Deskhead Hackathon solutions

Medha Venkatapathy

January 14, 2025

1 Part A

1.1 General Idea

My mental model was that if there are n points, we can graph all valid **vector_s** solutions on an n-dimensional surface. I think the surface would be smooth but have many local minima. We could solve this problem by using packages in Python like Gurobi or PuLP, which efficiently solve linear programming problems.

To maximize the objective function, I used two different techniques: backtracking/DP and {}.

1.2 Backtracking/DP

Algorithm: The general idea is to explicitly try all the integer possibilities for each s_i up to some max_s. The dynamic programming (DP) table stores the best value of the objective function achievable from a pair (s_i, s_{i+1}). Then, we try all the possibilities for s_{i+2}, and the DP table gets updated accordingly. When we reach the index_answer (as defined in the code), we will have tried every single valid slack vector. At this point, we can pick the highest product in the final layer and backtrack to find the actual **vector_s** that caused this result.

This function takes as inputs a list of slopes and the max_s, ensuring the values don't grow arbitrarily large (this is only important for cases with small n).

Runtime: We initialize the DP table with s_1 and s_2. At each layer i, we must check $(\text{max\_s}+1)^2$ states for the next key (s_i, s_{i+1}). For s_{i+2}, there are another $\text{max\_s} + 1$ possibilities, resulting in a total complexity of $O((\text{max\_s} + 1)^3)$ per layer. If n is the number of points, there are $n - 2$ levels of the DP table to cover, giving an overall runtime of

$$O((n - 2) \cdot (\text{max\_s} + 1)^3).$$

Pros/Cons: This solution is guaranteed to find the best **vector_s** that maximizes the product in the objective function. Moreover, it can be easily modified to solve Part B by adding another constraint and efficiently iterating through all solutions.

However, if n is large or max_s is large, the DP method becomes extremely slow. In most practical applications, n will likely be large, so another technique may be better suited for these cases.

1.3 Two-Pass

Algorithm: The general idea here is to traverse **vector_s** in two passes: a forward pass to determine the maximum possible values for the slack, and a backward pass to determine the minimum possible values. These bounds are derived from the constraints given in the problem.

Conceptually, the first constraint implies that **vector_s** must be non-increasing or static. The second constraint limits the sum of consecutive $s[i]$ values to prevent them from becoming arbitrarily large.

With the calculated maximum and minimum bounds, the optimal s_i values can be selected. Assuming convexity, the optimal s_i lies closest to the boundary, which can be proved using a greedy algorithm. Intuitively, if a different s_i were chosen, an alternative solution for the objective function could be obtained by moving s_i toward the boundary.

Runtime: The forward and backward passes each run in $O(n)$ time, and selecting the optimal s_i values also occurs in $O(n)$ time. Thus, the overall runtime is

$$O(n).$$

Pros/Cons: This method is significantly faster and can easily scale to larger n. However, it relies on the convexity constraint for correctness.

Regarding the test cases, all returned a product of 1. However, for a larger number of points, the product is unlikely to remain 1.

All test cases are in the python file.

2 Part B

Because of the extra constraint, the overall objective function is no longer linear. There are packages, like Gurobi or a tool made by Google, that can handle these cases. In Part A, we used an efficient two-pass method. However, we cannot use the same approach here because we would need to ignore S. Instead, we could explore a Monte Carlo-like algorithm.

The idea is to first run the two-pass algorithm from Part A. If the sum of s_i is less than or equal to S, it is a valid solution for Part B. Otherwise, we would need to remove some slack from the vector, likely using a greedy algorithm to adjust the distribution. While this approach might not be optimal, it should generally perform well as an approximate algorithm. This method would be relatively efficient ($O(n)$) but would come with a bounded error margin.

For an exact solution, however, a dynamic programming (DP) approach can be used.

2.1 Algorithm

We have three constraints: the convexity constraint, the global sum constraint, and $s[i] \geq 0$. To handle these, we will use a 3D DP table. The three dimensions are: 1. The index (0 to num_points $-$ 2), 2. The sum of slack values so far, 3. The pair (s_i, s_{i+1}) (from Part A).

Each entry in the DP table stores a tuple consisting of the optimal product and the information needed to reach this point. We use the same logic and proof of correctness from Part A. First, we fill in all values in the DP table, use the final index to select the best product, and then backtrack to reconstruct the solution.

Filling in Values: Initially, we enumerate all (s_0, s_1) pairs that satisfy $m[1] + s_1 \leq m[0] + s_0$. For each valid pair, we update the corresponding DP table entry:

$$\text{dp}[0][s_0 + s_1][(s_0, s_1)] = ((1 + s_0) \cdot (1 + s_1), \text{None}).$$

Next, we iterate through the layers of the DP table, propagating values forward based on previously filled layers.

To find the optimal global product (deterministic), we look at all $\text{dp}[n-1][\text{sum}]$ entries for $\text{sum} \leq S$ (global constraint). We select the key corresponding to the largest product.

Backtracking: Once the optimal product is identified, we backtrack using the "pred" variable in the value of the highest key. This tells us the preceding pair (s_{n-3}, s_{n-2}). We repeat this process backward until we reconstruct the entire **vector_s**.

2.2 Proof of Correctness

The correctness of Part A's DP solution can be reduced to Part B's because Part A is a more general case of Part B (where S can be any number). Thus, we only need to prove correctness for Part B.

Base Case: For $i = 0$, we consider every possible (s_0, s_1) combination that satisfies the constraints $s_0 + s_1 \leq S$ and $m[1] + s_1 \leq m[0] + s_0$. Since no cases or combinations are left out, the base case is correct.

Inductive Step: Assume we have correctly accounted for all valid assignments $(s_0, \ldots, s_{i+1})$ for some i. Then, we can correctly assign $(s_0, \ldots, s_{i+2})$ because any valid s_{i+2} satisfying earlier_sum + $s_{i+2} \leq S$ will be included.

This ensures that every possible case is accounted for, and no cases are overcounted because each state in the DP table is unique. The algorithm is guaranteed to find the maximum product.

2.3 Runtime

If n is the number of slopes, there are $n-1$ layers to transition between. The DP table has at most $O(n \cdot S \cdot (\text{max\_s\_var} + 1)^2)$ states. Each state requires at most max_s_var operations, resulting in an overall time complexity of:

$$O(n \cdot S \cdot (\text{max\_s\_var} + 1)^3).$$

2.4 Test Cases

All test cases for all parts are provided in the Python files.

3

Thank you so much for this opportunity, I loved solving the questions.