Central European Olympiad in Informatics 2017

July 2017, Ljubljana, Slovenia

Problem Solutions - Day 2

Building Bridges

Tomaž Hočevar

Let's first slightly change the problem to focus only on the pillars that support the bridge and not the other ones which should be destroyed. The overall cost consists of the cost due to differences in heights plus the cost of destroying unused pillars. The latter one is equal to the sum of all destruction cost minus the cost of those that are retained.

Dynamic programming. Consider a subproblem of building the bridges such that the last one ends on pillar i. Let $f(i)$ represent the minimum cost of solving this subproblem. The last bridge in the solution will span from some pillar $j < i$ to i, which results in the formula:

$$\begin{aligned} f(1) &= -w_1 \\ f(i) &= \min_{j<i} \left(f(j) + (h_j - h_i)^2 - w_i \right) \end{aligned}$$

A dynamic programming solution has to solve $O(n)$ subproblems, spending $O(n)$ time on each.

Two pillars. Can we do better if we know that the optimal solution consists only of two additional pillars besides the first and the last? If there's only one additional pillar, we can simply try them all. For the case of two additional pillars, we'll try all second pillars i and for each of them choose an optimal first pillar j.

The contribution of the first pillar to the overall cost is $(h_1 - h_j)^2 + (h_j - h_i)^2 - w_j$. Let's ignore w_j for now (assume $w_j = 0$). Intuitively, we would want to choose h_j close to the average of h_1 and h_i. Calculating where derivative equals zero confirms our intuition. Therefore, we are interested in only two pillars — the largest among those smaller or equal to $(h_1 + h_i)/2$ and the smallest among those larger. As we try different pillars i from 1 to n, we can maintain a tree structure of pillars $j < i$ ordered by their height. This allows us to find the two we need in $O(\log n)$. For example, we can use the set data structure from C++ for this purpose.

All that's left is to handle the term w_j. Their limited range of values is helpful in this case. We can maintain separate tree structures for every value of w_j and look for the optimal pillar j for every possible value of w_j. The time complexity is $O(na \log n), a = \max |w_i|$.

Convex optimization. Let's rewrite the recursive definition from the dynamic programming solution.

$$f(i) = \min_{j<i} \left(A(j) \cdot h_i + B(j) + C(i) \right)$$

$$A(j) = -2h_j \quad B(j) = h_j^2 + f(j) \quad C(i) = h_i^2 - w_i$$

Note that C is a constant term that depends only on i and is the same regardless of the choice of j. If the term $A(j) \cdot h_i$ weren't present, we could simply maintain the best choice of j as we compute values of $f(i)$ for increasing i. Term h_i in the product complicates things. We can consider every pair $A(j)$ and $B(j)$ as a slope and y-intercept of a line. The problem of finding the best j reduces to finding the lowest line at $x = h_i$. To do this in $O(\log n)$ we'll maintain a lower envelope of a set of lines. The data structure has to be dynamic — it should support insertion of a new line in $O(\log n)$. Note that the lines comprising the lower envelope are ordered by their slope. Therefore, we can maintain a tree structure of these line segments. It allows us to quickly find the optimal line for a given x as well as insert a new one. We can abuse C++'s set for this purpose. This optimization is often called the convex hull trick in relation to dynamic programming and solves the problem in $O(n \log n)$ time.

Chase

Vid Kocijan

What is the maximal difference in number of pigeons we can make by dropping some breadcrumbs while running through the park? It is obvious from description that park represents a tree where each statue is a node and each path is an edge.

Exhaustive search. The tree has $O(n^2)$ possible paths in it, each determined by its beginning and end. On a path we have $O(2^n)$ possible ways to drop the crumbs. The exhaustive search can simulate all the possible strategies in $O(n^2 \cdot 2^n)$ time complexity.

Greedy approach. To approach the problem better, we must first make some observations. Let us say Jerry is currently stationed in node v and arrived here from node t. Jerry will visit node w in the next step. Let us denote $N(v) = \sum_{(v,x)\in E} p_x$ number of pigeons in all the neighbours of the node v. The gain $g(v)$ from dropping a breadcrumb is $g(v) = N(v) - p_t$ regardless of where the other breadcrumbs were dropped.

