Fall 2020 mBIT Standard Editorial

November 14, 2020

This editorial provides the intended solutions to each problem as well as accepted programs in each supported language. In some cases, the given programs may employ different algorithms than the one described in the editorial. For more complex problems, multiple solutions may be given, in which case there will be programs for each solution. Nevertheless, problems are likely to have solutions which are not covered here and we would be interested to hear about any such solutions the reader may devise.

Contents

1 Apple Pie 2

2 Double Trouble 3

3 Explorers 4

4 Banquet 5

5 Climbing Trees 6

6 Weights 7

7 Stone Piles 8

8 Number Game 9

9 Heating Rocks 11

10 Calendars 13

11 Cathedral 14

12 Gemstones 15

§1 Apple Pie

Solution 1

Claire has N apples and will need to buy $K \cdot 10 - N$ more apples. In order to do this, she must purchase $\lceil \frac{10K-N}{M} \rceil$ more baskets.

Time complexity: $\mathcal{O}(1)$

Solution 2

Keep adding M to the number of apples Claire has (representing buying baskets) until she has enough.

Time complexity: $\mathcal{O}(K)$

Problem: Ayush Varshney
Flavortext: Gabriel Wu
Editorial: Claire Zhang
Code: C++, Java, Python

§2 Double Trouble

Looping through the string, keep track of "runs", substrings composed of only one character. When you encounter a new character, check to see if the current run has a length of 2; if so, it should be ignored, but otherwise, it should be printed. The final output will consist of exactly the desired runs.

Time complexity: $\mathcal{O}(N)$

Problem: Gabriel Wu
Flavortext: Gabriel Wu
Editorial: Claire Zhang
Code: C++, Java, Python

§3 Explorers

We can simulate the paths of Zheng and Yang by keeping track of their current positions and the directions they are facing in. Zheng will initially face left. Each next region will be one region further in his horizontal direction, except for when that direction would bring him out of bounds, in which case he will always moves upwards and then invert his horizontal direction. Similarly, Yang will initially face downwards and always move one region further in his vertical direction, except for when that brings him out of bounds, in which case he will always move right and invert his vertical direction. At the start of each move (to include the first region), check if Zheng and Yang are at the same region.

Time complexity: $\mathcal{O}(NM)$

Problem: Gabriel Wu
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§4 Banquet

In this problem, it is fundamental that $\sum_{i=1}^{n} a_i = n$, which means that every plate must be allocated to a stack in the final state. Let stack i be the stack of plates in position i in the final position. We claim that it is optimal to move the first a_1 plates to stack 1, the next a_2 plates to stack 2, and so on, regardless of the positions of the stacks.

If we let s_x denote the position of the stack plate x moves to, then for every pair of plates at positions i and j with $i < j$, $s_i \leq s_j$. To prove the optimality of this strategy, we will assume that there exists a pair of plates at positions i and j such that $i < j$ but $s_i > s_j$ and show that it is just as good or better to swap the stacks they move to. There are three unique (possibly overlapping) cases for the relative positions of the plates and their stacks:

1. $i \leq s_j < s_i \leq j$. The original time for moving plates i and j is $(s_i - i) + (j - s_j) = ((s_j - i) + (s_i - s_j)) + ((j - s_i) + (s_i - s_j)) > (s_j - i) + (j - s_i)$.
2. $s_j \leq i < j \leq s_i$. The original time is $(s_i - i) + (j - s_j) = ((s_i - j) + (j - i)) + ((i - s_j) + (j - i)) > (s_i - j) + (i - s_j)$.
3. Exactly one of i or j is in between s_i and s_j, inclusive. Without loss of generality, assume i is in between, so $s_j \leq i \leq s_i < j$. The original time is $(s_i - i) + (j - s_j) = ((i - s_j) + (s_i - i) - (i - s_j)) + ((j - s_i) + (s_i - s_j)) = (i - s_j) + (j - s_i) + 2(s_i - i) \geq (i - s_j) + (j - s_i)$.

Therefore, an optimal solution would be to process the plates from left to right and calculate the time needed to move each plate to the leftmost incomplete stack.

Time complexity: $\mathcal{O}(N)$

Problem: Jeffrey Tong
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§5 Climbing Trees

We can observe that if Joe can visit a tree with height j immediately after a tree with height i, he can also visit all trees with heights between i and j. Thus, we can construct an algorithm as follows:

First, we sort the trees by height. Starting at the lowest tree, we move upwards to the next tree and keep track of how many meters we have ascended. If the next tree is not reachable (or if we are at the tallest tree), store the height climbed. Then repeat until we reach the tallest tree in the forest, making sure to overwrite the stored height climbed if we just climbed a greater distance in the current section.

