# 2024 KAIST RUN Spring Contest
# Editorial

Sponsored By:

Jane Street

STARTLINK

SAMSUNG SOFTWARE MEMBERSHIP

## Problem A. RUN Number

Problem Idea: Eunjae Jin (`ai4youej`)
Preparation: Eunjae Jin (`ai4youej`)

### Subtask 1 (5 points)

If $N = 1$, then $K$ is a RUN number. If $N = 2$, we can solve in $\mathcal{O}(K^3)$ using brute-forcing.

### Subtask 2 (10 points)

It is possible to solve it with dynamic programming or backtracking.

### Subtask 3 (25 points)

Let $A$ be a non-decreasing number if its digits are in non-decreasing order. If we notate $A = \overline{s_1 s_2 \dots s_N}$, then by thinking $B = A - \overline{s_1 s_1 \dots s_1}$, $B$ is also a non-decreasing number. Since $B$ has fewer digits than $A$, we can get an answer easily by applying the above repeatedly.

### Subtask 4 (60 points)

In Subtask 3, we subtract the largest RUN number which is smaller than or equal to $A$. Find the largest RUN number to subtract repeatedly like a greedy algorithm. The algorithm below shows one way to find it faster.

1. Let $u_x$ be a $x$-digit number whose digits are all 1s. (e.g., $u_4 = 1111$)
2. For $k$-digit number $A$, we only have to check $u_{k-1}, 2 \times u_{k-1}, \dots, 8 \times u_{k-1}, 9 \times u_{k-1}, u_k, 2 \times u_k, \dots, 8 \times u_k, 9 \times u_k$.

The proof of the correctness of the algorithm and the existence of the solution is written below. The intended solution is $\mathcal{O}(N^2)$.

### Proof of Algorithm Correctness

Let $u_k$ be a $k$-digit number whose digits are all 1s. (e.g., $u_4 = 1111$)
Proposition $P_n :=$ "$\forall x \le u_{n+1}, x$ can be represented as the sum of $(n+1)$ or fewer RUN numbers".
Let's prove $P_n$ is always true (since $u_{n+1}$ is larger than any $n$-digit number).

1. $P_1$ is true. $1, 2, \dots, 9$ and $11$ are all RUN numbers. $10 = 9 + 1$, which can be represented as the sum of two RUN numbers.
2. Assume $P_{n-1}$ is true, then $P_n$ is true. Let's define $m$ as a given number.
   - If $m = u_{n+1}$, it is clearly true.
   - If $m < u_{n+1}$, then $m \le u_{n+1} - 1 = 10 \times u_n$. Choose the maximum number $a \times u_n$ among $0, u_n, 2u_n, \dots, 9u_n$, which is smaller than or equal to $m$. Then $m - a \times u_n \le u_n$. Since $P_{n-1}$ is true, we can represent $m - a \times u_n$ with the sum of $n$ or fewer RUN numbers. Therefore, $m = (m - a \times u_n) + a \times u_n$ can be represented as $(n+1)$ or fewer RUN numbers. ■

## Problem B. Sequence and Queries

Problem Idea: Seyun Roh (`knon0501`)
Preparation: Seyun Roh (`knon0501`)

### Subtask 1 (8 points)

The following equality holds for $1 \le i, j \le n$ and $1 \le k \le \min(n-i+1, n-j+1)$:

$$f(i,j,k) = \begin{cases} 1 & \text{if } f(i,j,k-1) = 1 \land s_{i+k-1} \le s_{j+k-1} \\ 0 & \text{otherwise} \end{cases}$$

and $f(i,j,0) = 1$.
Therefore, all $f(i,j,k)$ can be calculated in $\mathcal{O}(n^3)$.

### Subtask 2 (92 points)

Let's define $g(i,j)$ as follows:

$$g(i,j) = \sum_{k=1}^{\min(n-i+1,n-j+1)} f(i,j,k)$$

Then, the following equality holds for $1 \le i, j \le n$:

$$g(i,j) = \begin{cases} g(i+1,j+1)+1 & \text{if } s_i \le s_j \\ 0 & \text{otherwise} \end{cases}$$

and $g(i,n+1) = g(n+1,j) = g(n+1,n+1) = 0$.
Therefore, the sum of all $g(i,j)$ can be calculated in $\mathcal{O}(n^2)$ by using dynamic programming.

