

Editorial

2026 ICPC Bolivia National Contest
Bolivian Programming Competition, Phase 2

2026

Note: this document contains the complete solution of each problem, with progressive hints in the style of a Codeforces editorial.

Problem A — Borders

Estimated difficulty: Easy-medium (approx. 1500) • Topics: Computational geometry and Union-Find

Hints

Hint 1. Each conflict only joins two communities; it never splits them. That sounds like a problem where it pays to maintain a disjoint-set structure (Union-Find / DSU) indexed by town, and to store *something* associated with each component.

Hint 2. You do not need to store all the towns of a community to answer the question. The area of the territory depends only on its convex hull, so it is enough to store, for each component, the vertices of its own hull.

Hint 3. When you merge two communities, is it enough to put the two boundaries (the two lists of vertices) together to get the boundary of the union? Draw two nearby triangles and join them: can a vertex of one triangle end up *inside* the resulting pentagon/hexagon?

Hint 4. Yes, it can happen (look at example 3 of the statement). Therefore, after joining two components you must recompute the convex hull of the union of their vertices *from scratch*, not simply concatenate the two lists. A *monotone chain* (Andrew) algorithm does this in $O(k \log k)$, where k is the number of vertices being joined.

Hint 5. The area is requested as twice the real area precisely so that it can be computed with integers using Gauss's formula (*shoelace*) without worrying about fractions. How large, in the worst case, is the sum of the sizes of the sets recomputed over all conflicts? With $N \leq 2000$, that total bound ends up being manageable even if each merge costs $O(N \log N)$.

Solution

We maintain a Union-Find structure over the N towns. The important part is what information each component stores: we do **not** store all the towns that belong to it, only the vertices of its current convex hull. Initially, each town is its own community and its “hull” is the single point.

When a conflict between towns a and b arrives:

- If they are already in the same component ($\text{Find}(a) == \text{Find}(b)$), there is nothing to do: the hull does not change.
- If they are in different components, we merge them with the DSU (union by size/rank to keep the amortized complexity), and we **recompute from scratch** the convex hull of the union: the vertices of both hulls are concatenated into a single list of points and *monotone chain* is run over that whole list. The result replaces the hull stored at the root of the merged component; the component that is no longer a root can clear its list (it is never queried again).

Why do we need to recompute instead of simply combining the two boundaries (for example, “splicing” them sorted by angle)? Because a vertex that was extreme in *its own* community may end up strictly inside the hull of the union. Example 3 of the statement is exactly this case: two triangles that, when merged, form a pentagon, and one of the six original vertices ends up inside. There is no cheap way to know in advance which vertex survives without, essentially, running the convex hull algorithm again on the merged set.

Once we have the vertices of the hull (in order, either clockwise or counterclockwise), the signed area is computed with Gauss’s formula:

$$2 \cdot \text{Area} = \left| \sum_{i=0}^{k-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

where indices are taken modulo k (the vertex after the last one is the first). If the hull has fewer than 3 vertices (a community of size 1 or 2, or all of its points collinear), the area is 0. Since all coordinates are integers, this expression is computed exactly with 64-bit integer arithmetic (`long long`), and that is why the problem asks for *twice* the area: this way the answer is always an integer.

After each conflict (whether or not an actual merge happened), we print twice the area of the hull of the component that now contains a (and b ; it is the same one).

Key fragment of the merge (adapted from `sols/main.cpp`):

```

1  int root = dsu.Union(x, y);
2  int other = (root == x) ? y : x;
3
4  vector<Point> merged = hull[root];
5  for (int k = 0; k < hull[other].size(); k++)
6      merged.push_back(hull[other][k]);
7  hull[root] = convexHull(merged);
8  hull[other].clear();

```

Complexity. Time: $O(N^2 \log N + M)$ Space: $O(N)$

Each of the $N - 1$ conflicts that actually merges two components recomputes a hull over, in the worst case, $O(N)$ points in $O(N \log N)$; the rest of the conflicts (up to $M - (N - 1)$ of them) are already in the same component and cost $O(1)$. With $N \leq 2000$ this is perfectly feasible within 2 seconds.

