SAPO Finals 2016 Solution Manual

Robert Spencer, Robin Visser and Thomas Orton

October 2016

Contents

1 Day 1

1.1 Consecutive Sums

This problem had a number of solutions which would award the three subtasks.

Subtask 1

In this subtask, for a given N we could simply try all values of S and T less than N and sum the numbers between them to check.

In big Oh notation this would take time $\mathcal{O}(N^3)$.

Subtask 2

We can improve by noting that

$$S + (S+1) + \ldots + T = \frac{T^2 + T - S^2 + S}{2}$$

and perform the above sum in constant time.

This was sufficient for the first two tasks.

Alternatively, we could perform a "line sweep." That is we start with $S = T = 1$ and move T on one if the sum between S and T is too small and S on one if it is too large. Updating the sum is then constant time, and we obtain a $\mathcal{O}(N)$ solution.

Subtask 3

For subtask 3 one needed a general approach.

First we observe that only powers of two are not attainable. Secondly, we find that in general there is a solution

$$S = \left\lfloor \frac{N}{2M} \right\rfloor - M + 1$$
$$T = \left\lfloor \frac{N}{2M} \right\rfloor + M$$

where M is the largest power of two dividing N. If S is less than zero, we can replace $S \to -S + 1$ as

$$-S + (-S + 1) + \ldots + (-1) + 0 + 1 + \ldots + S + (S + 1) \ldots = (S + 1) + \ldots$$

The inspiration for these equations comes from the observation that for odd numbers, half the number rounded up and down works. Ie if N is odd, we choose

$$S = \left\lfloor \frac{N}{2} \right\rfloor \qquad T = \left\lceil \frac{N}{2} \right\rceil$$

For numbers that are even, but not a multiple of four we divide by four and take two numbers on each side of that (eg for 14, we divide by four to get 3.5 and take 2, 3, 4 and 5). This pattern holds for higher powers of two as well.

However, a power of two itself can never be written as the sum of a string of consecutive integers. Proving this is as simple as observing that

$$2^k = \frac{T^2 + T - S^2 + S}{2} \iff 2^{k+1} = (T + S)(T - S + 1)$$

and that one of the terms on the right hand side is odd.

1.2 Beautiful Words

The solution to this problem involves dynamic programming. This is the concept of breaking down a problem into simpler sub-problems, then solving each of these sub-problems, from which we can then obtain the full solution. We consider our sub-problems to the problem as prefixes of the given word. We denote $\mathrm{dp}(p, c, t)$ as the beauty value for the first p characters of the given word S, where the pth character is c and where only t changes have been made.

To calculate $\mathrm{dp}(p, c, t)$, we can simply note the value of $\mathrm{dp}(p - 1, \alpha, t - 1)$ for some character α and then add on the bonus value of $\mathrm{bonus}(\alpha, c)$. Taking the maximum value out of all characters α gives us $\mathrm{dp}(p, c, t)$. We also note a separate case where α is equal to the character of s at position $p - 1$. In this case, there is no need to decrements t, since no character is being changed.

Hence, we can find the value $\mathrm{dp}(p, c, t)$ in terms of simpler values by noting that

$$\begin{aligned}\mathrm{dp}(p, c, t) = \max(&\max(\mathrm{dp}(p - 1, \alpha, t - 1) + \mathrm{bonus}(\alpha, c) \mid \alpha = \text{'a'}, , \ldots, \text{'z'}, \alpha \neq \mathrm{word}[p - 1]),\\ &\mathrm{dp}(p - 1, \mathrm{word}[p - 1], t) + \mathrm{bonus}(\mathrm{word}[p - 1], c))\end{aligned}$$

where our initial values are $\mathrm{dp}(0, i, j) = 0$. To calculate the solution, we simply take the maximum value out of $\mathrm{dp}(|S|, i, j)$ (since one cannot know what the final character should be or how many changes should be made).

Since there are $|S|Kc$ states to calculate and each state takes $\mathrm{O}(c)$ time to calculate, this therefore provides a solution which runs in $\mathrm{O}(|S|Kc^2)$ time (where c is the alphabet size, which in our case is $c = 26$).

Other solutions

One can also simply brute force every possible word of length $|S|$ and check if there are no more than k characters which have been changed. Simply taking the max beauty value out of all the possibilities provides us with the solution. This runs in $\mathrm{O}(c^{|S|})$ time.

For the case where $K = |S|$, one can simply note that there is no limit to how many characters may be changed. Hence, doing the same DP solution as noted above will work, although one does not need to keep track of the t parameter. Hence, a $\mathrm{O}(|S|c^2)$ DP solution is possible.

1.3 Lost Cartographers

This was the most open question of the day.

Possible solutions were:

- To head for a corner. Both parties ignore each other and just head for a prearranged corner where they will meet. This works for the square boards, but nothing else. This obtained approximately 20 points.
- Smart corners / wall hugging. In this case both parties also head for a corner but don't only blindly go up and left (for example) but skirt around obstacles. This obtained approximately 40 points.
- Matching up. Here, the "maps" discovered by the two parties are overlaid and a match is hopefully found, allowing a traditional search (A*/Djikstra/BFS) to bring the two parties together. However, time constraints meant that agents had to be clever with this.

2 Day 2

2.1 Armour

In this problem one tries all the valid starting points and selects the best.

