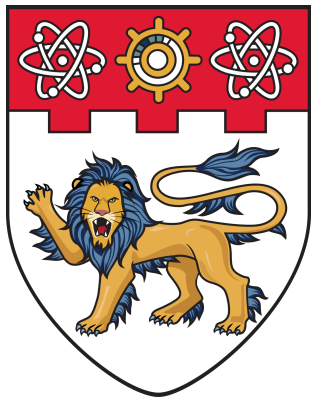
Nanyang Programming Contest 2025

Stage 4:

Lee Zong Yu, Pu Fanyi, Zhou Xuhang

Nanyang Technological University

11 October 2025



- Easy
  - Problem A: Card
  - Problem B: Glass Bridge
- Medium
  - Problem C: Sudoku
  - Problem D: Maze Runner
  - Problem E: Minecraft
- Hard
  - Problem F: Subsequence 3

Problem A: Card

Problem Author: Lee Zong Yu
Developement Lee Zong Yu
Editorial: Lee Zong Yu

Problem

Abridged Problem Statement

Given you an array, find a subsequence with maximum sum.

- Subtask 1: All elements are positive
- Subtask 2: Elements can be positive or negative

Solution

Subtask 1

Since all elements is positive, it is always optimal to select the entire array. The answer is just the sum of integers on the array.

Subtask 2

Negative integer will just decrease the total sum, hence it is always optimal to not select any negative integer. The answer is just the sum of positive integers on the array.

Problem B: Glass Bridge

Problem Author: Lee Zong Yu
Developement Lee Zong Yu
Editorial: Lee Zong Yu

Problem

Abridged Problem Statement

Squid game (glass bridge) but each level has k glass. Each player will jump to the correct glass if it is known, to the his/her favourite number otherwise.

- Subtask 1: $n, m \leq 100,\ k \leq 2$
- Subtask 2: $n, m, k \leq 10^6$

Subtask 1

Solution

- Maintain a variable that stores the latest level such that the real glass is known
- Enumerate for each player until the latest level such that real glass is known (the variable above), and continue to check if the real glass of next round is same as his/her favourite number
- Continue until the end of the levels is reach.
- Time Complexity: $O(nm)$
- Remark: Actually any make sense brute force can solve this.

Subtask 2

Solution

- Actually, we may just use the above solution except that we start the looping from the latest level such that the real glass is known
- You can imagine that you have two pointer - one pointing to the levels, one pointing to the players.
- If any of the pointer reach the end of the array, the loop break
- The answer is n - the index the pointer is pointing to (1 -index)
- Time Complexity: $O(n+m)$ because the first pointers move at most m times and the second pointer move at most n times.

Problem C: Sodoku

Problem Author: Zhou Xuhang
Developement Lee Zong Yu
Editorial: Lee Zong Yu

Abridged Problem Statement

Given n, find the smallest integer $\geq n$ such that the integer does not have repeating digits.

- Subtask 1: $n \leq 10^6$
- Subtask 2: $n \leq 9'876'543'210$

Subtask 1

- Since $n \leq 10^6$, n is small enough so that you just need to enumerate starting from n until you find an integer does not have repeating digits.
- Time complexity: $O(n)$

Subtask 2

- If $n \leq 9'876'543'210$, you cannot just enumerate.
- However, just a math question, how many integer fulfill the no repeating digits.
- Only integers of length at most 10 fulfil. -> By pigeonhole principle, there must be repeating digit
- $\sum_{i=1}^{10}$ number of integers with length $i \leq 10\cdot$ Number of integers with length $10 = 10 \cdot 10! = 36'288'000$
- Hence, you may generate all such integers. Then go through all of them to find the smallest such integer greater than n.

Problem D: Maze Runner

Problem Author: Lee Zong Yu
Developement Lee Zong Yu
Editorial: Lee Zong Yu

Maze Runner

Abridged Problem Statement

Given a **connected** graph, find the minimum-bottleneck path between 1 and n. That is, find the path from 1 to n with maximum edge weight on a path minimized

- Subtask 1: Graph is a Tree
- Subtask 2: The weights are only powers of 2.
- Subtask 3: No additional Subtask

Subtask 1

Graph is a Tree

Theorem

If a connected graph is a tree, then there is a unique path between any pairs of vertex.