Time complexity: $\mathcal{O}(N \log N)$

Problem: Claire Zhang
Flavortext: Evan Wang
Editorial: Evan Wang
Code: C++, Java, Python

§6 Weights

The grouping with the maximum difference between the centers of mass must consist of one group with the k lightest weights $(1 \leq k \leq N-1)$ and the other with the remaining weights. If this were not the case, by swapping some pair of weights to meet this condition, the lighter group would have a lower center of mass and the heavier group a higher one, leading to a greater difference.

To find the optimal split, k, and the difference, we can first sort the weights, compute prefix sums, and test all splits by computing the difference between the centers of mass.

Time complexity: $\mathcal{O}(N \log N)$

Problem: Gabriel Wu
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§7 Stone Piles

There are many ways for Gabe to move the stones in linear time and output complexity. One method is as follows:

Represent each pile as a stack for easy retrieval. This requires that the stones for each pile be read backwards.

1. Move all stones to pile 1. This takes at most N moves.
2. While pile 1 is nonempty, if the top stone is labeled 1 or 2, move it to pile 2. Otherwise, move it to its final stack. This takes exactly N moves.
3. While pile 2 is nonempty, if the top stone is labeled 1, move it to pile 1, and if it is labeled 2, move it to pile 3. This takes at most N moves.
4. Move all the stones labeled 2 that are on top of stack 3 back to stack 2. This takes at most N moves.

Because each step uses no more than N moves, the total number of moves used is bounded by $4N$ or $4 \cdot 10^5$ for $N = 10^5$; the bound in the statement is set higher to discourage reverse-engineering of the solution. In practice, it is usually possible to use significantly fewer moves than $4N$.

Time complexity: $\mathcal{O}(N)$
Output Complexity: $\mathcal{O}(N)$

Problem: Gabriel Wu
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§8 Number Game

Solution 1

Let $s[a:b]$ be the number represented by the substring of s from indices a to b, inclusive (with indices starting from 1). Let $|y|$ be the number of digits in y.

For any digit $0 \le d \le 9$, $0 \le d^2 \le d^3 \le 729$, meaning that each digit in x corresponds to between 1 and 3 digits in y. Thus, if a valid x exists, there must be some $1 \le k \le 3$ such that last k digits of y correspond to some d and the first $|y| - k$ digits of y also corresponds to a valid x. This recursive structure enables us to use dynamic programming.

Let

$$f(n) := \begin{cases} \text{The smallest digit } d \text{ such that } d^2 = n \text{ or } d^3 = n & \text{if } d \text{ exists} \\ -1 & \text{otherwise} \end{cases}.$$

We will compute an array dp such that for all $1 \le i \le |y|$, $dp[i]$ is the minimum integer k such that there is an x_i corresponding to $y[1:i]$ and x_i's last digit corresponds to the last k digits of $y[1:i]$, or -1 if no such k exists.

For convenience, we also set $dp[0] = 0$, representing that it is possible to reach the beginning of y using an empty x_0, and $dp[i] = -1$ for $i > 0$.

The transition is

$$dp[i] = \begin{cases} k & \text{if } dp[i-k] \ne -1 \text{ and } f(y[i-k+1:i]) \ne -1 \text{ for some } 1 \le k \le 3 \\ -1 & \text{otherwise} \end{cases}.$$

(If more than one k value is possible, return the first one found.)

Once we reach the end of y, if $dp[|y|] = -1$, we know that no such x exists and we output -1. Otherwise, we can work backwards to reconstruct x using dp since each index of dp gives the length of a substring for which a corresponding d exists, so it points to a previous index of dp for which we also found a valid x_i. Keep moving backwards through y until you reach the beginning.

Note: This solution always finds the value of x with the most digits because of the order in which we select $dp[i]$. By simply reversing this order, you can find the shortest value of x.

This solution is used by the C++ and Python code.

Time complexity: $\mathcal{O}(\log y)$

Solution 2

Let the set of digit strings in y that can correspond to a single digit in x be S. Since `49`, `81`, and `125` can each be decomposed into two shorter members of S, it is possible to construct x from all valid y's using only the set of corresponding strings $T = \{0, 1, 4, 9, 16, 25, 36, 64, 8, 27, 216, 343, 512, 729\}$. Dividing T into lists based on the strings' first digits produces the following:

First digit	Possibilities
0	0
1	1, 16
2	25, 27, 216
3	36, 343
4	4
5	512
6	64
7	729
8	8
9	9

Notice that none of the possibilities for a fixed starting digit is the prefix of another possibility with that starting digit with the exception of `1`. However, `6` must always be followed by `4` in a y with a corresponding x, meaning that any y starting with `16` will start with `164`. It cannot be required that the `4` be used as the start of a new number, as shown previously, so it suffices to show that the prefix can be created in two ways: `16` + `4` and `1` + `64` (the greedy method). Therefore, if it is possible to construct x, it is possible to do so with a greedy algorithm instead of dynamic programming by checking the first two digits of each chunk in y and using casework.

