Problem A. Bingo

Problem Idea: Jaemin Choi (jh05013)
Preparation: Jaemin Choi (jh05013)

Subtask 1

Because $k \leq n-1$, there is just no way to form a bingo line! All you need to do is fill any k cells. The easiest way to do this is to fill the first k cells of the first row.

Subtask 2

This subtask can be solved by brute-forcing, since there are at most $2^{16} = 65536$ ways to fill the grid. It can be implemented by backtracking, or using a bitmask to represent the set of filled cells by a single number.

There is another way to solve this subtask, which can also be extended to a full solution. Note that if we can fill k cells, then we can also fill k' cells for any $k' < k$, by picking any appropriate subset of the filled cells. Therefore we just need to fill the maximum number of cells without forming any bingo line. Since n is quite small, we can find the solutions by hand. Here is an example:

```
          ###.
      ##.  #.##
   #. #.#  ##.#
.  .. .##  .###
```

Subtask 3

To fully solve the problem, we need a general way to fill the maximum number of cells. We can see that $n^2 - n$ is an upper bound for k, because we need at least one unfilled cell in each row. Now the question is: is this upper bound tight? That is, can we fill exactly $n^2 - n$ cells? It turns out the answer is yes, except when $n = 2$. Here we present one of the ways to do that.

If we ignore the diagonal bingo lines for now, we can consider arranging the unfilled cells diagonally:

```
      #####.
####.  ####.#
###.#  ###.##
##.##  ##.###
#.###  #.####
.####  .#####
```

But if we consider the diagonal bingo lines again, this approach sometimes doesn't work - in particular, when n is even, the diagonal going from the top-left to bottom-right will always form a bingo line. To fix this problem, we can slightly modify this approach by swapping the unfilled cells on the first and last row:

```
       .#####
.####  ####.#
###.#  ###.##
##.##  ##.###
#.###  #.####
####.  #####.
```

Now this solution almost works, but still fails when $n = 2$. Indeed, we saw earlier that the maximum number of filled cells for $n = 2$ is 1; there is no way to fill $n^2 - n$ cells. Fortunately $n = 2$ is the only exception for this approach, so we can just treat it as a separate case.

Shortest solution: 368 bytes

Problem B. Histogram

Problem Idea: Suchan Park (tncks0121)
Preparation: Jongseo Lee (leejseo)

Subtask 1 (10 points)

There are $h_1h_2\cdots h_N$ ways to choose the integers. Since N and h_i are very small, we can iterate through all cases and check whether the remaining area is connected or not.

To check if the area is connected, we can apply a graph traversal algorithm. However, there is an easier way: after subtracting a sub-histogram, the remaining cells of the i-th column correspond to $[x_i+1, h_i]$. Therefore, the remaining area is connected if and only if $[x_{i-1}+1, h_{i-1}] \cap [x_i+1, h_i] \neq \varnothing$ for each $1 < i \leq N$.

Subtask 2 (5 points)

We can divide the problem into three cases:

- $N = 1$: This case is very simple because there is only one column. The answer is h_1.
- $N = 2$: $[x_1+1, h_1] \cap [x_2+1, h_2]$ should be satisfied. It can be re-written as $\max(x_1, x_2)+1 \leq \min(h_1, h_2)$, which is equivalent to $0 \leq x_1, x_2 < \min(h_1, h_2)$. Therefore, the answer is $\min(h_1, h_2)^2$.
- $N = 3$: In the similar way as the case of $N = 2$, we can deduce that

  $$x_1 < \min(h_1, h_2),\ x_2 < \min(h_1, h_2, h_3),\ \text{and}\ x_3 < \min(h_2, h_3).$$

  Therefore, the answer is $\min(h_1, h_2) \cdot \min(h_1, h_2, h_3) \cdot \min(h_2, h_3)$.

Since the answer can be large, be careful **not to have integer overflow**.

Subtask 3 (10 points)

Because the height of each row is equal, the remaining area is connected regardless of the values of x_i. Therefore, the answer is h^N where $h = h_1 = h_2 = \cdots = h_N$.

