Nanyang Programming Contest 2025

Stage 2: The Second Contest

Lee Zong Yu, Pu Fanyi, Zhou Xuhang

Nanyang Technological University

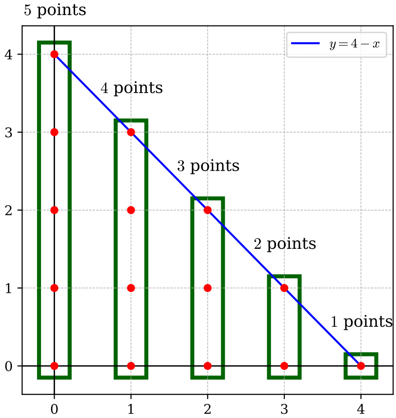
05 April 2025

- Easy
  - Problem A: Triangle - Math & Two Pointer
  - Problem B: Tree 2 - Tree Structures
- Medium
  - Problem C: Subsequence 2 - DP or Two Pointer
  - Problem D: Game - DP
- Hard
  - Problem E: Eating - Greedy, Implementation

Problem A: Triangle

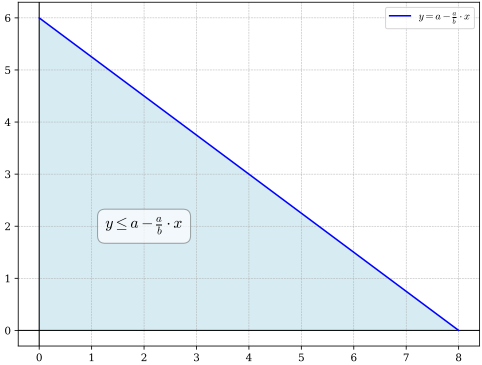
Problem Author:	Lee Zong Yu & Pu Fanyi
Developement	Pu Fanyi
Editorial:	Pu Fanyi

Triangle [$a = b$]



$$s = (a+1) + a + \cdots + 1 = \frac{(a+2)(a+1)}{2}$$

Triangle [$\mathcal{O}(\max\{a, b\})$]



Count (x, y):

$$y \leq a - \frac{a}{b}x$$

Triangle $[\mathcal{O}(\max\{a, b\})]$

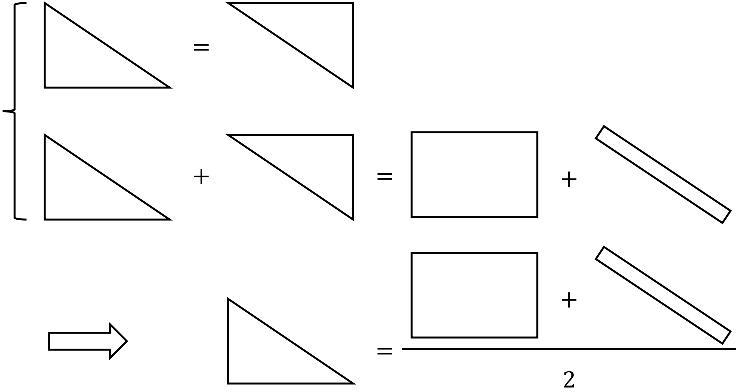
For all $x \in [0, b]$, we need:

$$0 \le y \le a - \frac{a}{b}x$$

So for one x, we have $1 + \left\lfloor a - \frac{a}{b}x \right\rfloor$ valid y.
So the answer is:

$$\sum_{x=0}^{b} 1 + \left\lfloor a - \frac{a}{b}x \right\rfloor$$

Triangle [$\mathcal{O}(\log \max\{a, b\})$]



Triangle [$\mathcal{O}(\log \max\{a, b\})$]

For simplicity:

$$\triangle = \frac{\square + \diagdown}{2}$$

It is easy to know that

$$\square = (a+1)(b+1)$$

So the question is to calculate $\diagdown$.