Problem B — MiniMax

Estimated difficulty: Medium (approx. 1950) • *Topics: Data structures, greedy, binary lifting*

Hints

Hint 1. Fix a starting point i and forget about the queries for a moment. For that single i , what is the longest valid segment (with $\max - \min \leq k$) that can be formed starting at i ? Is that segment unique?

Hint 2. To solve a query $[l, r]$ with the fewest segments, it pays to make each segment as long as possible. An exchange argument shows that it is never better to cut earlier than necessary: if a segment can keep growing without breaking the condition, cutting it earlier never helps and may force you to use an extra segment later on.

Hint 3. Define $\text{next}[i]$ as the first position that, if added to the segment starting at i , makes $\max - \min$ exceed k (that is, $[i, \text{next}[i] - 1]$ is the longest valid segment starting at i). Computing $\text{next}[i]$ for a single i by brute force costs $O(n)$; doing it that way for all n values of i would be $O(n^2)$. Can you compute the *whole* array $\text{next}[\cdot]$ in a single pass of $O(n)$ total, with two pointers and something that gives you the maximum and minimum of a window in amortized $O(1)$?

Hint 4. With $\text{next}[\cdot]$ computed, each query could be answered by jumping $l \rightarrow \text{next}[l] \rightarrow \text{next}[\text{next}[l]] \rightarrow \dots$ until you go past r , counting the jumps. But that costs up to $O(n)$ per query, and with up to 10^5 queries that is too slow. Which classic technique allows you to *double* the length of a jump at each level to traverse a chain of jumps in $O(\log n)$?

Hint 5. Build a *binary lifting* table $\text{up}[j][i]$ = the position reached after applying the next jump 2^j times from i (with $\text{up}[0][i] = \text{next}[i]$ and $\text{up}[j][i] = \text{up}[j-1][\text{up}[j-1][i]]$). To answer $[l, r]$, go through the levels from highest to lowest and take the jump of 2^j segments whenever the destination does not go past r ; at the end add 1 for the last segment, which is truncated at r .

Solution

The key observation is that the optimal partition can always be built **greedily**: to minimize the number of segments in $[l, r]$, each segment should extend as far as possible before cutting. If at some point we cut a segment before it actually becomes invalid, we never gain anything (the remaining part to the right only becomes harder to cover), and in the worst case we end up needing an additional segment. Therefore, it is enough to know, for each position i , the longest valid segment that starts there.

We define $\text{next}[i]$ as the first position (0-indexed) such that $[i, \text{next}[i] - 1]$ is valid (difference $\leq k$) but adding the element at $\text{next}[i]$ makes it invalid. This value is computed for *all* i in a single pass using two pointers: we maintain a window $[\text{left}, \text{right}]$ and two monotonic queues – an increasing one whose front is always the index of the minimum of the window, and a decreasing one whose front is the index of the maximum. Every time the window stops being valid (the

max – min of the queues exceeds k), we set $\text{next}[\text{left}] = \text{right}$ and advance left, popping from the queues the indices that fall outside the new window. Each index enters and leaves each queue at most once, so the total cost is $O(n)$.

With $\text{next}[\cdot]$ computed, each query $[l, r]$ could be answered by repeatedly jumping with next until going past r , but that is $O(n)$ per query in the worst case (for example, an array that alternates $-10^9/10^9$, where every valid segment has length 1). To speed this up we use **binary lifting**: we precompute a table $\text{up}[j][i] = \text{position after } 2^j \text{ jumps of next from } i$, with $\text{up}[0][i] = \text{next}[i]$ and $\text{up}[j][i] = \text{up}[j-1][\text{up}[j-1][i]]$. To answer a query, we go through the levels j from highest to lowest: if $\text{up}[j][\text{pos}] \leq r$ (0-indexed), we take the jump (advance pos and add 2^j to the segment counter); at the end, we add 1 for the last segment, which reaches r truncated.

