Problemset discussion

Santa

December 21, 2014

A Claussian

In this problem for a given string s we are supposed to cout the number of occurences of the word `Santa`. You can do it by either a simple loop over s or by using some library function. For example, in Python the whole solution can be coded in a single line!

```
print raw_input().count("Santa")
```

B Presents

If there are two bags of the same size, the answer is `YES` – Santa can just leave one in each zone. Otherwise, the answer is `NO`. To see this, take any allocation, and consider the largest bag that Santa gave away; suppose zone 11 received it, and let its size be 2^k. Then the total number of presents received by zone 12 is at most $2^{k-1} + 2^{k-2} + ... + 2 + 1 = 2^k - 1 < 2^k$.

C Chessboard

Solution: put rooks on cells $(1,1), (2,2), (3,3), ..., (k,k)$, where k is the largest number such that the k-th line contains at least k cells (that is, k is the largest number such that the cell (k,k) exists). We can see that this covers the chessboard (each cell is either in column 1 to k or in row 1 to k), and that it is not possible to use less rooks, because covering the sub-chessboard $\{1, ..., k\} \times \{1, ..., k\}$ requires k rooks.

D Santa's Bag

This problem can be recognized as a version of knapsack packing. We can solve it by dynamic programming. Define:

$$dp[i] = \text{ minimum value of a bag of total size } i$$

for $i = 0, 1, ..., m$. The answer to the problem is $dp[m]$. We compute those values in the order of increasing i using the following formula:

$$dp[i] = \min\{dp[i - s_j] + v_j : j = 1, 2, ..., n\}.$$

Computing one such value is done by iterating over all n items, so it takes $O(n)$ time. The total time is thus $O(nm)$.

E Cameras

For each square, just enumerate the number of its sides which do not border another square. To do this, one can first take a pass through the data, inserting the squares' coordinates into a data structure such as a set, a hash-set, or an array (and sorting that array later), and then take another pass: for every square (x, y), find out how many of the squares $(x + 1, y), (x, y + 1), (x - 1, y), (x, y - 1)$ belong to the data structure.

F Sequence

There are at least two different ways to solve this problem efficiently. Both are applications of the dynamic programming paradigm. The first idea was used by all three teams who were able to solve the problem during contest time, and it involves computing medians of all subsegments. However, we are going to present a different solution here; we believe it is a little simpler, although it may not be the first one that comes to mind.

Our crucial observation is that there exists an optimal sequence where all elements come from the original sequence. Indeed, if we consider a maximal subsegment $b_i, b_{i+1}, ..., b_j$ of equal values x in the sequence b, then taking x to be the median of the sequence $a_i, ..., a_j$ minimizes the cost. So we can afford to keep the last-added element in our dynamic programming state as we build the sequence b. Thus we define:

$$\begin{aligned} dp[i][k][j] = {} & \text{the minimum cost of turning the prefix } (a_1, ..., a_i) \text{ into} \\ & \text{a } k\text{-Laussian sequence } (b_1, ..., b_i) \text{ with } b_i = a_j \end{aligned}$$

where $0 \le i \le n$, $0 \le k \le K$ and $1 \le j \le n$. We can observe the following recurrence:

$$\begin{aligned} dp[i][k][j] = \min(& dp[i-1][k][j] + |a_i - a_j|, \\ & \min\{dp[i-1][k-1][j'] + |a_i - a_{j'}| : j' \in \{1, ..., n\}\}). \end{aligned}$$

Since there are $O(n^2k)$ states and each of them is computed in $O(n)$ time, this gives us an $O(n^3k)$ algorithm. Such an algorithm is unfortunately too slow. However, we can easily speed it up by computing an auxiliary array:

$$aux[i][k] = \min\{dp[i-1][k][j'] + |a_i - a_{j'}| : j' \in \{1, ..., l\}\}.$$

It allows us to compute a single $dp[i][k][j]$ value in only $O(1)$ time! In total, our running time becomes $O(n^2k)$, which is fast enough to pass.

(We also need to be careful with memory usage – the full $O(n^2k)$ array is too large to store. But we need only store two "rows" of this array (for the current and the previous i), making the memory usage $O(nk)$.)

G Reind Air

We can split all the vertices of the graph into two categories, articulation and non-articulation points, and handle them separately.

Non-articulation points. If a vertex v is not an articulation point, then the number of pairs that get separated by removing v from the graphs are only the $n-1$ pairs of the form (v, u).

Articulation points. Recall that there is a DFS-like algorithm that finds all the articulation points in linear by computing the low function. Let T be the rooted DFS tree computed by this algorithm. Given vertex u, by $T(u)$ we denote the subtree of T rooted at u.
Let v be an articulation point, and p be its parent in T (if v is the root of T, then $p = null$). Let $u_1, u_2, \ldots, u_k$ be the children of v. Note that none of the children equals p. Then, it can be easily seen that by the removal of v the graph splits into the following maximal connected components:

- for each vertex u_i with $low[u_i] > v$, $V(T(u_i))$ is one maximal connected component;
- the rest of the vertices are in a single maximal connected component.

So, once we identify an articulation point, by traversing its children in T we can identify components $C_1, C_2, \ldots, C_p$ the graph will split into after removing v. (Note that in general p might be different than k.) Now, every two vertices w and z such that $w \in C_i$ and $z \in C_j$, for $i \neq j$, add 1 to the number of pairs that get disconnected by the removal of v. Therefore, the total number of pairs can be computed as $\sum_{i<j} |V(C_i)| \cdot |V(C_j)|$. There are many way to compute this

summation in $O(p)$ time, once we have sizes of the components, and one of them relies on the following observation

$$\sum_{i<j} |V(C_i)| \cdot |V(C_j)| = \frac{(\sum_i |V(C_i)|)^2 - \sum_i |V(C_i)|^2}{2}.$$

This step can be computed in a linear time of the degree of v, and thus overall for all the vertices this step takes $O(n+m)$ time.

However, to compute this quantity we need to know $|V(C_i)|$ for each component. But, as $V(C_i) = V(T(u_j))$ for some j, knowing $|V(C_i)|$ is the same as knowing $|V(T(u_j))|$, and the latter can be computed easily during the DFS traversal, e.g. $|V(T(v))| = 1 + \sum_{i=1}^{k} |V(T(u_i))|$.

In addition, as in the non-articulation point case, we have to count all pairs of the form (v, u), but there are simply $n-1$ of them.

Each of the aforementioned steps takes linear time, so the whole algorithm takes $O(n+m)$ time.

A note on articulation points can be found at
https://courses.cs.washington.edu/courses/cse421/04su/slides/artic.pdf.

H Catching Santa

This task turned out to be easier than F or G for experienced participants, as its hardness comes mostly from requiring some background in algorithms.

The problem to be solved is called the minimum vertex s, t-cut problem (here, s and t are the school and the station). The first step is to reduce it to a minimum capacity edge s,t-cut problem instance (question: what is the minimum total weight/capacity of a set of edges whose removal would disconnect s and t?) as follows:

- transform every vertex v into two vertices v_{in} and v_{out} and a directed edge $(v_{\text{in}}, v_{\text{out}})$ with capacity 1,
- transform every undirected edge (v, w) into two edges $(v_{\text{out}}, w_{\text{in}}), (w_{\text{out}}, v_{\text{in}})$ with capacity ∞.

In other words, we "stretch out" every vertex to allow its removal, and disallow the removal of any "real" edge. Now, thanks to the Min-Cut Max-Flow Theorem, solving this problem amounts to finding the maximum $s_{\text{out}}, t_{\text{in}}$-flow in this graph. This can be done using any standard maximum flow algorithm.