- Hence, do a DFS/BFS Traversal from 1 to n and return the maximum edge weights on the unique path.
- Time Complexity: $O(n)$

Subtask 2

The weights are only powers of 2.

Enumerate the answers

- There are at most $\log(10^9)$ unique weight values (32). $2^{32} > 10^9$ (maximum weights)
- Hence, you may do an enumeration on all the weight values W ($W \in \{2^0, 2^1, \ldots, 2^{31}\}$
  - For each W, do a DFS/BFS traversal and ignore all edges with weight $> W$.
  - If n is reachable from 1, then W is a valid answer (a valid upper bound of the answer).
- Choose the lowest valid answer (i.e. lowest W such that n is reachable from 1).

Time Complexity: $O((n+m)\log(10^9))$

Subtask 3

Since there are at most m distinct weight values, subtask 2 solution is unable to pass.

Binary Search the answers

- Notice that, the question "Is n is reachable from 1 considering all edges with weights $\leq W$" is **monotonic** in terms of W.
- You can do a binary search on the answer (i.e. binary search on the lowest W from $[1, 10^9]$ such that the answer to the above question is YES)
- Similar to Subtask 2, for each W you try do a BFS/DFS to check if the question above is correct.
- Binary Searching the range of possible answer is $O(\log 10^9)$ hence it reduced to the same time complexity as above.

Alternative Solution to Subtask 3

This solution extends the solution of Subtask 1. When the given graph is a tree, it is easy.

Problem Reduction

Given a graph G, it can be proven that path from 1 to n on any Minimum Spanning Tree (MST) of the graph G is a minimum bottleneck path from 1 to n of the graph G.

Proof. Suppose, for a contradiction, there is an MST such that the **unique** path on the MST is not a minimum bottleneck path on G.

- Follows from assumption, the maximum edge on a unique path $>$ maximum edge on the minimum bottleneck path.
- You may replace the maximum edge on the unique path of MST to the maximum edge on the minimum bottleneck path.
- Now you form a spanning tree with total weight lesser than MST (Contradiction)

Alternative Solution to Subtask 3 (Cont.)

- Solution here is clear. Run any MST algorithm (i.e. Kruskal or Prim) to reduce the graph into a tree.
- Then you use the algorithm in Subtask 1 (BFS/DFS) to get the maximum weight on the unique path of tree.
- Time Complexity: $O(m \log n)$ from Kruskal or prim.

Problem E: Minecraft

Problem Author: Zhou Xuhang
Developement Lee Zong Yu
Editorial: Lee Zong Yu

Abridged Problem Statement

Abridged Problem Statement

Given an array of length n, add the minimum 1 to any index such that the array is convex (i.e. increasing order then decreasing order)

m is the number of strings
q is the number of query strings
maxlen is the maximum length of each string
sumlen is the sum of the length of each string

- Subtask 1: $1 \leq n \leq 1000$
- Subtask 2: $1 \leq n \leq 1000000$ There is only two height value
- Subtask 3: $1 \leq n \leq 1000000$

Greedy Structure

The following greedy subproblem structure is used on each solution.

Greedy subproblem Structure

- Suppose we are building an array of increasing order only.
- Suppose the the array prefix at $i-1$ is optimal with ans increment and $a[i-1]$ has final height of h_{i-1} (It may be same as the original value or greater).
- The optimal answer for array prefix at i is to increase $a[i]$ to h_{i-1} or not increase it if it is greater or equal to h_{i-1}.
- The optimal answer when array prefix at 0 is simply just not increase it.

Likewise for decreasing order because elements is in decreasing order is equilvalent to in increasing order but read from the back of the elements.

Subtask 1

With the greedy subproblem structure introduction, we can have a brute force solution by

- Enumerate the highest point / turning point of the convex. Say k
- It can be seen that k must be increament to the maximum value in the array if not the maximum value.
- Then apply the greedy subproblem structure from 0 to $k-1$ and also from $n-1$ back to $k+1$.