Complexity. Time: $O((n + q) \log n)$ Space: $O(n \log n)$

A fragment of the computation of $\text{next}[\cdot]$ (equivalent to $\text{up}[0][\cdot]$) with the two monotonic queues:

```

1  std::deque<int> minimum, maximum;
2  int left = 0;
3  for (int right = 0; right < n; ++right) {
4      while (!minimum.empty() && a[minimum.back()] >= a[right])
5          minimum.pop_back();
6      minimum.push_back(right);
7      while (!maximum.empty() && a[maximum.back()] <= a[right])
8          maximum.pop_back();
9      maximum.push_back(right);
10
11     while (a[maximum.front()] - a[minimum.front()] > k) {
12         up[0][left] = right; // next[left] = right
13         if (minimum.front() == left) minimum.pop_front();
14         if (maximum.front() == left) maximum.pop_front();
15         ++left;
16     }
17 }
```

And the answer to a query $[l, r]$ (already 0-indexed) using the binary lifting table:

```

1  int position = l;
2  int answer = 1; // the last segment, truncated at r
3  for (int level = levels - 1; level >= 0; --level) {
4      if (up[level][position] <= r) {
5          position = up[level][position];
6          answer += (1 << level);
7      }
8  }
```

Problem C — Firewood in Corani

Estimated difficulty: Easy (approx. 1300) • Topics: Implementation, data structures
(`std::set`)

Hints

Hint 1. Forget about the whole pyramid for a moment and look only at the base. When Pakai removes a position, it splits the base into pieces. A log in an upper row depends on two consecutive supports: what must happen at the base for a log up there to stay standing?

Hint 2. A log in an upper row survives if and only if the two base positions it depends on (and, transitively, the whole range of base positions below it) remain intact. In other words: a **contiguous** stretch of intact base positions supports its own sub-pyramid, independent of the other stretches.

Hint 3. How many logs in total does an intact stretch of length L support (counting the base stretch itself and everything stacked on top)? It is the sum $L + (L - 1) + \dots + 1$: a triangular number.

Hint 4. Each removal splits an existing stretch into two smaller stretches (or leaves a single one, if the removal is exactly at an end). If at every moment you know the length of the stretch containing the removed position, you can update the total answer in constant time per operation, without rebuilding anything. You need a structure that gives you, for any position, its nearest removed neighbors to the left and to the right, in logarithmic time: a `std::set` with the removed positions (plus two sentinels at 0 and $N + 1$ so the ends are not special cases) solves exactly that.

Solution

The key observation is that a log in an upper row is standing if and only if a whole contiguous range of base positions below it remains intact. Therefore, the pile decomposes in real time into **maximal stretches of intact base positions**, and each stretch supports its own sub-pyramid, without interacting with the other stretches.

An intact stretch of length L supports exactly

$$T(L) = 1 + 2 + \dots + L = \frac{L(L + 1)}{2}$$

logs in total (the base stretch itself, plus a row of $L - 1$ on top, and so on, up to the apex of that sub-pyramid), with $T(0) = 0$. The answer at any moment is simply the sum of $T(L)$ over all living intact stretches, and at the beginning (no removals) there is a single stretch of length N , that is, $T(N)$.

Each removal of a position p takes the stretch containing p (say it goes from $a + 1$ to $b - 1$, with length $b - a - 1$, where a and b are the nearest removed positions to the left and right of p) and replaces it with two new stretches: one of length $p - a - 1$ and another of length $b - p - 1$. The answer is updated in $O(1)$:

$$\text{answer} -= T(b - a - 1); \quad \text{answer} += T(p - a - 1) + T(b - p - 1).$$

To find a and b quickly, we maintain a `std::set` with all the positions already removed, plus two fixed sentinels at 0 and $N + 1$ (so a stretch always has valid neighbors, even if it touches a real end of the base). With `lower_bound` we locate b in $O(\log Q)$, and its predecessor in the set is a . Then we insert p into the set and move on to the next removal.

```

1  std::set<i64> removed{0, n + 1};
2  i64 remaining = triangular(n);
3
4  for (int i = 0; i < q; ++i) {
5      i64 p; std::cin >> p;
6      const auto right_it = removed.lower_bound(p);
7      const i64 right = *right_it;
8      const i64 left = *std::prev(right_it);
9
10     remaining -= triangular(right - left - 1);
11     remaining += triangular(p - left - 1);
12     remaining += triangular(right - p - 1);
13     removed.insert(right_it, p);
14
15     std::cout << remaining << '\n';
16 }
```

An important caveat: $T(L)$ and the accumulated answer can reach the order of $N^2 \approx 10^{18}$, so 64-bit integers (`int64_t/long long`) must be used throughout the computation, never 32-bit `int`.