This solution is used by the Java code.

Time complexity: $\mathcal{O}(\log y)$

Problem: Gabriel Wu
Flavortext: Jeffrey Tong
Editorial: Jeffrey Tong
Code: C++, Java, Python

§9 Heating Rocks

Solution 1

Compute an array a such that $a_i = \max(X - T_i, 0)$, the number of degrees that rock i must be heated. The problem is now equivalent to finding the minimum time needed to reduce each element of a to zero by decrementing up to two elements each second.

Let k be the index of an element in a equal to the maximum element and let S be the sum of a. We must spend a_k seconds reducing a_k to zero. If $2a_k > S \iff a_k > S - a_k$, it is possible to achieve this lower bound: since a_k exceeds the sum of all other elements, decrement a_k and any other positive element of a (if such an element exists) a_k times.

Otherwise, $a_k \leq S - a_k$. Since we can decrease S by at most 2 in any second, a lower bound for the answer is $\lceil \frac{S}{2} \rceil$. We claim that $\lceil \frac{S}{2} \rceil$ is the answer in this case.

Proof. We will achieve the lower bound of $\lceil \frac{S}{2} \rceil$ seconds as long as we decrement exactly one element of a during at most one second (if S is odd) and two elements of a during every other second. If S is odd, $S - a_k \neq a_k$, so $S - a_k > a_k$, meaning that we can spend one second to decrement any element other than a_k to make S even while maintaining that $S - a_k \geq a_k$ and a_k is the maximum. It now suffices to prove that it is possible to solve this case for even S in exactly $\frac{S}{2}$ seconds.

Consider a representation of our strategy as a $\frac{S}{2} \times 2$ table such that all the rocks which we put in the r-th row are heated during the r-th second. The strategy satisfies the problem as long as we never put a rock in the same row of the table twice and we put the i-th rock in the table a_i times. To construct this strategy, we can simply start from the top of first column and fill in $a_i \times 1$ rectangles for the i-th rock going from $i = 1$ to n, breaking up the rectangle into two and wrapping around if it exceeds the bottom of the column; the construction for $S = 12$ and $a = [2, 5, 3, 1, 1]$ is shown as an example:

r	1	2
1	a_1	a_2
2		a_3
3	a_2	
4		
5		a_4
6		a_5

Notice that only one element can wrap around; if we call this element a_w, it remains to verify that the two pieces of a_w never overlap. However, this is easy to do, as $a_w \leq a_k \leq \frac{S}{2}$ and the pieces of a_w start from opposite ends of the columns, so they cannot overlap. □

Time complexity: $\mathcal{O}(N)$

Solution 2 (subtask 1 only)

For small N and X, it is also valid to greedily find the coldest two rocks and heat them during each second that the rocks are not both at X °C already. Each iteration can be done directly in $\mathcal{O}(N)$ or in $\mathcal{O}(\log N)$ with a priority queue; in both cases, this algorithm will pass subtask 1 but not subtask 2.

It can be proven that this strategy is optimal via an induction argument.

Time complexity: $\mathcal{O}(N^2X)$, $\mathcal{O}(NX\log N)$

Problem: Ayush Varshney
Flavortext: Gabriel Wu
Editorial: Jeffrey Tong
Code: C++, Java, Python

§10 Calendars

The brute-force approach of computing $dist(A, C)$ after rotating B $0, 1, 2, ...,$ and $N - 1$ times to the right runs in $\mathcal{O}(N^2)$, which is too slow. However, if we are able to determine by how much $dist(A, C)$ changes after each rotation in constant time, we can reduce our algorithm's time complexity to $\mathcal{O}(N)$.

First, compute $dist$ at the beginning. Let Δ denote the change in $dist$ that will occur after we rotate C one more time to the right if we pretend that the last number simply moves forward one more space instead of wrapping around. Δ may be computed at the beginning by noting the relative positions of each number in A and B.

During each rotation, we first need to add $\Delta - 1$ to $dist$ to account for the movement of the first $N - 1$ elements. Let y be the last element of C; we also add $(pos_A(y) - 0) - (N - 1 - pos_A(y)) = 2pos_A(y) - N + 1$ to $dist$ to account for the wrapping of y. Therefore, we add $\Delta + 2pos_A(y) - N$ in total to $dist$ for this rotation.