Triangle [$\mathcal{O}(\log \max\{a, b\})$]

An observation is

$$\diagdown = \diagup$$

So we need to count (x, y):

$$y = \frac{a}{b}x \Longrightarrow ax = by$$

Let $c = ax = by$, we have $a, b \mid c$, which is

$$c = k \cdot \text{lcm}(a, b), k \in \mathbb{Z}$$

Triangle [$\mathcal{O}(\log\max\{a, b\})$]

And we have constrain $0 \le c \le ab$ as $0 \le x \le b, 0 \le y \le a$.

$$0 \le k \cdot \operatorname{lcm}(a, b) \le ab \Longrightarrow 0 \le k \le \frac{ab}{\operatorname{lcm}(a, b)} = \gcd(a, b)$$

So

$$\diagdown = \gcd(a, b) + 1$$

Which means the answer is

$$\triangle = \frac{\square + \diagdown}{2} = \frac{(a+1)(b+1) + \gcd(a, b) + 1}{2}$$

We can use *Euclidean algorithm* to calculate $\gcd(a, b)$ in $\mathcal{O}(\log\max\{a, b\})$ time. (Or `std::gcd` in C++, `math.gcd` in Python)

Triangle [Optional, Bézout's Identity]

At first, I didn't find $\diagdown = \diagup$, so I tried to solve $\diagdown$ directly.
We can have:

$$y = a - \frac{a}{b}x \implies ax + by = ab$$

This is a classic Linear Diophantine equation, which can be easily solved using Extended Euclidean Algorithm and Bézout's identity. This approach also has a time complexity of $\mathcal{O}(\log \max\{a, b\})$.

Triangle [Optional, Pick's theorem]

Theorem (Pick's theorem)

Suppose that a polygon has integer coordinates for all of its vertices:

$$\mathcal{A} = \mathcal{I} + \frac{\mathcal{B}}{2} - 1$$

- $\mathcal{A}$ *is the area of the polygon*
- $\mathcal{I}$ *is the number of interior points*
- $\mathcal{B}$ *is the number of boundary points*

Triangle [Optional, Pick's theorem]

So the answer is

$$\mathcal{I} + \mathcal{B} = \left(\mathcal{A} - \frac{\mathcal{B}}{2} + 1\right) + \mathcal{B} = \mathcal{A} + \frac{\mathcal{B}}{2} + 1 = \frac{ab}{2} + 1 + \frac{\mathcal{B}}{2}$$

So the question is the find $\mathcal{B}$, which is

$$a + b - 1 + \diagdown = a + b + \gcd(a, b)$$

So

$$\mathcal{I} + \mathcal{B} = \frac{ab + a + b + \gcd(a, b)}{2} + 1$$

Problem B: Tree 2

Problem Author:	Pu Fanyi
Developement	Pu Fanyi
Editorial:	Pu Fanyi

Tree [30 pts?]

- Question: find the distance to the root for all nodes
- Idea: for all nodes, "climbing" to the root

procedure CLIMBING(o: Node)
    $r \leftarrow$ root of the tree
    $d \leftarrow 0$ ▷ depth of o
    while $o \neq r$ **do**
        $o \leftarrow p_o$ ▷ "Climb" to the top
        $d \leftarrow d + 1$
    end while
    return d
end procedure

Tree [60 pts!]

- This algo actually earns 60 pts!
- Because the tree generated randomly using $p_i = \text{rand}(1, i-1)$ has an expected depth of $O(\log n)$.
- The proof will be at the end of the solution.

Tree [100 pts]

- DFS and BFS is a good idea for most of the tree problem
- Because this method allows the entire tree to be traversed in a specific order.
- In both types of search, the parent node always arrives before the child nodes.
- So, if we let d_i be the distance of node i, during our DFS/BFS, d_{p_i} is always computed before d_i.

Tree [100 pts]

procedure DFS(o: Node)
    for s **in** SonOf(o) **do**
        $d_s \leftarrow d_o + 1$ ▷ Calculate the depth of o's son
        DFS(s) ▷ DFS s recursively
    end for
end procedure

Tree [100 pts]

```
procedure BFS(r: Root)
    q ← Empty Queue
    ENQUEUE(q, r)
    while q is not empty do
        o ← DEQUEUE(q)            ▷ Choose the front of the q as o
        for s in SONOF(o) do
            d_s ← d_o + 1         ▷ Calculate the depth of o's son
            ENQUEUE(q, s)         ▷ Push s to the back of the queue
        end for
    end while
end procedure
```

Tree [Proof of 60 pts]