## Problem C. Construct a Graph

Problem Idea: Yumin Park (`flakepowders`)
Preparation: Yumin Park (`flakepowders`)

### Subtask 1 (9 points)

If there exist $i, j, k$ such that $D_{i,j} + D_{j,k} < D_{i,k}$, then the length of the path from $i$ to $k$ through $j$ is less than $D_{i,k}$, so a graph satisfying the condition does not exist.
Otherwise, a complete graph with the weight of the edge connecting $i$ and $j$ being $D_{i,j}$ satisfies the condition.

### Subtask 2 (19 points)

If an edge exists between two vertices $u, v$ in the constructed graph, the weight of that edge must be $D_{u,v}$.

- If the weight is less than $D_{u,v}$, the shortest path length between $u, v$ will become less than $D_{u,v}$.
- If the weight is greater than $D_{u,v}$, this edge will not affect any shortest paths. This is because a path with the shortest path between $u, v$ instead of this edge will always be shorter. Since the goal is to minimize the total weight of the edges, there is no reason to include this edge.

Therefore, we can think of $D_{u,v}$ as the cost of connecting vertices $u, v$. Here's how to construct the graph:
Iterate through the pairs of vertices $(u, v)$ in ascending order of $D_{u,v}$. For each pair $(u, v)$, check if the length of the current shortest path between $u$ and $v$ is greater than $D_{u,v}$. If it is, you must directly connect the two vertices with an edge of weight $D_{u,v}$. This is because any other remaining vertex pairs that are not yet connected have a connection cost of $D_{u,v}$ or more, so they cannot affect the shortest path between $u$ and $v$. After adding the edge, update the shortest paths that pass through this new edge in $\mathcal{O}(N^2)$ time.
After iterating through all elements of $D$, verify if the shortest paths in the constructed graph are correct. If they are, return the graph. Otherwise, return that no graph satisfies the conditions.

### Subtask 3 (72 points)

There's no need to manage the shortest paths in the graph we're constructing. This is because when iterating a vertex pair $(u, v)$, the shortest path length between any pair of vertices $(i, j)$ with $D_{i,j} < D_{u,v}$ will already be equal to $D_{i,j}$.

- This is obvious if $i$ and $j$ are directly connected.
- If $i$ and $j$ were not directly connected, it implies that the shortest path length between them was already $D_{i,j}$.

Also, if a path of length $D_{u,v}$ exists between $u$ and $v$ without directly connecting them, the weights of the edges forming the path will be strictly less than $D_{u,v}$. Therefore, if any vertex $m (m \neq u, v)$ is on such path, it must satisfy $D_{u,v} = D_{u,m} + D_{m,v}$. The existence of such vertex is equivalent to the existence of such path, and we can check it in $\mathcal{O}(N)$ time.

## Problem D. Closet

Problem Idea: Yerin Lee (`abra_stone`)
Preparation: Yerin Lee (`abra_stone`)

### Subtask 1 (4 points)

Let's fix $k$. The answer would be the maximum of $\max_{1 \le j < k}(d_j - d_{j+1})$ and $\max_{k \le j < N}(d_{j+1} - d_j)$. We can solve this by maintaining prefix/suffix max for every $1 \le k \le N$ in $\mathcal{O}(1)$.

### Subtask 2 (21 points)

From this subtask, we will perform a binary search on $x$. Now we only need to solve the decision problem of whether there exists a subsequence of length at least $N - M$ that forms an almost beautiful closet.
Define $dp[u][0]$ as the maximum length of subsequence $i_1 < i_2 < \cdots < l_p = u$ such that $d_{i_j} - d_{i_{j+1}} \le x$ for all $1 < j < p$. Simillarly, define $dp[u][1]$ as the maximum length of subsequence $i_p = u > i_{p-1} > \cdots > i_1$ such that $d_{i_{j+1}} - d_{i_j} \le x$ for all $1 < j < p$. The maximum length of a beautiful closet would be $\max_i(dp[i][0] + dp[i][1] - 1)$.
We could solve $dp[u][0]$ as $dp[u][0] = \max_{v<u, d_v - d_u \le x}(dp[v][0] + 1)$.
$dp[u][1]$ could be solved similarly. Each decision problem can be solved in $\mathcal{O}(N^2)$ time, resulting in $\mathcal{O}(N^2 \log X)$ time complexity solution. ($X = \max(c_i)$)