If a node x is neighbour to the v and $w \neq x \neq t$, the observation is quite straightforward. If we drop the breadcrumb at node v, p_x pigeons will be added to the path Tom traverses, otherwise not. Pigeons from node t do not affect Jerry's path, nor do they affect Tom's path. Jerry already met them once (and won't meet them again, even if they moved to node v) and Tom will only meet them once regardless of their position. Pigeons from node w move to node v, which means they won't count toward a total number of pigeons Jerry met. They will count towards the total number of pigeons Tom met. The values p_x that counted toward gain $g(v)$ have not been changed from the start. Jerry has not yet reached these nodes or any of their neighbours. Hence the gain of dropping a breadcrumb at a node v only depends on the last visited node.

If the starting node of a walk is known in advance, all the gains $g(v)$ can be calculated in advance. Now all we have to do is determine an ending point in such way, that sum of top v gains will be maximal. This can be done with a depth first search by storing all the gains on the current path in priority queue and selecting top v. This approach takes $O(n \cdot v \cdot \log(v))$ time if we know the starting node or $O(n^2 \cdot v \cdot \log(v))$ if we have to try every possible initial node.

Dynamic programming. Let us root the tree in node i. Let us denote $c[i][j]$ maximal difference we can make by starting a path in node i and continuing in one of its subtrees, dropping at most j breadcrumbs in the subtree. Let us denote $b[i][j]$ maximal difference we can make by starting a path in a subtree of a node i and ending it in i, dropping at most j breadcrumbs in subtree or at node i. Values of these two can easily be computed from values of b and c of i-th children. k denotes the children of node i.

$$\begin{aligned}
c[i][0] &= 0 \\
c[i][j] &= \max_k(c[k][j], c[k][j-1] + g(k) - p_i) \\
b[i][0] &= 0 \\
b[i][j] &= \max_k(b[k][j], b[k][j-1] + g[i] - p_k)
\end{aligned}$$

The best path running through node i and its subtrees makes the difference $\max_{j=1}^{v} b[i][j] + c[k][v-j]$. Note that path must not begin and end in the same subtree. A way to achieve this is to temporarily store the best two values for each possible $b[i][j]$, $c[i][j]$ and in which subtree that path started/ended. The best solution which does not use the same subtree twice can be easily calculated from these. This approach takes $O(n \cdot v)$ time.

Palindromic Partitions

Mitja Trampuš

Let us think about the problem of creating an optimal (i.e. longest) chunking in terms of an iterative process of chipping off equal-sized chunks from both sides of the input string s.

Exhaustive search. How large should these chipped-off prefixes and suffixes be at each step? Naively, we can try chipping off every possible length k, then recursively figure out the optimal chunking for the remainder:

$$\text{solution}(s) = \max_{k \in 1..|n|/2} \begin{cases} 2 + \text{solution}(s[k:-k]), & \text{if } s[:k] = s[-k:] \\ 1, & \text{otherwise} \end{cases}$$

The expression above uses python-like notation: $s[:k]$ is the prefix of length k of s, $s[-k:]$ is the suffix, and $s[k:-k]$ is everything *but* the prefix and suffix.

Dynamic programming. This solution takes time exponential in the length of the string, and will only solve the easiest set of inputs. However, note that the input to *solution*() is fully defined just by its length. It follows that there are only $O(n)$ possible different calls to solution(), and also that we can pass that length, rather than the actual substring of s, to solution(). If we memoize the function outputs, this results in a $O(n^3)$ runtime. This (or the equivalent dynamic programming solution without memoization) solves the first two sets of inputs.

Greedy. We do not have to consider all values of k, however. A greedy approach, always chipping off the smallest possible chunk, gives optimal results. Let us sketch the proof (in the interest of space, notation is not fully formal).