Complexity. Time: $O(Q \log Q)$ Space: $O(Q)$

Problem D — Cookies for the Feed

Estimated difficulty: Easy (approx. 1100) • Topics: Implementation / simulation

Hints

Hint 1. Each letter of the alphabet is completely independent of the others: A molds are never used to bake a B. Think about the problem letter by letter instead of word by word.

Hint 2. For a fixed letter, the only thing that matters about each batch is *how many times* that letter appears in that batch's word. The rest of the word is irrelevant for that letter.

Hint 3. A mold used in batch i cannot be reused in batch $i + 1$ (it is still cooling down), but it can be in batch $i + 2$. That is, in any pair of *consecutive* batches the molds used cannot be shared: different molds are needed for each use in those two batches.

Hint 4. For a letter x , look at each pair of consecutive batches $(i - 1, i)$ and add up how many times x appears in those two words together. The worst pair (the one that requires the most molds of that letter simultaneously) determines how many molds of x must be bought.

Solution

Since each letter uses its own set of molds, it is enough to solve the problem separately for each of the 26 letters and then add up the results.

Let us focus on a letter x . Let $c_i(x)$ be the number of times x appears in the word of batch i (with $c_0(x) = 0$, since no mold had been used before the first batch). The rule in the statement says that the molds of x used in batch $i - 1$ are still cooling down during batch i , so they are useless for that batch: *new* molds (or molds that are already cold, from two or more batches back) are needed to cover the $c_i(x)$ uses of the current batch.

This means that in any batch i , the number of molds of x that are “committed” (used in the previous batch or in the current one) is $c_{i-1}(x) + c_i(x)$, and that amount can never exceed the total number of molds of x we bought. Therefore, the minimum number of molds of x needed is

$$M(x) = \max_{1 \leq i \leq N} (c_{i-1}(x) + c_i(x)),$$

that is, the largest value over any two consecutive batches. It is enough to go through the batches once, keeping the per-letter count of the previous batch, and update $M(x)$ with $c_{i-1}(x) + c_i(x)$ at each step.

The final answer is simply $\sum_x M(x)$ over the 26 letters.

The first example (ANA, LIA, ANA) shows the idea: for the letter A, $c_1 = 2$, $c_2 = 1$, $c_3 = 2$, and the maximum of consecutive sums is $\max(2 + 1, 1 + 2) = 3$; for N, $\max(1 + 0, 0 + 1) = 1$; for L and I, 1 each. In total $3 + 1 + 1 + 1 = 6$ molds, which is the expected answer.

Complexity. Time: $O(L + 26N)$ Space: $O(L_{\max} + 26)$

where L is the sum of the lengths of all the words, N the number of batches and $L_{\max}$ the length of the longest word (there is no need to store all the words at once, only the current batch and the previous one).

The core of the algorithm is very short:

```

1  std::array<int, 26> previous{};
2  std::array<int, 26> needed{};
3
4  for (int i = 0; i < n; ++i) {
5      std::string word;
6      std::cin >> word;
7      std::array<int, 26> current{};
8
9      for (char letter : word)
10         ++current[static_cast<std::size_t>(letter - 'A')];
11
12     for (std::size_t letter = 0; letter < current.size(); ++letter)
13         needed[letter] = std::max(needed[letter],
14                                     previous[letter] + current[letter]);
15     previous = current;
16 }

```

Problem E — Waso Number

Estimated difficulty: Easy-medium (approx. 1700) • Topics: Combinatorics (combinations with repetition / stars and bars)

Hints

Hint 1. A waso number is simply a non-negative integer whose digits, read from left to right, never decrease (for example, 1, 23, 113, 99). Before thinking about how to find the k -th one, ask yourself: for a fixed length L , how many waso numbers with exactly L digits exist?

Hint 2. A waso number with L digits (no leading zero) is the same as choosing, regardless of order, a multiset of size L from the digits $\{1, \dots, 9\}$: once the multiset is chosen, there is a unique way to arrange it in non-decreasing order. Counting multisets is the classic problem of *combinations with repetition*, which is solved with the *stars and bars* technique.