Let

$$d_i = \mathbb{E}[\text{depth of node } i]$$

As $p_i \sim \text{uniform}(1, i-1)$, we can have

$$\begin{aligned}
d_i &= \mathbb{E}_{p\sim\text{uniform}(1,i-1)}\left[d_p + 1\right] \\
&= \mathbb{E}_{p\sim\text{uniform}(1,i-1)}\left[d_p\right] + 1 \\
&= 1 + \frac{1}{i-1}\sum_{p=1}^{i-1} d_p \\
&= 1 + \frac{1}{i-1} d_{i-1} + \frac{1}{i-1}\sum_{p=1}^{i-2} d_p
\end{aligned}$$

Tree [Proof of 60 pts]

As

$$d_i = 1 + \frac{1}{i-1}\sum_{p=1}^{i-1} d_p$$

so

$$\sum_{p=1}^{i-1} d_p = (i-1)(d_p - 1)$$

Thus

$$\begin{aligned} d_i &= 1 + \frac{1}{i-1} d_{i-1} + \frac{1}{i-1}\sum_{p=1}^{i-2} d_p \\ &= 1 + \frac{1}{i-1} d_{i-1} + \frac{1}{i-1} \cdot (i-2)(d_{i-1} - 1) \\ &= \frac{1}{i-1} + d_{i-1} = \sum_{i=1}^{i-1} \frac{1}{i} \end{aligned}$$

Tree [Proof of 60 pts]

So

$$\mathbb{E}[\text{depth of node } i] = \sum_{j=1}^{i-1} \frac{1}{i}$$

And this is a very famous formula called Harmonic series
It can be easily prooved that

$$\mathbb{E}[\text{depth of node } i] = \sum_{j=1}^{i-1} \frac{1}{i} \leq 1 + \int_{1}^{i-1} \frac{1}{x} \mathrm{d}x = 1 + \log(i-1)$$

Problem C: Subsequence 2

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

Abridged Problem Statement

Abridged Problem Statement

Given two non-descending arrays, find the Longest Common Subsequence (LCS) between the two arrays

- Subtask 1: $1 \leq N, M \leq 20$
- Subtask 2: $1 \leq N, M \leq 500$
- Subtask 3: $1 \leq N, M \leq 10000$
- Subtask 4: $1 \leq N, M \leq 100000$

Subtask 1

Complete Search

- Do a complete search (brute force) over all the possible subsequences of array A.
- During the search, an array C is constructed. To verify that array C is a subsequence of array B, you may enumerate the array B and maintain a "pointer" on array C.
- During the enumeration of array B, if the current element is same as the element the pointer is pointing to at array C, increment the pointer.

Implementation and Analysis of Algorithm

- To implement the algorithm, you may write a **recursive function** (backtracking). Alternatively, you may use a bitmask to write an **iterative function**. Please check the code for the implementation.
- The time complexity is as such:
  - There are 2^N possible subsequences (Each element has two possibilities - chosen in the subsequence or removed from the array).
  - To verify whether each potential subsequence is a subsequence of B, you have to do an $O(N + M)$ enumeration. That is $O(N)$ for the enumeration on the subsequence C and $O(M)$ for the enumeration on the array B.
  - The total time complexity is $O(2^N \cdot (N + M))$

Subtask 2

- $1 \leq N, M \leq 500$
- With this constraint, $O(N \times M)$ solution are able to pass. Therefore, it is just a classic DP solution.

Dynamic Programming (DP)

- DP[i][j] stores the length of LCS of the array $[a_0, a_1, \ldots, a_{i-1}]$ and the array $[a_0, a_1, \ldots, a_{j-1}]$.
- The transition between the states is as follows:
  - DP[i][j] $=$ DP[i - 1][j - 1], if $a_i == a_j$ and $i \geq 1, j \geq 1$.
  - DP[i][j] $=$ max(DP[i - 1][j - 1], DP[i][j - 1]), otherwise (if $i - 1$ or $j - 1$ would be negative, just ignore it).
- The base state DP[0][0] $= 0$.
- The answer is DP$[N][M]$ You may build a bottom-up DP or top-down DP.