### Subtask 3 (16 points)

Here, we extend the solution for subtask 2.
When solving $dp[u][0]$ in ascending order of $u$, maintain $dp'[y] = \max_{v<u, d_v = y} dp[v][0]$. Now we can solve dp values by $dp[u][0] = \max_{y - d_u \le x} dp'[y] + 1$. $dp[u][1]$ could be solved similarly in descending order of $u$.
Each decision problem can be solved in $\mathcal{O}(NX)$, resulting in $\mathcal{O}(NX \log X)$ time complexity solution. ($X = \max(c_i)$)

### Subtask 4 (59 points)

Speed up the dp transition of subtask 3 using the segment tree. First, use coordinate compression to reduce the range of $d_i$ into $\mathcal{O}(N)$. Then, by increasing $u$, maintain a segment tree that maintains $dp'$.
Each decision problem can be solved in $\mathcal{O}(N \log N)$, resulting in $\mathcal{O}(N \log N \log X)$ time complexity solution. ($X = \max(c_i)$)

## Problem E. Two Histograms

Problem Idea: Geunyoung Jang (`azberjibiou`)
Preparation: Geunyoung Jang (`azberjibiou`)

### Subtask 1 (30 points)

Let's simplify the condition for the 3rd grid. Since $(A \cup B)^C = A^C \cap B^C$, squares that are colored white should be a union of two white histograms, one falling from the top and one progressing from right to left. For every white cell, every cell right to it should be white, or every cell above it should be white. So no white cell has a black cell right to it and a black cell above it. Let's call this structure "L structure".

If we color a subset of cells of the third grid without making an L structure, we can color the rest without making an L structure. This can be done by coloring a cell white if and only if there is no black cell above it or right to it.
Now let's color the ends of $K \times 1$ rectangles. Let the WB rectangle be a rectangle where the left end cell is colored white and the right end cell is colored black. Let the BW rectangle be a rectangle where the left end cell is colored black and the right end cell is colored white.
For every $K \times 1$ rectangle, we can know whether there is a black cell right to the white cell regardless of how we color the other rectangles.
There is no black cell right to its white end cell if and only if it is a WB rectangle and no rectangle is right to the rectangle. Let a rectangle be a "special rectangle" if there is no rectangle right to it. Let other rectangles be non-special rectangles.
For there to be no L structure, the white end cell of every non-special rectangle and special WB rectangle should not have a black cell above it. Let's call this condition as $\star$.

Consider rectangles that have the same $x$ value. Every rectangle above a WB rectangle should be a WB rectangle by $\star$. For $p$ rectangles having the same $x$ value, there are at most $p + 1$ ways of coloring the end cells.
Define $dp[i][j]$ as the maximum score by coloring end cells of rectangles whose left end cell has $x$ value less than or equal to $i$. There should be $j$ BW rectangles whose left end cell is at $x = i$.
$dp[i+1][k] = \max(dp[i][j] + score[i+1][k])$. $score[i+1][k]$ is the score obtained by coloring rectangles whose left end cell is at $x = i + 1$, $k$ BW rectangles below, and WB rectangles above.
By coloring $j$ BW rectangles for rectangles having $x = i$ left end cells and $k$ BW rectangles for rectangles having $x = i + 1$ left end cells, we should check whether it satisfies the $\star$ condition. This could be done in constant time. Therefore we have a $\mathcal{O}(N^2)$ dynamic programming solution.

### Subtask 2 (30 points)

By some casework, dp transition could be done in linear time, resulting in $\mathcal{O}(N)$ solution.

### Subtask 3 (40 points)

For the $\star$ condition, all we need to see is the end cells of the rectangle that are above or below the end cell of a rectangle. After checking whether the rectangles are special, we can group the rectangles by the remainder of the $x$ value divided by $K - 1$. Now we can solve $K - 1$ cases of $K = 2$ instances.

## Problem F. Discount Event

Problem Idea: Youngwoo Park (`flappybird`)
Preparation: Youngwoo Park (`flappybird`)

### Subtask 1 (10 points)

Let's simplify the problem. The given cities and roads can be thought of as a weighted tree. For each query, we need to find the diameter when the weight of the path ending at a given vertex in the tree becomes 0.
We can calculate the diameter of a tree in $\mathcal{O}(N)$ time complexity using dynamic programming. Let's assume the tree is rooted at vertex 1. Define $dp_1[v]$ as the longest distance from $v$ to any descendants of $v$, $dp_2[v]$ as a diameter of the subtree of vertex $v$.
The answer to the query is $dp_2[1]$.
These two values can be calculated by DFS in $\mathcal{O}(N)$ time complexity. There are $Q$ queries, we can solve the problem in $\mathcal{O}(NQ)$ time complexity.