Hint 3. Since k can be as large as 10^{18} , the answer can have hundreds of digits: you cannot generate the waso numbers one by one nor store them in a 64-bit integer. You first need to **fix the length** L of the number you are looking for, comparing k against cumulative sums of the counts per length (and those cumulative sums also have a closed formula; there is no need to add them one by one).

Hint 4. Once L is fixed, build the number **digit by digit**, from left to right: at each position try the smallest valid digit (greater than or equal to the last one chosen) and use the count of how many ways there are to complete the remaining positions with digits $\geq d$ to decide whether the k -th number falls within that block or whether it must be discarded (subtracting its size from k) and $d + 1$ tried. It is the same idea as finding the k -th element of a lexicographic order, but counting with binomials instead of explicitly going through all the possibilities.

Hint 5. All the binomials you need to compute are small ($\binom{n}{r}$ with $r \leq 9$ and n bounded by a few hundred): they fit perfectly in a 64-bit integer (**long long**). The only thing that can be huge (up to ~ 410 digits) is the **final answer**, and that one is built directly as a **string**, never as a number.

Solution

A waso number is a non-negative integer whose digits, read from left to right, form a non-decreasing sequence. We will count and build the answer in three steps.

Step 1: the trivial one-digit cases. The first 10 waso numbers are, in order, 0, 1, 2, $\dots$, 9 (a single digit is always trivially non-decreasing, and the statement considers 0 as the first one). So if $k \leq 10$, the answer is simply $k - 1$.

Step 2: count by length and fix L . For $k > 10$, we subtract those 10 cases and look for the length $L \geq 2$ of the number we want. A waso number with L digits and no leading zero is a non-decreasing sequence of L digits from $\{1, \dots, 9\}$, which is equivalent to choosing a multiset of size L from those 9 symbols. By *stars and bars* there are exactly

$$\binom{L+8}{8}$$

such sequences. We go through $L = 2, 3, 4, \dots$ accumulating this count and subtracting it from k until the count for the current length is at least the remaining k : that is the correct length.

Step 3: build the answer digit by digit. With L fixed, we build the number from left to right. Let

$$\text{seqCount}(\ell, d) = \binom{\ell + (9 - d)}{9 - d}$$

be the number of non-decreasing sequences of length ℓ using only digits from $\{d, \dots, 9\}$. At each position, with low equal to the last digit placed (or 1 at the first position, to avoid a leading zero), we try $d = \text{low}, \text{low} + 1, \dots, 9$: if $k \leq \text{seqCount}(\text{remaining}, d)$, we place that digit and move on; otherwise, we subtract that count from k and try the next d .

```

1  long long seqCount(long long len, int d) {
2      return nCr((int)len + (9 - d), 9 - d);
3  }
4  ...
5  string res;
6  int low = 1;
7  for (long long pos = 0; pos < L; pos++) {
8      for (int d = low; d <= 9; d++) {
9          long long cnt = seqCount(L - pos - 1, d);
10         if (k <= cnt) { res += char('0' + d); low = d; break; }
11         k -= cnt;
12     }
13 }
```

All the binomials involved have $r \leq 9$ and n bounded by a few hundred (for $k \leq 10^{18}$ the maximum length turns out to be $L = 410$), so they are computed without trouble with a Pascal table in `long long` precomputed only once; no arbitrary-precision arithmetic is needed for the counts, only to *print* the answer (which is built directly as a character string).

Complexity. Time: $O(L_{\max}) \approx O(410)$ per query, after an $O(500 \times 10)$ precomputation of the binomial table Space: $O(500 \times 10)$ for the table + $O(L_{\max})$ for the answer

Problem F — Lights for the SanSiCup

Estimated difficulty: Very hard (approx. 2450) • Topics: Linear algebra over $\mathbb{F}_2$ (XOR basis) and linear codes

Hints

Hint 1. A reachable light configuration is exactly the XOR of the masks of some subset of the original controls. That is, the set of reachable configurations is the *span* (over $\mathbb{F}_2$) of the N vectors $a_1, \dots, a_N$. The only thing the problem cares about in a set of controls is that span, not the individual controls. Which control(s) are really essential in order not to lose dimensions of that space?