Subtask 2

It Does not helps to get subtask 3!!!
With maximum height only 2, the answer is a sequence of 1 and then a sequence of 2 then it is followed by another sequence of 1.
Since, element cannot be decreased, the elements between the first 2 and the last 2 must be the same (i.e. 2).
This subtask is meant to help you in observing the greedy subproblem structure.

Subtask 3

The concept of this subtask 3 is precomputation.

- After you enumerate the highest/turning point k, you need an additional loop to calculate the optimal answer prefix at $k-1$ and suffix at $k+1$.
- That is, prefix[i] stores the optimal answer for array prefix at i, suffix[i] stores the optimal answer for array suffix at i
- Suppose max is the largest element, the answer for highest point at k is prefix[k - 1] + $(max - a[k])$ + suffix[k + 1].
- Time Complexity: $O(n)$

Note: This problem extends on the concept of prefix sum in *DP*

Problem F: Subsequence 3

Problem Author: Zhou Xuhang & Lee Zong Yu
Developement Lee Zong Yu
Editorial: Lee Zong Yu

Problem

Abridged Problem Statement

Given a sequence and an integer x, find the longest subsequence such that the multiplication of neighbouring elements does not exceed x

- Subtask 1: $x = 10^{18}$
- Subtask 2: $n \leq 20$
- Subtask 3: $n \leq 1000$
- Subtask 4: $n \leq 10^5$

Subtask 1 & 2

Subtask 1

The answer is n. This subtask is awarded to students who understand the problem. Since $X = 10^{18}$ and each elements is at most 10^9. So, multiplication of any two elements is at most $10^{18} < x$

Subtask 2

- There are a total of 2^n subsequences. Since $n \leq 20$, there are only a total of $2^{20} \approx 10^6$ subsequences.
- Hence, just enumerate all of them and do an iteration to verify if it satisfies the limit condition. Among all subsequences that satisfy the limit condition, choose the longest one.
- The time complexity is $O(2^n \cdot n)$

Subtask 3

The solution is similar to the DP for Longest Increasing Subsequence.

Dynamic Programming

- Let $dp[i]$ denote the length of the longest valid subsequence ending at position i (i.e., within the array $[a_1, a_2, \ldots, a_i]$), where a_i must be included in the subsequence.
- To compute $dp[i]$, we apply the following transition:

$$dp[i] = \max_{j \text{ such that } j<i \text{ and } a_j \cdot a_i \leq X} (1 + dp[j]).$$

- Time complexity is $O(n^2)$

Subtask 4

There are two possible solution. Lets discuss the first one which extends from Subtask 3.

Optimise the DP (Alternative DP Formulation)

Lets try to construct a DP depends on a_i

- Let $dp[i][j]$ denote the length of the longest valid subsequence within the prefix $[a_1, a_2, \ldots, a_i]$, where the last element of the subsequence is j.
- The transitions are as follows:
  - If $j \neq a_i$, then $dp[i][j] = dp[i-1][j]$
  - If $j = a_i$, then $dp[i][a_i] = \max_{k \text{ such that } k<X/a_i} (1 + dp[i-1][k])$

Subtask 4 (Cont.)

Further Optimise the DP Formulation

- The first dimension (i) only depends on the $i - 1$, so we can compress the dimension by iterating over i from 1 to n.
- Since the second dimension (j) is only affected if $j = a_i$. Hence, at each iteration of i, only $j = a_i$ needs to be updated.

Subtask 4 (Cont.)

Alternative Understanding

At iteration i, the following operation is executed

- Range Maximum query (RMQ) - Query the maximum value from index 1 to X/a_i.
- Point update - Update the value at index a_i to the result of the previous query plus 1.

Hence, it is a classical Range Query Point update problem which can be solved by segment tree.

Subtask 4 (Cont.)

Optimise the DP (Coordinate Compression)

If $\max_{1 \le i \le n} a_i = 10^9$, we cannot apply the $DP[j]$ up to 10^9. However, since the number of elements is at most 10^5, there are at most 10^5 distinct values. Thus, we can perform a coordinate compression by the following:

- Sort and duplicate the values of a_i and $\lfloor X/a_i \rfloor$.
- Assign each unique value a compressed index via a map

We also include $\lfloor X/a_i \rfloor$ in the compressed array because in this way, both updates and queries are performed on compressed indices.