Subtask 4 (15 points)

Since $h_{i-1} \leq h_i$ for every i, x_{i-1} does not affect the connectivity between the $(i-1)$-th and the i-th column. In the same way as in subtask 2, we can deduce that $x_i < h_{i-1}$ in order for the $(i-1)$-th and the i-th column to be connected. Therefore, the answer is $h_1 \cdot h_1h_2h_3\cdots h_{N-1}$.

Subtask 5 (60 points)

We can deduce that the problem's condition is equivalent to $\max(x_i, x_{i+1}) + 1 \leq min(h_i, h_{i+1})$ for each $1 \leq i < N$, and it is equivalent to

$$x_i < \min(h_{i-1}, h_i, h_{i+1}) \quad i = 1, 2, \cdots, N$$

where $h_0 = h_{N+1} = \infty$. Therefore, the answer is $\prod_{i=1}^{N} \min(h_{i-1}, h_i, h_{i+1})$ and it can be easily computed in $O(N)$ time.
Shortest solution: 206 bytes

Problem C. Opinion Pool

Problem Idea: Suchan Park (tncks0121)
Preparation: Jaeung Lee (L0TUS)

Subtask 1 (10 points)

Let x be a sequence consisting of numbers x_i, such that $x_i = 1$ if i is in support of the issue, and $x_i = 0$ otherwise.

Let $f(x) = \min_j \left(\sum_{i \in S_j} x_i / |S_j|\right)$. Then $f(x)$ is the maximum possible value of p for such a sequence x. That is, if $p \leq f(x)$, we cannot be certain that everyone is in support of the issue. Now we brute-force over all possible sequences x except $[1, 1, \cdots, 1]$, and find the largest value of $f(x)$. The total time complexity is $O(2^N \cdot \sum_{j=1}^{M} |S_j|)$.

Subtask 2 (15 points)

If more people are in support of the issue, $f(x)$ increases. Formally, let x' be a new sequence obtained from x by changing one of the 0's into a 1. Then $f(x') \geq f(x)$. Thus, we need to compute $f(x)$ only for x of the form $[1, 1, \cdots, 1, 0, 1, \cdots, 1]$ (i.e. there is only one zero). The total time complexity is $O(N \cdot \sum_{j=1}^{M} |S_j|)$.

Subtask 3 (75 points)

Consider a sequence x in which $x_i = 0$ and all the remaining elements are 1. Then $\sum_{i \in S_j} x_i / |S_j|$ is $1 - \frac{1}{|S_j|}$ if $i \in S_j$, and 1 if $i \notin S_j$. Therefore, $\sum_{i \in S_j} x_i / |S_j|$ is minimized iff S_j is the smallest set that contains i.

For each i, let m_i be the smallest $|S_j|$ such that $i \in S_j$. These values can be computed by iterating over all S_j, and for each $i \in S_j$, comparing m_i with $|S_j|$ and overwriting m_i if $|S_j|$ is smaller. The total time complexity is $O(N + \sum_{j=1}^{M} |S_j|)$.

Shortest solution: 401 bytes

Problem D. Three Slices

Problem Idea: Jaehyun Koo (koosaga)
Preparation: Chaejun Lee (juney)

Subtask 1 (10 points)

This subtask can be solved by brute-forcing. For each $M = 1, 2, \cdots$ and $i = 0, 1, \cdots$, check if the three inequalities are satisfied in $O(M)$ time. The total time complexity is $O(N^3)$.

Subtask 2 (20 points)

We can check the three inequalities in $O(1)$ time by maintaining a *prefix sum*. Let $S_{-1} = 0$, and $S_i = S_{i-1} + A_i$ for each i. This S can be computed in $O(N)$ time before running the brute-force. Then $\sum_{j=l}^{r} A_j = S_r - S_{l-1}$, which can be computed in $O(1)$. The total time complexity is $O(N^2)$.

Subtask 3 (70 points)

Beware that we can't simply replace brute-forcing with a binary search to obtain an $O(N \log N)$ solution. Binary searching over M is impossible, because the existence of i for a certain value of M does not necessarily imply the existence of i for smaller values of M. We need a different approach.