### Subtask 2 (24 points)

For each vertex $v$, let $p$ be the parent vertex of $v$. Then, define $dp_3[v]$ as the length of the longest path starting from $p$ and going to a vertex that is a descendant of $p$ but is not a descendant of $v$.
Consider the vertices that are descendants of p but are not descendants of v. For all these vertices $x$, let $S_v$ be the set of the length of all $p - x$ paths. For each vertex $v$, calculate the greatest 3 elements of $S_v$. This can be done in $\mathcal{O}(N)$ time complexity with DFS.
We can calculate the answer to the query for vertices $v_0(= 1), v_1, \ldots, v_k$ by these two values.
If the diameter of the tree meets the given path, the answer is the sum of two different values of $S_{v_1} \cup S_{v_2} \cup \cdots \cup S_{v_k}$. Otherwise, the length of the diameter is $\max\{dp_3[v_1], dp_3[v_2], \ldots, dp_3[v_k]\}$.
For all vertices, these values can be calculated in the order of DFS in linear time complexity.

### Subtask 3 (27 points)

First, define $dp_4[v]$ as the diameter of the tree when $v$'s descendants (excluding $v$) are removed from the tree. This can be calculated in $\mathcal{O}(N)$ time complexity using DFS.
For each query, let's assume that vertex $X_i$ is the parent vertex of vertex $Y_i$.
We can classify the candidate paths of the diameter.

- Diameter includes $(X_i, Y_i)$ edge. The length of the diameter is $dp_3[X_i] + dp_1[Y_1]$.
- All vertices forming the diameter are descendants of v. The length of the diameter is $dp_2[X_i]$.
- None of the vertices forming the diameter are descendants of v. The length of the diameter is $dp_4[X_i]$

The maximum value among the three cases is the answer to the query.

## Subtask 4 (39 points)

For each query, we can classify the candidate paths of the diameter.

- Diameter and $uv$ path doesn't have an intersection, and the closest vertex from the diameter among $uv$ path is $u$ or $v$.
- Diameter and $uv$ path doesn't have an intersection, and the closest vertex from the diameter among $uv$ path is neither $u$ nor $v$.
- Diameter and $uv$ path has an intersection.

The first case can be calculated using $dp_2[X_i]$, $dp_2[Y_i]$.
The second case is similar to the solution of subtask 2. We need to find the maximum value of $dp_3$ among the paths. To perform this, we can use a sparse table to speed up.
The last case is also similar to subtask 2. We can find the longest two paths which start from the $X_iY_i$ path using a sparse table. The sum of the two paths is the length of the diameter.
Total time complexity is $\mathcal{O}((N+Q)\log N)$.

## Problem G. One, Two, Three

Problem Idea: Geunyoung Jang (`azberjibiou`)
Preparation: Geunyoung Jang (`azberjibiou`)

### Subtask 1 (20 points)

Define $c$ by replacing 3 with $-1$ from $a_i$. Let $S$ be a prefix sum of $c$. $c_i \equiv a_i \pmod 4$.

**Lemma 1.** *We can remove every element of the sequence if and only if $S_N \geq 0$ and $S_N$ is divisible by 4.*

*Proof.* ($\rightarrow$) For every removed segment, the sum of $c$ is divisible by 4 and is greater or equal to 0.
($\leftarrow$) Let's use induction on $N$.
If $N \leq 4$, the sum of $a$ is less than 12. Since $c_i \equiv a_i \pmod 4$ and the sum of $c_i$ is divisible by 4, the sum of $a$ is divisible by 4. We can erase every element at once.
Now assume $N > 4$. If there is adjacent 3 and 1 or adjacent 3 and two 2s, remove the segment. Since this does not change $S_N$, we can apply the induction hypothesis.
Now assume 3 and 1 are not adjacent, and no segment is equal to 233, 323, 332. Since $N > 4$ and 3 and 1 is not adjacent, $S_N > 0$. From $S_0, S_1, S_2, S_3, S_4$, there exists two value that has the same remainder by 4. Let it be $S_p, S_q$. $S_q - S_p$ is 4 or 8. If $S_q - S_p = 8$, then elements between $p$ and $q$ are 2222. By changing $q$ into $p + 2$, we get $S_q - S_p = 4$. By removing this segment, $S_N$ remains greater than or equal to 0. By using the induction hypothesis, we can remove all elements. □

Assume we have decided whether we would remove elements from 1 to $i$. Then we could

1. Decide to not erase the next element, which is $(i + 1)$th element.
2. Decide to erase a segment that starts from $i + 1$. In this case, the sum of $c$ of the segment should be greater than or equal to 0 and should be divisible by 4.