Subtask 4 (Cont.)

Optimise the DP (Becareful when X is negative)

Notice that X and a_i can be negative. After a thorough case discussion, we can conclude the following:

- If a_i is positive, the RMQ query the maximum value from $-\infty$ to $\lfloor X/a_i \rfloor$ before coordinate compression.
- If a_i is negative, the RMQ query the maximum value from $\lceil X/a_i \rceil$ to ∞

Thus, we apply two different ranges of RMQ for different cases.

The Time complexity is: $O(n \log \max_{1 \le i \le n} a_i)$

Small conclusion for the route of optimizing the DP

Overall Flow

- From creating DP depends on index become depends on value
- Formulate the new DP into a Range Query Point Update (RQPU) problem
- Apply coordinate Compression so that the number of distincts value is $O(n)$
- Do a case discussion for $X < 0$.

Alternative Solution for Subtask 4

Alternative Solution is to have an observation. We provide another form of subtask to help you in deriving the observation.

- Subtask a: $a_i, X > 0$
- Subtask b: $X > 0$
- Subtask c: a_i, X can be negative

It corresponds to three different cases

Subtask a: $a_i, X > 0$

If all elements and X are positive, we can use a greedy approach to solve the problem optimally. The strategy is as follows:

Greedy

Enumerate the array, and maintain the length of the subsequence and the last element of the subsequence at each step. At step i,

1. If the product of the last element in the subsequence and $a[i]$ does not exceed X, then it is optimal to include $a[i]$ in the subsequence. The length of the subsequence is increased by 1.
2. Otherwise, if $a[i]$ is smaller than the last element in the subsequence, then it is optimal to replace the last element in the subsequence with $a[i]$.

Intuition on correctness

Intuition of the correctness: For both cases, if a_i is not included in the subsequence, the resulting subsequence will be no longer than the subsequence that includes a_i. Thus, this approach ensures the longest valid subsequence is achieved by greedily including or replacing elements based on different cases.

The full rigorous proof of the correctness can be found here: (`https://drive.google.com/file/d/1quENZdQvE2JIV33YL28hmmDnmboKto98/view?usp=drive_link`).

Subtask b: $X > 0$, a_i can be negative

Modified Greedy Algorithm

In this case, we can follow the same strategy as subtask a, with one adjustment. When comparing $a[i]$ to the last element in the subsequence, we compare the absolute values instead of the values themselves.

The proof of correctness can be found here(`https://drive.google.com/file/d/1quENZdQvE2JIV33YL28hmmDnmboKto98/view?usp=drive_link`).

Subtask c: X can be negative

Observation

When X is negative, we notice that the neighbouring elements of any valid subsequence must have different parity.

Definition

In this case, we maintain two different states during the enumeration:

- One for when the last element in the subsequence is positive.
- One for when the last element is negative.

Let pos_{len} (resp. neg_{len}) be the length of the subsequence and pos_{last} (resp. neg_{last}) be the value of the last element in the subsequence if the last element must be positive (resp. negative).

Algorithm when X is negative

Algorithm

At each step i, if a_i is positive, we check the following:

1. If $a_i \cdot neg_{last} \leq X$ and $neg_{len} + 1 > pos_{len}$, we update $pos_{len} = neg_{len} + 1$ and set $pos_{last} = a_i$.
2. Otherwise, if absolute value of a_i is greater than pos_{last}, we update pos_{last} to a_i.

Intuition/Proof of correctness

The intuition behind the strategy is as such:

Case Discussion

1. Case 1 ensures that if, after including element a_i, the subsequence with the negative last element is longer, we prefer that subsequence by updating it.
2. Case 2 ensures that we always replace the last element with the greatest possible absolute value. It is because given any two neighbouring elements, let's say a_i is the positive element and a_j is the negative element, the constraint on them is $a_i \cdot a_j \leq X$. It follows that $a_i \cdot (-a_j) \geq (-X)$. This constraint contrasts with the one when X is positive and hence can be proven using a similar method.

Time complexity: $O(n)$

Final Note of this problem

The author also proposes the problem to Malaysian Computing Olympiad 2025 competition and was accepted.
You may read the editorial written here.
`https://codeforces.com/blog/entry/142321`