Let's call the intervals $[i, i + M - 1]$, $[i + M, i + 2M - 1]$, and $[i + 2M, i + 3M - 1]$ as slices. If the starting index of the second slice, j, is fixed, what is the largest value of M such that the condition is satisfied? If we can answer that in $O(\log n)$, then we get an $O(n \log n)$ solution by iterating over all indices as j.

For each index x, let $L(x)$ be the smallest index such that $\sum_{j=L(x)}^{x} A_j \le K$, and $R(x)$ be the largest index such that $\sum_{j=x}^{R(x)} A_j \le K$. Since all A_i are positive, these L and R are monotonically increasing as x increases. Thus we can compute them by a two-pointer algorithm.

For a given j (the starting index of the second slice) and M, the condition is satisfied iff:

1. $M \le j - L(j - 1)$.
2. $R(j) - j + 1 \ge M$.
3. $R(j + M) - (j + M) + 1 \ge M$.

The first two inequalities limit the values of M as $M \le \min(j - L(j - 1), R(j) - j + 1)$. Among such values of M, we should find the largest value such that $R(j + M) - 2M \ge -j - 1$. The $-2M$ in the inequality makes it hard to find such a value directly, but we can remove it by defining $R'(j) = R(j) - 2j$ so that $R(j + M) - 2M = R'(j + M) + 2j \ge -j - 1$.

Since j is fixed, the problem reduces to finding the largest index $k \in [l, r]$ such that $R'(k) \ge v$. The author and the problem testers have come up with various ways to do it:

- Build a maximum segment tree over R'. To find k, start from the root node and traverse down, which can be done in $O(\log n)$ time.

- Note that v is decreasing for each j because $v = -3j - 1$. Therefore, the set of k such that $R'(k) \geq v$ will keep growing, and no index in the set will disappear. Now figure out exactly how R' grows by sorting the values of $R'(k)$, then maintain a `std::set` of such indices and apply the lower bound operations.
- There is also a solution that maintains a stack consisting only of the indices that can possibly be the answer, and does a binary search inside it.

All of the above methods take $O(n \log n)$ time. Some $O(n \log^2 n)$ solutions pass as well, but those solutions needed to have a low constant factor.

Challenge: How fast can you solve it if four slices are required?

Shortest solution: 806 bytes

Problem E. Factory Balls

Problem Idea: Jaemin Choi (jh05013)
Preparation: Jaemin Choi (jh05013)

Subtask 1 (16 points)

This subtask can be solved with some intuition, but here we'll prove the solution as well. Note that this solution only solves the first subtask, and the full solution uses a completely different approach.

For a piece of equipment E_r (which covers the region r), let $e_1, u_1, e_2, u_2, \cdots, e_k, u_k$ be the sequence of indices i such that E was equipped or unequipped at the i-th operation. (e denotes equipping, and u denotes unequipping.) If E was never equipped, let's say $k = 0$.

Let's say a region r is **painted** if the ball is immersed into a paint can while the region is not covered by E_r. The optimal solution has the following properties:

- For each region r, if $k \geq 1$, r is never painted after e_1. Why? If r is painted after e_1, then we don't need to equip E_r at e_1 and unequip it at u_1, saving two operations.
- For each region, $k \leq 1$. Why? Suppose $k \geq 2$. Since r is never painted after e_1, we don't need to unequip E_r at u_1 and equip it at e_2, saving two operations.
- The above observation implies that once any piece of equipment is unequipped, you should never immerse the ball into a paint can anymore. There is no reason to equip another piece of equipment either, because you can't paint the ball anyway.
- If there is no paint can used (assuming that's actually possible), there shouldn't be any piece of equipment used either.
- If the last paint can used has the color c, each equipment E_r is equipped and unequipped once iff the target color of the region r is not c.
- Let d be the number of different target colors except 1. Then the number of paint cans used is $d + 1$ if the last paint can used has the color 1, and otherwise d.

Using the last three properties, we can try each color as the color of the last paint can used, and compute the minimum number of operations for that case.

Subtask 2 (84 points)

We would like to apply a breadth-first search, but there are too many states to run a BFS on: each region has one of the K colors, and each piece of equipment is either equipped or unequipped, leading to $K^N 2^M$ states in total.