By using dynamic programming of minimum sum of $a$ and maximum sum of $b$, we have a $\mathcal{O}(N^2)$ solution.
Note that $N \leq 3000$, so dynamic programming solution without observation would not pass, such as time complexity with $\mathcal{O}(N^3/w)$ using a bitset.

### Subtask 2 (30 points)

For each removed segment, we need to find a way to erase every element. By the induction hypothesis, there always exists a segment where the sum of $a$ is 4 or 8 and maintains the sum of $c$ greater than or equal to 0. By recursively finding such segment in $\mathcal{O}(N)$, we have a $\mathcal{O}(N^2)$ solution.

### Subtask 3 (20 points)

Let's define dp as (minimum sum of $a$, $-$maximum sum of $b$) to use the min function. Dynamic programming transition works as the minimum of the following.

$$dp_i = dp_{i-1} + (a_i, -b_i)$$

$$dp_i = \min_{\substack{j<i,\ S_j \leq S_i \\ S_j \equiv S_i \pmod 4}} dp_j$$

By speeding up the transition using a segment tree, we have a $\mathcal{O}(N \log N)$ solution.

### Subtask 4 (30 points)

Let's find a segment where the sum of $a$ is 4 or 8 and the sum of $c$ is greater than or equal to 0. Since $S_N$ decreases as we remove a segment, a segment that could not be removed due to the condition of the sum of $c$ cannot be removed later. By using stack and sweeping from the first element to the last element, we could repeatedly find a way to remove a segment that lasts at the element that we are tracking. This results in a solution of time complexity of $\mathcal{O}(N)$.

## Problem H. Tree Kadane

Problem Idea: Jinhan Park (`jinhan814`)
Preparation: Jinhan Park (`jinhan814`)

### Subtask 1 (10 points)

Let's assume the tree is rooted at vertex 1. Let $dp[i]$ be the maximum value of $\sum_{i \in S} A_i$, where $S$ is a connected subset of the subtree rooted at vertex $i$ that includes vertex $i$. Given $A_1, \dots, A_N$, we can calculate $dp[1], \dots, dp[N]$ using tree dp. The answer to each query is $\max(dp[i])$, and the total time complexity is $\mathcal{O}(NQ)$.

### Subtask 2 (20 points)

If the degree of each vertex in the tree is less than 3, we can consider the tree as a one-dimensional array. In this case, we just need to solve the problem with two types of queries: updating the value of a vertex and calculating the maximum contiguous subarray sum. By using a segment tree to manage the sum of $A_i$, the maximum prefix sum of $A_i$, the maximum suffix sum of $A_i$, and the maximum sum of contiguous subarray of $A_i$, we can handle each query in $\mathcal{O}(\log N)$ time and total time complexity is $\mathcal{O}(N + Q \log N)$.

### Subtask 3 (70 points)

Using Heavy Light Decomposition, we can split the tree into multiple chains. First, select a chain that includes the root. Then, recursively connect the subtrees rooted at vertices connected by light edges to the vertices in the chain. After completing this process, the tree will have a recursive structure where subtrees are connected to the chain that includes the root.
Let's manage the chains using a segment tree same as subtask 2. For each vertex $i$, let $acc[i]$ be the sum of $\max(c_j, 0)$, where $j$ is a vertex connected with vertex $i$ by light edge, and $c_j$ is the maximum connected subset-sum of the subtree rooted at $j$ that includes vertex $j$. Similarly, let $mx[i]$ be $\max(d_j)$ where $d_j$ is the maximum connected subset-sum of the subtree rooted at $j$. By constructing a segment tree using $A_i, acc[i], mx[i]$ for each chain, we can find the answer using segment tree operations on the chain that includes the root.
Now, let's consider the update query that changes $A_i$. Starting from the chain containing vertex $i$, update the segment tree according to the changes in the chain. Then, propagate the updates up to the chain containing the parent vertex of the chain's top, updating $acc[i]$ and $mx[i]$ as needed. By managing $mx[i]$ with a data structure like std::multiset in C++ or a max heap, we can handle each update query in $\mathcal{O}(\log^2 N)$ time, as we are updating each chain, $acc[i]$ and $mx[i]$ $\mathcal{O}(\log N)$ times. The total time complexity is $\mathcal{O}(N + Q \log^2 N)$.