To compute the next Δ, we notice that it will change between rotations exactly when, for some $1 \leq x \leq N$, $pos_A(x) - pos_C(x)$ changes signs. This can occur in two cases: when $pos_A(x) = pos_C(x)$ and when $pos_C(x)$ wraps around from N to 1. Thus, if k is the number of x values for which $pos_A(x) = pos_C(x)$ after the rotation, Δ will change by $2k - 2$ (-2 since y's contribution to Δ will always switch from $+1$ to -1, except when $pos_A(y) = 1$, in which case the $2k$ immediately reverts it anyways).

Thus, if we precompute k for each rotation in a hashmap during precomputation and track the last number in C, we can recompute $dist$ after each rotation in constant time.

Time complexity: $\mathcal{O}(N)$

Problem: Maxwell Zhang
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§11 Cathedral

Since P is non-increasing, all subarrays of P must also be non-increasing. Because of this, each circulation of a subarray decreases the left-side sum and increases the right-side sum. Therefore, the difference in the left and right sums changes monotonically.

For each query, we can binary search for the number of circulations after which the array obtains minimum unevenness. After r circulations (we only need to check $0 \leq r \leq k-1$ by symmetry), the left sum is $\sum_{i=s+r}^{s+r+k-1} P_i$, and the right sum is simply $\sum_{i=s}^{s+2k-1} P_i -$ (left sum). We can obtain these sums in $\mathcal{O}(1)$ time by precomputing prefix sums of P.

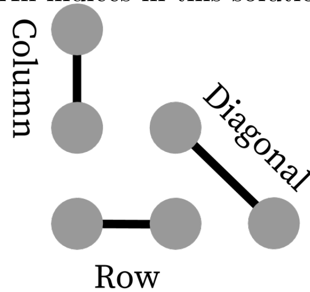
Time complexity: $\mathcal{O}(N + Q \log N)$

Problem: Claire Zhang
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§12 Gemstones

The maximum number of strings is $N-1$, attained exactly when one string of each length between 2 and N is used, which is always possible; thus, Maxwell must use all possible lengths. Notice that the string of length N must cover an entire edge of the triangle, leaving a subproblem with size $N-1$. Proceed with row-column DP.

If we rearrange the gems into a triangle with one side parallel to the x-axis and one to the y-axis, as shown, we can classify each string as being on a *row*, *column*, or *diagonal*. All indices in this solution start from 0.



Let $p(\text{“}row\text{”}, i, j, len)$ be the price of the string starting at gem j in row i with length len and define p likewise for “*col*” and “*diag*”. p can be calculated directly in $\mathcal{O}(N)$.

Define $f(r, c, len)$ as the maximum total price for the subtriangle with bottom-left corner at row r, column c, and side length len, and memoize f's results in an array dp. If

$$f(r,c,len) = \begin{cases} 0 & \text{if } len = 1 \\ dp[r][c][len] & \text{if } dp[r][c][len] \neq -1 \\ \max(& \\ \quad p(\text{“}row\text{”}, r, c, len) + f(r-1, c, len-1), & \\ \quad p(\text{“}col\text{”}, c, r-len+1, len) + f(r, c+1, len-1), & \text{otherwise} \\ \quad p(\text{“}diag\text{”}, r-len+1-c, c, len) + f(r, c, len-1) & \\ ) & \end{cases}$$

, $f(0, 0, N)$ will be the final answer, and as f is computed on each DP state (no more than $N \cdot N \cdot N$) exactly once, the computation will be done with $\mathcal{O}(N^4)$ time and $\mathcal{O}(N^3)$ memory.

To solve subtask 2, we must reduce the time complexity to $\mathcal{O}(N^3)$ by computing p in constant time. Let $row[i][j][k]$ be the price of the string in row i starting from gem j with length k, and define col and $diag$ analogously. These arrays can each be computed in $\mathcal{O}(N^3)$ so that string prices can be accessed in $\mathcal{O}(1)$ in the transition.

To solve subtask 3, only precompute the number of gems of each color in the prefixes of each row, column, and diagonal so that p can be computed via prefix sums, reducing the time and memory complexity of preprocessing to $\mathcal{O}(N^2)$. A final optimization that will probably be necessary is to rewrite dp and f to only use the lengths currently needed in the transition (len and $len-1$), reducing the memory complexity of our DP to $\mathcal{O}(N^2)$.

Time complexity: $\mathcal{O}(N^3)$

Problem: Gabriel Wu
Flavortext: Jeffrey Tong
Editorial: Jeffrey Tong
Code: C++, Java, Python