However, if the color of a certain region does not match the target color, then the exact color does not matter. All we need for each region is whether the color matches the target or not. Therefore each region has only two "colors." This reduces the number of states to 2^{N+M} and the time complexity of the BFS to

$O(2^{N+M}(KN+M))$. To represent each state, we can use a pair of bitmasks (or a single bitmask if you combine them).

Using bitwise operators more cleverly, We can do even better by immersing the ball into a paint can in $O(1)$ time, leading to $O(2^{N+M}(K+M))$.

Shortest solution: 1041 bytes

Problem F. AND PLUS OR

Problem Idea: Jaehyun Koo (koosaga)
Preparation: Jaehyun Koo (koosaga)

Subtask 1 (14 points)

Enumerating all possible i, j suffices for $O(4^N)$ solution.

Subtask 2 (17 points)

Consider each index $i \in [0, 2^N - 1]$ as a subset of $\{0, 1, \ldots, N-1\}$. We also denote a_i as a set function $a(i)$. Let $x = i \cap j$, $y = i - j$, $z = j - i$. Then we want to find three disjoint sets x, y, z such that $a(x \cup y) + a(x \cup z) < a(x) + a(x \cup y \cup z)$, or $a(x \cup y) - a(x) < a(x \cup y \cup z) - a(x \cup z)$.

For a fixed x, y, let $f(z) = a(x \cup y \cup z) - a(x \cup z)$. If such z exists, $f(\emptyset) < f(z)$.

Let $z_j = \{i_1, i_2, \ldots, i_j\}$ ($z_{|z|} = z$). Then, there exists some $1 \le j \le |z|$ such that $f(z_{j-1}) < f(z_j)$. If we put $x' = x \cup z_{j-1}$, we have $a(x' \cup y) - a(x') < a(x' \cup y \cup \{i_j\}) - a(x' \cup \{i_j\})$.

In conclusion, there exists an answer of the form $(x, y, \{e\})$ where e is an element that does not belong to $x \cup y$. The number of such tuples is at most $3^N \times N$, so we can enumerate all of them. For implementation details, see Suboptimal Solution section in this post.

Subtask 3 (69 points)

If you swap i, j from the above solution and proceed identically, you can see that there exists an answer of the form $(x, \{e\}, \{f\})$, where $e \neq f$ and $\{e, f\} \cap x = \emptyset$. The number of such tuples is at most $2^N \times N^2$, so we can enumerate all of them and obtain an $O(2^N N^2)$ solution.

Shortest solution: 371 bytes

Problem G. Endless Road

Problem Idea: Jaehyun Koo (koosaga)
Preparation: Jaehyun Koo (koosaga)

Subtask 1 (7 points)

Direct implementation of the described algorithm suffices. For each turn, we can find the member in $O(CN)$ time and then plant the flower in $O(C)$ time, where $C \leq 100$. The time complexity is $O(CN^2)$.

Subtask 2 (9 points)

Use coordinate compression. Let's call each length 1 interval after coordinate compression an *atomic interval*. In other words, coordinate compression partitions the road into at most $2N$ atomic intervals.

For each member, we maintain the total length of the unplanted intervals $remain[i]$, so that we can find the member to plant in $O(N)$ time. When a member plants the flowers, we plant them for each atomic interval one by one. Planting for some atomic interval implies reducing the value $remain[i]$ for all members containing that interval, which can be processed in $O(N)$ time. This happens for at most $O(N)$ times, so we obtain an $O(N^2)$ algorithm.

Subtask 3 (17 points)

By the condition of the subtask, if we sort the members by pair (L_i, R_i), we have $L_i \leq L_{i+1}, R_i \leq R_{i+1}$. If we remove an atomic interval, the members for which we should reduce the value $remain[i]$ form an interval over that sorted sequence. This interval can be found with binary search.