Subtask 3

Motivation

- $1 \leq N, M \leq 10000$
- With this constraint, $O(N \times M)$ solution are still able to pass. However, if the two-dimensional DP array is constructed with size $N \times M$, your solution would get Memory Limit Exceeded (MLE). This is because the space complexity is $O(N \times M)$
- From SC1006, if the DP arrays store 32-bit integers, what is the maximum number of bytes used by the DP array?
- 32-bit is equivalent to 4 bytes. If $N = M = 10000$, the number of bytes is $N \times M \times 4 = 4 \times 10^8 B \approx 381 MB$.
- The memory limit is $512 MB$. Therefore, including some required memory storage (TODO how much), the DP array would use too much memory space.
- The idea is to use "Rolling DP" to compress the space to $O(N + M)$.

Rolling DP

"Rolling DP"

- Notice that, for any i, all the DP with first dimension is i only depends on the DP with first dimension is $i-1$. That is, given i, $\forall j \in [0, M]$, $\text{DP}[i][j]$ only depends on $\text{DP}[i-1][j]$ or $\text{DP}[i-1][j-1]$.
- If bottom-up DP is used (Nested Loop of i followed by j), it is not required to store any states with first dimension $\leq i-2$.
- Therefore, the transition is as follows:
    - $\text{DP}[i\%2][j] = \text{DP}[(i-1)\%2][j-1]$, if $a_i == a_j$ and $i \geq 1, j \geq 1$.
    - $\text{DP}[i\%2][j] = \max(\text{DP}[(i-1)\%2][j], \text{DP}[i\%2][j-1])$, otherwise (if $i-1$ or $j-1$ would be negative, just ignore it).
- The answer is $\text{DP}[N\%2][M]$

Subtask 4

- $1 \leq N, M \leq 100000$. $O(NM)$ solution would TLE
- The idea is to take advantage that arrays A and B are in ascending order.
- For a given i and j, if $a_i < b_j$ and denote $i' > i$ as the first index such that $a_{i'} \geq b_j$, we know that for all $j'' \geq j$ and $i \leq i'' \leq i'$, $\text{DP}[i''][j''] = \text{DP}[i][j]$.
- With this, we do not need to compute the answer to many DP states. We just need to increment i to i' and compute the solution. Likewise, if $a_i > b_j$, we increment j until $a_i \leq b_j$.
- If $a_i = b_j$, then we increment both i and j by one and add 1 to the answer.
- This leads to a two pointer algorithm.

Problem D: Game

Problem Author:	Zhou Xuhang
Developement	Zhou Xuhang
Editorial:	Zhou Xuhang

Abridged Problem Statement?

Abridged Problem Statement

Given an array that consists of integers. Use the set of cards to move to get the maximal sum.

- For all task, $1 \le N \le 10^5, 1 \le R \le 10^9, 0 \le |a| \le 10^9$
- Subtask1 & 2: $K = 1$
- Subtask3: $K = 2$
- Subtask4: $K = 4$
- Subtask5: $K = 8$

Observation

Observation 1

The total distance moved is fixed until the set of movement cards is flushed.

Observation 2

Always use the energy card as soon as possible.

Subtask 1 & 2

- $K = 1$. (There is only one movement card)
- The movement sequence is fixed. Just use the following strategy:
  - Use the energy card.
  - Use the movement card, check whether the current energy is negative.
  - If the destination is not reached, the cards are replenished. Repeat this strategy.

Motivation

These subtasks are designed for contestants to verify their understanding of the problem statement, especially about **"whether an energy card could be used when we have already arrived at the destination"**.

Subtask 3

- $K = 2$ (There are only two movement cards)
- If we use the energy card first, then the only two possible strategies are as follows:
  - Use the first movement card, then the second movement card.
  - Use the second movement card, then the first movement card.
- It is easy to compare which is better (by implementing both strategies and taking the better one). Just be careful about **if there is a sudden death situation**.

Subtask 4

- $K = 4$
- Based on the two observations and the methodology we used in subtask 3, we can have the following basic idea.

Key Conclusion

If we already stand on the pillar at position x, we do not care what happened before that. We only want to have higher energy at the current time.