A simple brute force calculation of the sums will get you 12 points.

If you observe that there must be repeated queries for the second subtask and don't recalculate them, you will get 48 points.

The observation that you don't need to perform the sum every time will get you full marks. Indeed, let s_i be the sum of all the values to the left of i. Then for a given M,

the value of the items from j to $j + M$ is just $s_{j+M} - s_j$. Thus one can try all starting positions in time.

An equivalent solution is to perform a "sliding window" where one updates the sum smartly.

2.2 Word Square

This is a simple brute force.

2.3 Marbles

Subtask 1

This can be solved with a brute force/case bash approach, or with some simple heuristics.

Subtask 2

For slash i, we can put marble i on the left and all other marbles on the right. This will cut all of the strings.

Since there are at most 1000 marbles, we will use at most 1000 slashes.

Subtask 3

This subtask can be solved with a divide and conquer algorithm (this approach can also be used on subtask 4).

Consider the recursive function $f(a, b, i)$ which cuts all the strings between marbles a to b. Here i is an auxiliary variable called the "level" of f, which will help us keep track of slashes.

First, f divides the marbles $\{a, \ldots, b\}$ into two groups, with marbles $\{a, \ldots, (a+b)/2\}$ on the left hand side and marbles $\{(a + b)/2 + 1, \ldots, b\}$ on the right hand side. Then f makes a slash between these two groups.

Now the only remaining strings between marbles $\{a, \ldots, b\}$ are those between marbles a to $(a + b)/2$ and those between marbles $(a + b)/2 + 1$ to b.Therefore f must call $f(a, (a + b)/2, i + 1)$ and $f((a + b)/2 + 1, b, i + 1)$ to cut all the remaining strings.

We can now simply call $f(1, n, 0)$ to make the necessary slashes. If we merge all cuts made on the same level into a single slash (which we can do, since different slashes on the same level don't use any of the same marbles), then the number of slashes that we make is equal to the largest level f reaches, which is $\lceil \log_2(N) \rceil$. Thus we can always solve this problem within $\lceil \log_2(1000) \rceil = 10$ slashes. The time complexity for this solution depends on how quickly one can find strings between sets of marbles, but it is possible to implement in $O(N \log(N))$ time with dictionaries (hash maps).

Subtask 4

We can reapply the solution for subtask 3 by noticing that we only need to consider the marbles which have string attached to them (at most $2M$), and so we can get a solution which is guaranteed to use at most $1 + \lceil \log(M) \rceil \approx 18$ slashes.

In general, the approach to solving this subtask is to try and cut at least half of the remaining strings at each slash so that the number of slashes needed is at most logarithmic in the number of strings. Playing around with heuristics to achieve this can bring the number of cuts needed down to 9 or fewer.

For example, here are two methods one could use. The first is a probabilistic argument. For each slash, simply randomly choose whether a marble is on the left or the right. The expected number of strings between the two groups is equal to half of the remaining strings. In other words, simply making random slashes can give very reasonable performance in this question. Trying a number of random solutions until a good solution is found should give a low number of slashes.

A second approach, which builds on this idea, and was implemented for the model solution, is the following: at each slash, group the marbles such that it is guaranteed at least half of the remaining strings are cut. We can construct such a grouping in linear time as follows: for each marble, in the order they are labeled, assign that marble to the left or to the right depending on whether it has more strings connected to marbles already allocated to the left or more strings connected to marbles already allocated to the right. For example, if we are adding marble i and marble i has more strings connected to marbles already allocated on the left, we will allocate marble i to the right. This algorithm maintains the invariant that at each step, at least half the number of edges between marbles which have been allocated either left or right will be slashed. Therefore, at least half of the strings will be slashed once all the marbles are allocated. Counting the number of left or right marbles that connect to marble i can be done in constant time by keeping a table of counts. This gives an $O(E \log(E))$ time solution. In practice, this solution performs better than the worst case scenario, which is why only at most 9 cuts are necessary for the particular test cases.

2.4 Paths

This is a standard problem in graph theory to find the shortest path from node 1 to node n in a weighted graph. However, we are also given triples a_i, b_i, c_i which we are forbidden to travel along. If no such triples were given ($K = 0$), we could simply use Dijkstra's shortest path algorithm to solve the problem. Otherwise, we consider the following modification to Dijkstra:

Consider storing d_{xy} which is the shortest tentative distance to node y where the previous node is node x, i.e. a path of the form $1 \to \cdots \to x \to y$. Then when we update the distance along some edge from y to z with weight w_{yz}, we simply update:

$$d_{yz} = \min(d_{yz}, d_{xy} + w_{yz} \mid (x, y, z) \text{ not forbidden})$$

The solution is therefore the same as Dijkstra, except that the above update is done for d_{yz} instead of d_z with $d_{11} = 0$ as the starting node. In order to efficiently check if (x, y, z) is forbidden, one can simply use a set or a map.

Since there are N^2 states to update and insert and remove operations for a priority queue are $\mathrm{O}(\log N)$, this gives a total run time of $\mathrm{O}(N^2 \log N)$

Other solutions

If $w_i = 1$, then we just have an unweighted graph and a standard depth-first search can be done to calculate the shortest path.

If the underlying shortest path algorithm used is the Bellman Ford algorithm, then the run time is $\mathrm{O}(N^2M)$.