Without loss of generality, assume that the greedy algorithm and the optimal algorithm differ in the *first* chunk they chip off the input string s: the greedy algorithm selects a chunk C, and the optimal algorithm selects a strictly larger chunk C^*, $|C^*| > |C^0|$. Let us denote the prefix/suffix as $C^* = CA$ and $C^* = BC$:

We consider two cases:

1. $|\mathbf{B}| \geq |\mathbf{C}|$.

$s =$	C^*		S' (i.e. remainder of s)	C^*	
$=$	C	A	S' (i.e. remainder of s)	B	C

   Equating $CA = BC$ gives us $C^* = CXC$, where X is possibly an empty string. So chunking the prefix/suffix C^* as $(C)(X)(C)$ yields a longer palindrome than the "optimal" chunking of (C^*). Contradiction.

2. $|\mathbf{B}| < |\mathbf{C}|$.

$s =$	C^*		S' (i.e. remainder of s)	C^*	
$=$	C	A	S' (i.e. remainder of s)	B	C

   Equating $CA = BC$ and taking into account $|B| < |C|$ gives us $C = BC^{(1)}$ for some prefix $C^{(1)}$ of C; note that $C^{(1)}$ is both a prefix and a suffix of C. We can keep applying the same logic to get $C = BC^{(1)} = BBC^{(2)} = BBBC^{(3)} = \ldots = BB\ldots BBC^{(k)}$, until $|C^{(k)}| \leq |B|$. $C^{(k)}$ is both a prefix and a suffix of C (by the same logic that $C^{(1)}$ was), and thus of C^*. This prefix and suffix do not overlap, because $|C^{(k)}| \leq |B| < |C|/2 < |C^*|/2$. So C^* can be written as $C^{(k)}XC^{(k)}$ for some X, and could be chunked more finely. Contradiction as in case 1.

Since both cases lead to a contradiction, we conclude that the greedy solution is optimal.

A straightforward implementation of the greedy approach solves the first three sets of inputs. It still takes $O(n^2)$ time because the string equality test (to determine whether a chunk of size k can be chipped off) takes $O(n)$.

Rolling hashes. As a final optimization step, we can speed up string comparison using rolling hashes. Let us denote the ASCII value of a character c with $A(c)$, and choose a prime p larger

than $A(\mathtt{z})$, e.g. $p = 131$. We can then define the hash of a sequence of characters $c_0c_1 \cdots c_m$ as $h(c_1c_2 \cdots c_m) = \sum_{i=0}^{m} A(c_i)p^i \mod M$, where M is another suitably large prime. As we grow our candidate prefix and suffix, looking for the first matching pair, we only perform the expensive string comparison when the hash values of the prefix and the suffix match. These hash values can be updated in $O(1)$ for each additional character, and strings can be compared in approximately $O(1)$ (neglecting the cases where hash comparison yields a false positive), thus roughly $O(n)$ for the input overall.

Deterministic time complexity. A greedy solution with naive string equality testing is not that bad as long as the chunks are small enough. On the other hand, we could process the entire remaining string to find the smallest chunk in $O(n)$. We can use the z-algorithm or KMP's failure function for this purpose. But that would again waste a lot of time if the smallest chunk turns out to be small. However, we can make a compromise. Choose a length threshold t and use naive equality testing for $k < t$. If that doesn't yield a solution, process the entire string. Worst case time complexity is $O((k^2 + n)\frac{n}{k})$. Choosing $k = \sqrt{n}$ gives a $O(n^{1.5})$ solution.

A deterministic linear-time solution also exists, although we are not aware of one that is practical to implement in a contest. Let us build a suffix array a and store for every suffix $s[i :]$ its index $p[i]$ in the suffix array. Assume that we have already chipped off i characters on each side of s, and that we are trying to determine if k is a good size for the next prefix and suffix; in other words, if $s[i : i + k] = s[n - i - k : n - i]$. The size of a chunk k defines a range $a[l] \ldots a[r]$ of relevant suffixes in the suffix array and k is a good size exactly when $l \leq p[n - i - k] \leq r$. The bottleneck of this solution is in adjusting the bounds of the range l and r when increasing k. Building a suffix tree to traverse and determine bounds l and r instead of binary searching over a suffix array leads to a linear-time solution.