Thus, we want to find the minimum over $remain[i]$, find atomic intervals newly removed, and for each of them do a range decrement operation. The first and third operations can be done by a segment tree with lazy propagation in $O(\log N)$ time. The second operation can be done by maintaining a set of unplanted atomic intervals. This in total gives an $O(N \log N)$ algorithm. To avoid processing a member twice, set $remain[i]$ to a sufficiently large number after the turn. Keep in mind that ties should be broken by the original indices of the members.

Subtask 4 (13 points)

We need a following key observation:

Lemma 1. *For two different members i, j, if $L_i \leq L_j, R_j \leq R_i$, then member i always works earlier than member j.*

Proof. Observe that $remain[i] \leq remain[j]$ always holds for such a pair of members. If $remain[i] < remain[j]$, the statement holds. If $remain[i] = remain[j]$, then $i < j$ by the non-decreasing length condition in the problem. □

By the condition of the subtask, for every pair of members, their intervals are either disjoint ($A \cap B \neq \emptyset$) or nested ($A \subseteq B$ or $B \subseteq A$). If we define the parent of the member x as the smallest interval containing x,

then we can represent the members as a rooted forest. By the lemma, we will be only removing leaves from the rooted forest. For each leaf, we can compute the length of the remaining intervals, since it's simply the length of itself minus the lengths of all children. You can find the forest with stacks, and maintain leaves with priority queues. This subtask can be solved in $O(N \log N)$ time.

Subtask 5 (54 points)

The full solution takes a hybrid approach from preceding subtasks. Using the lemma from Subtask 4, we can take the set of members where there is no other member that certainly works earlier. Those members satisfy the condition of Subtask 3, so we can apply the solution to it.

Let's call a member *candidate* if it does not contain any other members. In other words, it does not certainly work later than other members. To compute the set of candidates, we sort all members in the increasing order of L, where ties are broken by the decreasing order of R. Then we scan in the decreasing order of sorted indices ($N-1$ to 0), adding a member if its R_i is smaller than all other scanned intervals. In other words, a member in the i-th sorted order is a candidate if the suffix minimum of index i is different from that of $i+1$. Be careful that the index here differs from the index given in the input.

Given that we know the candidates, we can modify the solutions of subtask 3. Maintain the candidates in `std::set`, and for each member not in the candidate, set $remain[i]$ to a sufficiently large number. Then the intervals could be found with an `lower_bound` operation in the set, and all other things can be done accordingly.

However, the set of candidates changes after we process each member. Specifically, we delete the candidate after the turn, and the removal of the candidate creates other possible candidates. To solve this, it's helpful to get back to the naive algorithm of finding candidates: It computes the suffix minimum and finds where it changes. As the member is removed (or, the member's value becomes infinity, in terms of suffix minimum), we have to reconstruct the position where it changes.

Let's store the list of endpoints in the range minimum segment tree. Then we repeatedly find the minimum element m (rightmost in case of a tie), add it to the candidate, and start ignoring the position m and left of it. This algorithm finds the candidate in the same way the above algorithm does. The only difference is that this runs in the backward direction. Now, if we remove candidate i from the set, we find the candidate j that is adjacent to the left of i from `std::set`. Now, we start in a state where j is the latest candidate found and repeat the algorithm. We finish the algorithm when we find the candidate already in the set, or we just run out of new candidates. This algorithm runs in $\log N$ time for each found candidate, and we don't find the same candidate twice, so the total complexity is $O(N \log N)$ for candidate reconstruction. If we finish processing each member, we have to set its value infinity in the segment tree.

When activating a new member, we don't know its actual $remain[i]$ value because we did not maintain it. We keep a segment tree that maintains the sum of the unplanted intervals. When we remove an atomic interval, we modify the set and the segment tree simultaneously. Then, in time of activation, we can compute the $remain[i]$ value for new candidates.

Summing it down:

- We keep a `std::set` for maintaining the candidates.
- We keep a minimum segment tree with lazy propagation to maintain $remain[i]$ array for candidates.
- We keep a `std::set` and sum segment tree (Fenwick tree) to maintain the unplanted atomic intervals.
- We keep a minimum segment tree to reconstruct the candidates.

The total time complexity is $O(N \log N)$.