Hint 2. Build an XOR basis of the a_i by bit-by-bit Gaussian elimination, from most significant to least significant. It is key that each element of the basis is literally one of the original controls (not a reduced combination): this way, “losing a basis element” means losing a real, concrete physical control. Each control that does *not* enter the basis is dependent: it can be written as the XOR of a subset of the r basis elements, and that subset (a vector of r bits) is its “signature”.

Hint 3. Think about what it means, for a basis element B_i , that *no* other control (existing or new) references it in its signature: if B_i is lost, nothing can rebuild that direction and some configuration becomes unreachable. And if two basis elements B_i, B_j are referenced by **exactly** the same set of other controls (same signature), losing precisely those two (B_i and B_j) is also fatal, even though each of them separately is “accompanied”.

Hint 4. The exact condition to survive the loss of any up to 2 controls is: extending each signature with the new controls that are built, all signatures must have weight ≥ 2 and be **pairwise distinct**. This is, no more and no less, asking for a linear code with minimum distance ≥ 3 (it tolerates 2 erasures). Group the basis indices that already have the same current signature: each new control you build adds an extra (free) bit to the signature of *every* basis element, and within the same group those new bits are the only thing that can distinguish its members.

Hint 5. For a group of size g whose current weight (before adding the k new controls) is w (saturated to 0, 1 or ≥ 2 , because more than 2 no longer contributes anything), the number of usable k -bit labels such that the final signature has weight ≥ 2 is $\text{cap}(0, k) = 2^k - k - 1$, $\text{cap}(1, k) = 2^k - 1$, $\text{cap}(\geq 2, k) = 2^k$. The group fits with k new controls if and only if $g \leq \text{cap}(w, k)$. Try $k = 0, 1, 2, \dots$ and keep the first one that satisfies all groups simultaneously (it is never necessary to go beyond $k = 7$).

Solution

First, a reformulation: a light configuration is reachable if and only if it is the XOR of the masks of some subset of controls, so the set of configurations reachable with a group of controls is exactly the *space* they span over $\mathbb{F}_2^b$. Building new controls and then losing up to two (original

or new) must leave that space intact: we need that, for *any* choice of up to two lost controls, the surviving controls still span the same space of dimension r (the rank of the a_i).

Step 1: build the basis. Perform Gaussian elimination on the a_i , bit by bit from $b - 1$ down to 0: for each control, while it has a bit set at a position where there is already a pivot, it is cancelled with that pivot; if it reaches a position without a pivot, that control itself becomes the new pivot of that position. The controls that are completely cancelled are *dependent*: their mask is the XOR of a subset of the r controls that did stay in the basis, and that subset (represented as a vector of r bits) is their **signature**. It is essential to keep track of which concrete original controls make up each pivot (not just its already reduced mask), because the correctness argument needs each basis element to be a real physical control that can be lost.

Step 2: signature of each basis element. Symmetrically, for each basis element B_i we define its own signature as the set of dependent controls (existing ones, and later the new ones that are built) that use it in their combination. If B_i alone is lost, at least *one other* control referencing it is needed to rebuild it; but that other control could be precisely the second lost control, so the signature of B_i actually needs weight ≥ 2 ; if B_i and B_j are lost together, there must be some control that distinguishes them (one that references exactly one of the two), that is, their signatures must be distinct.

Step 3: group and compute k . Group the basis indices by identical current signature (before adding new controls). Each new control that is built contributes an additional, completely free bit to the signature of each basis element (because any vector of $\mathbb{F}_2^r$ can be expressed as the XOR of a subset of the r basis elements). With k new controls, each group of size g and current weight w (saturated to 0/1/ ≥ 2) needs $g \leq \text{cap}(w, k)$, with

$$\text{cap}(0, k) = 2^k - k - 1, \quad \text{cap}(1, k) = 2^k - 1, \quad \text{cap}(\geq 2, k) = 2^k.$$

The answer is the smallest k that satisfies this inequality in **all** groups at once; it is enough to try $k = 0, 1, 2, \dots$ (it is never necessary to go beyond $k = 7$, because $2^7 - 7 - 1 = 120$ already covers any $r \leq 60$). If the rank of the a_i is 0 (all masks are zero), the only reachable configuration is “all off” and the answer is 0 without further analysis.

