fit in the time limit.



One can also model the problem as an $n \times n$ bipartite graph and try to find a minimum cost matching of size k. However, even a fast implementation of the Hungarian method will likely time out.

Due to the simple structure of this graph, it is also possible to simulate the min-cost flow algorithm in an easier way (or to come up with an algorithm which does essentially that, without having considered the flow network :)).

O. Hard

We describe the solution idea used both by the judge solution and the single team who solved this problem during the onsite contest. However, in the online mirror, two teams who have solved this problem seem to have done so in a much simpler way...

Anyway. The idea is to simulate the min-cost flow algorithm in a much more complicated, but faster way :) Namely, we want to simulate one phase (which wants to find a minimum-cost path from the source to the sink in the residual graph) in time $O(\log n)$.

In each phase/iteration, the MCMF algorithm wants to find a minimum-cost path (where it will send one unit of flow) from the source to the sink. It will pick some edge (s, i) and some edge (t, j), and then add flow on edges of the central path (middle edges) between vertices i and j. This is done in the residual graph; that is:

- if $i \leq j$, then this is always legal (the middle edges have some remaining capacity because we have placed fewer than k paths so far),
- if $i > j$, then the middle flow will go back (left on the picture); therefore this is only allowed if there is at least one unit of flow (going right) on each middle edge between j and i.

(After the ℓ-th iteration, the algorithm will have found the min-cost solution to the problem with $k = \ell$. One consequence of this is that our solution (set of days where problems are prepared and printed) for k is always a subset of our solution for $k + 1$.)

So, in one iteration, we should pick i and j (either any $i \leq j$ or legal $i > j$) so as to minimize the cost of the added path (which is $a_i + b_j$). Then we should add flow on the path. To do so, we need to maintain the current flows on the middle edges.

In our solution, we use a binary tree structure on the interval $[1, n]$ for this. The structure should allow the addition of flow on a path and finding the minimum-cost possible path. This tree, though not that unusual, is rather complicated in design (our implementation stores 13 integer values per node and uses lazy updates), and so we omit its details. It needs only $O(\log n)$ time per operation.