Remark. Theoretically, the monotone condition on $R_i - L_i$ is unnecessary. If we can select *any* member in case of tie, then the above solution holds. However, it is hard to prepare good tests and checker in such setting, so we needed a tie-breaker condition. Unfortunately, simply tie-breaking by lowest index makes Lemma 1 very tedious (possibly even wrong, we don't want to investigate). We wrote the statement in a way that the tie-break should prefer shorter interval, and still sounds as natural as possible.

Shortest solution: 4330 bytes

Problem H. Fake Plastic Trees 2

Problem Idea: Jaehyun Koo (koosaga)
Preparation: Jaehyun Koo (koosaga)

Subtask 1 (9 points)

Try all subsets of edges. Then, check if the condition is satisfied, with graph search algorithms. The time complexity is $O(2^N \times N)$.

Subtask 2 (24 points)

We use dynamic programming on the tree, where we solve a problem for all rooted subtrees and recursively merge the solutions. Root the tree at vertex 1. Let T_v be a subtree rooted at vertex v.

Let $DP[v][k][x]$ be a boolean value which is true if there are k **closed** subtrees and one **extendable** subtree that has the weight sum of x in T_v. An extendable subtree is either empty, or contains a vertex v. It is *extendable* in a sense that it may contain other vertices and increase its size. If an extendable subtree is empty, then all subtrees under v are closed, including the one that contains the vertex v.

Then, we have the following recursive rule:

- **Base case:** For a leaf v, you have two cases: add v to an extendable, or closed subtree. If you add it to a closed subtree, then $L \le A_v \le R$ should hold.
- **Merge:** For two children w_1, w_2 of v, we will shrink them to obtain one child w equivalent to both children. If $DP[w_1][k_1][x_1]$ and $DP[w_2][k_2][x_2]$ are both true, then we can consider $DP[w][k_1+k_2][x_1+x_2]$ to be true.
- **Extend:** For a vertex v with a single child w, we will add the vertex v as a parent. If $DP[w][k][x]$ is true, then $DP[v][k][x + A_v]$ is true. If $L \le x + A_v \le R$, then we may want to close the subtree rooted in v, which makes $DP[v][k+1][0]$ true.

Naively implementing this yields $O(NK^2R^2)$ time complexity, where the bottleneck comes from Merge rule. However, note that $k_1 \le |T_{w_1}|$ and $k_2 \le |T_{w_2}|$, which means the iteration count can be bounded as $c \times R^2 \times \sum_{v \in T}(\min(K, |T_{w_1}|) \times \min(K, |T_{w_2}|))$ for some constant c. Here, the following well-known lemma holds.

Lemma 2. *Given a binary tree T, $\sum_{v \in T} \min(K, |T_{w_1}|) \times \min(K, |T_{w_2}|) = O(|T|K)$.*

Proof. Use double counting. $\min(K, |T_{w_1}|)$ is the number of vertices in the left subtree with K highest DFS preorder numbers, $\min(K, |T_{w_2}|)$ is the number of vertices in the right subtree with K lowest DFS preorder numbers. The number of pairs of such vertices is at most the number of vertices with DFS preorder number difference at most $2K$, which is $O(|T|K)$. □

Thus the dynamic programming runs in $O(NKR^2)$ time and the subtask is solved. (Further optimization is possible with bitsets or FFT, but is not required.)

Subtask 3 (10 points)

The graph is a line. Let $DP[i][j]$ be a boolean value that is true if we can partition the vertex $1 \dots j$ into i components. By fixing the end of the last component, we can obtain simple recurrences. The time complexity is $O(N^2K)$ and could be improved to $O(NK)$ with prefix sums. Unfortunately, this solution does not extend to the full solution.

Subtask 4 (57 points)

We continue the solution from Subtask 2.

It is convenient to consider the DP value as a set of weights. Let $S(v,k) = \{x | DP[v][k][x]\}$. For two set A, B, define the *sumset* $A + B = \{a + b | a \in A, b \in B\}$. With this notation, we can revisit the recursive rule in much cleaner way.