With example 1 ($N = 5$, $b = 3$, masks 4, 2, 1, 6, 5): the basis consists of controls 4, 2, 1 (rank 3), and controls 6 and 5 are dependent with signatures $\{4, 2\}$ and $\{4, 1\}$ respectively (in basis coordinates). Grouping the three basis indices by their current signature (who references them) gives three groups of size 1 with weights 1, 1, 2; with $k = 1$ new control it is enough for the two groups of weight 1 to reach weight 2 (they need a non-zero 1-bit label, and there is only one: 1, which suffices because each group has size 1), giving the answer 1 from the statement.

Complexity. Time: $O(N \cdot b)$ Space: $O(N + b)$

The core of the algorithm (building the basis and the signatures of the dependent controls) is short:

```

1  for (Mask control : controls) {
2      Mask value = control;
3      Mask used = 0;
4      for (int bit = bits - 1; bit >= 0; --bit) {
5          if (((value >> bit) & 1ULL) == 0) continue;
6          if (pivot[bit] != 0) {
7              value ^= pivot[bit];
8              used ^= coordinates[bit];
9          } else {
10             pivot[bit] = value;                // reduced mask
11             coordinates[bit] = used ^ (1ULL << rank); // in original
12             ++rank;
13             break;
14         }
15     }
16 }
```

```
15     }
16     if (value == 0) dependent.push_back(used); // signature of the
17         dependent control
    }
```

Problem G — Sergio’s Happiest Office

Estimated difficulty: Medium-hard (approx. 2000) • Topics: Greedy, binary search on the answer, DSU, Fenwick tree

Hints

Hint 1. If you manage to give every office worker at least T units of light, can you always give them at least $T' < T$ with the same budget (or less)? That monotonicity should suggest a very standard technique for attacking “maximize the minimum”.

Hint 2. Good, so the problem reduces to: for a fixed target T , is there an assignment of additional openings x_j (with $0 \leq x_j \leq h_j - a_j$ and $\sum x_j \leq k$) such that every office worker receives at least T ? Think of this as a problem of *covering* each range $[l_i, r_i]$ with a “deficit” $d_i = \max(0, T - \text{current light})$, spending as little capacity as possible.

Hint 3. The order in which you decide to satisfy the office workers matters. Try processing them from left to right by their right endpoint r_i . When it is the turn of one who still has a deficit, which window should be used first: the leftmost or the rightmost one within $[l_i, r_i]$? Think about which windows can be *recycled* for future office workers (who you already know will have $r_j \geq r_i$).

Hint 4. Any light you add at window r_i (the right endpoint of the current range) is automatically available to **any** future office worker that overlaps the current one, no matter where their range starts – because the overlap always includes r_i . In contrast, light added near l_i only serves future office workers whose range also starts far to the left. That is why it pays to always fill the **rightmost possible** free window within the range.

Hint 5. To do this quickly: use a DSU over the windows where each saturated window (with no remaining slack) points to the nearest free window to its left, so you jump in amortized $O(\alpha(m))$ straight to the next usable window. And use a Fenwick tree to know, in $O(\log m)$, how much light has already been added within any range $[l_i, r_i]$. With that, each feasibility check of a candidate T runs in almost linear time, and you can wrap everything in a binary search over T .

Solution

As in any “maximize the minimum” problem, the function

$\text{feasible}(T) =$ is there a valid assignment of $x_1, \dots, x_m$ such that every office worker receives $\geq T$?

is monotone: if it is possible to reach T , it is also possible to reach any $T' < T$ (the same assignment works). This allows a **binary search on the answer**: the upper bound of the search can be set, for example, by taking for each office worker the light they would receive if they alone (ignoring the others) used all the capacity available in their range, bounded by the budget k ; the minimum of those values is a valid bound (nobody can exceed what they could achieve alone).

All that remains is to solve **feasible**(T) efficiently. For a fixed T , each office worker i has a deficit $d_i = \max(0, T - \text{current_light}_i)$, and we need to distribute the budget k among the windows (respecting $x_j \leq h_j - a_j$) to cover all the deficits simultaneously, using the smallest possible total budget (if that minimum exceeds k , T is not feasible).