Subtask 4

- Based on the conclusion, let's start with position 0.
- Firstly, we use the energy card and get $Energy = R$.
- Then, we perform a complete search over all possible orders of the movement cards used.
- Our goal is to find the best order which can provide us the highest $Energy$ without sudden death.
- The complexity in this step is $K \cdot K!$ (There are $K!$ possible permutations of movement cards, and we perform an iteration over the movement cards for each possible permutation).

Subtask 4

- Whatever the best order is, we will always stand at $\sum_i^k b_i$ th pillar.
- And we hold the highest possible *Energy* now (through complete search).
- Now our deck of cards is flushed. In the next round, our goal is similar to the first round.
- Our goal is to find the best order which can provide us the highest $\Delta Energy$ without sudden death.
- The complexity in this step is $K \cdot K!$.

Subtask 4

- This kind of consideration is called **Optimal substructure**.
- Just repeat this complete search until she arrives at the destination.

Time Complexity

In each round, we need a complete search, and we have $N/\sum b_i$ rounds.
The Time Complexity is $O\left(\frac{N}{\sum b_i} K \cdot K!\right)$, which cannot pass subtask 5.
But if you can solve the subtask, it is already a huge success!

Subtask 5

- We would like to introduce a type of dynamic programming problem here called **Bitmask DP**.
- The dynamic programming state is DP[i][bitmask] representing the best *Energy* we could have when
  - we stand at i-th pillar
  - The remaining cards we hold in our hand are encoded by the bitmask (which is explained below).
- bitmask is an integer (we understood it in binary representation), where the jth bit from Least Significant Bit is on $(= 1)$ implies that we still have the jth card on hand.

Explaination

For example DP[23][11]. We have $11_{10} = 1011_2$.
The value of DP[23][11] is the maximal possible energy he can get when he stands in the 23rd Pillar and holds the first, second and the fourth cards.

Subtask 5

State Transition

$DP[0][11111111_2] = R(\text{used the energy card})$
$\rightarrow DP[b[1]][11111110_2] = max(itself, DP[0][11111111_2] + a[b[1]]$
$\rightarrow DP[b[2]][11111101_2] = max(itself, DP[0][11111111_2] + a[b[2]]$
...
More generally, for DP[i][j] we can try to use each remaining card - condition:$(j\&2^{k-1} \neq 0)$
$DP[i + b[k]][j \oplus 2^{k-1}] = max(itself, DP[i][j] + a[i + b[k]])$

- Special case 1:If $j \oplus 2^{k-1} = 0$, we flush the set of cards and use the energy card immediately.
- Special case 2:If $i + b[k] > N$, we reached and destination.

Subtask 5

Time & Space Complexity

The size of the state space is $N * 2^K$ and each state have $bitcount(K)$ transitions.

- Space Complexity: $O(N * 2^K)$
- Time Complexity: $O(NK * 2^K)$

Problem E: Eating

Problem Author:	Zhou Xuhang
Developement	Zhou Xuhang
Editorial:	Zhou Xuhang

Abridged Problem Statement

Given n cakes with $size_i$.

Abridged Statement (Optimization Problem)

You can merge two cakes M times. And you can eat the cake by unit x infinite times, where x should be less than half the cake size. But you cannot eat the cake whose *size* is smaller than K.
Your goal is eat the cake as much as possible.

Observation

Observation 1

We could merge first and eat the cakes after that.
As you can eat the cake infinite times, you can eat 1 unit until the cake's size becomes K.

Observation 2

When the cake's size become K, only one bite on this cake in the future. Thus we always eat $K/2$ unit in this case.

Observation 3

Maximize the cake you eat means minimize your waste.

Greedy

Intuitive thinking

Let's denote w[i] as the remaining unit in the end.

- $w[i] = (k+1)/2$ if $a[i] >= k$,
- $w[i] = a[i]$ if $a[i] < k$.

To minimize the sum of w[i], a intuitive solution here is sort the w[i] and remove the first M w[i] in decreasing order.

Greedy

Issue?

But there is another better choice:
We can merge two cakes both of size $k >= a[i] > (k+1)/2$. In this case, we not only eliminated the wastage of one cake, but also decrease another cake's waste value from $a[i]$ to $(k+1)/2$.

Greedy

Hence, our solution is:
First merge all cake with size $(k+1)/2 \leq a[i] < k$ in decreasing order.
After that, we eliminate the cake by merging to a large cake one by one.