- **Base case:** $S(v,0) = \{A_v\}$ (if $A_v \le R$), $S(v,1) = \{0\}$ (if $A_v \in [L,R]$).
- **Merge:** $S(w,k) = \bigcup_{0 \le i \le k} S(w_1,i) + S(w_2,k-i)$.
- **Extend:** $S'(v,k) = S(w,k) + \{A_v\}$. $S(v,k) = S'(v,k)$. If $[L,R] \cap S'(v,k) \neq \emptyset$, then $S(v,k+1) := S(v,k+1) \cup \{0\}$.

And we want to determine if $0 \in S(1,K+1)$. This algorithm is exactly the Subtask 2, but here the size of S may grow exponentially.

Before continuing, we need the following key lemma.

Lemma 3. $\max S(v,k) - \min S(v,k) \le k(R-L)$.

Proof. The sum of weights in closed subtrees is in the range $[kL, kR]$, and the sum of weights in T_v is fixed. □

Note that we are only interested in if there is an element in some interval of length $R-L$. ($0 \in S$ is equivalent to $[-(R-L), 0] \cap S \neq \emptyset$).

Consider the following algorithm $reduce(S)$ for some set S. Let $a, b, c \in S$ and $a < b < c, c - a \le R - L$. Then, we can remove the middle element b, while not changing any interval query result of length $R-L$. An easy way to implement this algorithm is to place buckets of length $R - L + 1$ and take only the minimum and maximum elements over the buckets. For any set $S(v,k)$ the algorithm can be implemented in time $O(|S(v,k)| + k)$ and can guarantee $|reduce(S(v,k))| \le 2k$.

To obtain a polynomial-time algorithm, we want to apply $S(v,k) := reduce(S(v,k))$ after each Merge and Extend case. We perform the sumset operation for $O(NK)$ times (by Lemma 2), and each operation takes $O(K^2)$ time, thus the final time complexity is $O(NK^3)$ and the problem is solved. However, to prove its correctness, we have to show that the reduce operation preserves the query result even after applying the *sumset* operation. This can be proved in the following step.

Definition 1. *For $b \in \mathbb{N}$ and $A \subseteq \mathbb{N}$, define*

- $apx^-(b, A) := \max\{a \in A | a \leq b\}$
- $apx^+(b, A) := \min\{a \in A | a \geq b\}$

We say A Δ-approximates B if $A \subseteq B$ and for every $b \in B$, $apx^+(b, A) - apx^-(b, A) \leq \Delta$.

Lemma 4. *If A is a Δ-approximation of B, then $[x, x + \Delta] \cap A = \emptyset$ iff $[x, x + \Delta] \cap B = \emptyset$.*

Proof. $\leftarrow$ is trivial. Suppose $[x, x + \Delta] \cap A = \emptyset$ and $y \in [x, x + \Delta] \cap B$. Then $y \in B$ and $apx^+(y, A) - apx^-(y, A) > \Delta$. □

Lemma 5. *If A_1 Δ-approximates B_1 and A_2 Δ-approximates B_2, then $A_1 + A_2$ Δ-approximates $B_1 + B_2$.*

Proof. Consider all $x + y \in B_1 + B_2$. If $x + y \in A_1 + A_2$ or $x \in A_1$ or $y \in A_2$, then the statement holds. Otherwise, Let $x_1 = apx^-(x, A_1)$, $x_2 = apx^+(x, A_1)$, $y_1 = apx^-(y, A_2)$, $y_2 = apx^+(y, A_2)$, and WLOG $x_2 - x_1 \leq y_2 - y_1$.
Let Z be the nondecreasing sequence $\{x_1 + y_1, x_2 + y_1, x_1 + y_2, x_2 + y_2\}$. Observe that $Z \subseteq A_1 + A_2$, $x_2 - x_1 \leq \Delta, y_2 - y_1 \leq \Delta, (y_2 - y_1) - (x_2 - x_1) \leq \Delta$. We can see any element $b \in [x_1 + y_1, x_2 + y_2]$ have $apx^+(b, A_1 + A_2) - apx^-(b, A_1 + A_2) \leq \Delta$, which is the case for $x + y$. □

Lemma 6. *reduce(S) is a $(R - L)$-approximation of S.*

Proof. Immediate from the algorithm. □

Shortest solution: 4086 bytes