The key observation is about **order**: if we process the office workers sorted by increasing r_i , any future office worker that overlaps the current one necessarily shares window r_i (and the ones immediately to its left) – never only the windows close to l_i exclusively. Therefore, when light must be added to cover the deficit of the current office worker, it pays to use the **rightmost possible** free window within $[l_i, r_i]$: that light is automatically “recycled” for every future office worker that overlaps, whatever their left endpoint.

The complete algorithm, for a candidate T :

1. Sort the office workers by increasing r_i .
2. Maintain, per window, the remaining capacity $h_j - a_j$ (which gets consumed), and a **DSU** where every saturated window points to the nearest free window to its left (so **find**(r) directly gives the rightmost free window $\leq r$, skipping the saturated ones).
3. Maintain a **Fenwick tree** with the light already added, to compute in $O(\log m)$ the accumulated light of any range.
4. For each office worker, if their deficit is positive: repeatedly take the rightmost free window within their range (via the DSU), consume its remaining capacity, update the Fenwick tree and the budget used, until the deficit is covered. If at any moment there is no free window left within the range, or k is exceeded, T is not feasible.

```

1  auto feasible = [&](int64 target) {
2      FenwickTree added_light(windows);
3      ...
4      for (const Interval& interval : intervals) {
5          int64 current_light = opening_prefix[interval.right] -
6              opening_prefix[interval.left - 1];
7          current_light += added_light.sum(interval.right) - added_light
8              .sum(interval.left - 1);
9          int64 needed = max(0LL, target - current_light);
10         while (needed > 0) {
11             int window = find(interval.right);
12             if (window < interval.left) return false;
13             int64 added = min(needed, remaining_capacity[window]);
14             ...
15         }
16     }
17     return true;
18 };

```

Complexity. Time: $O((n \log n + m) \log(\text{answer}))$ Space: $O(n + m)$

Problem H — Racsó's Toy Train

Estimated difficulty: Medium-hard (approx. 2200-2300) • Topics: Graphs (bridge tree + LCA)

Hints

Hint 1. Forget for a moment that there are many queries. Think about it with a single new edge: if you add an edge between two vertices u and v of a connected graph, which set of vertices ends up 2-edge-connected to each other that was not before? (2-edge-connected means: they remain connected even if any single edge is deleted).

Hint 2. Adding the edge (u, v) creates a unique new cycle, and all the vertices and edges of that cycle no longer have bridges between them. Formally: two vertices x, y become 2-edge-connected if and only if *all* the original bridges of the graph that separated x from y are exactly bridges that were on the path between u and v (that is, if x and y end up “trapped” inside the newly created cycle).

Hint 3. This suggests working with the bridge structure of the original graph, which is the same for all queries (it does not depend on the hypothetical edges). Contract each 2-edge-connected component of the original graph into a single node: what remains is a tree (the *bridge tree*), where each tree edge is a real bridge and each node weighs as many vertices as the component it represents. Adding a new edge (u, v) is equivalent, in this tree, to merging **the whole path** between $\text{comp}(u)$ and $\text{comp}(v)$ into a single component.

Hint 4. Each query (a, b, c, d) then translates into two paths P_1 (between $\text{comp}(a)$ and $\text{comp}(b)$) and P_2 (between $\text{comp}(c)$ and $\text{comp}(d)$) of the bridge tree. If P_1 and P_2 share no node, neither of the two new edges helps the other: the answer is 0. If they share a sub-path I , every pair (x, y) with x in the part of P_1 that is **not** in I and y in the part of P_2 that is not in I either becomes 2-edge-connected only when **both** edges are present (each one closes its own exclusive half; without the other, that half is left unclosed). Count those pairs in terms of the *weights* (sizes, in vertices) of the exclusive parts of P_1 and P_2 .

Hint 5. To compute paths, path weights and the intersection of two paths in a tree, a handful of LCAs per query is enough (at most 3, with the “median” technique: the midpoint of three vertices x, y, z is the deepest among $\text{lca}(x, y)$, $\text{lca}(x, z)$, $\text{lca}(y, z)$). With *binary lifting* precomputed on the bridge tree, each LCA (and therefore each complete query) is obtained in $O(\log n)$.

Solution

First, the bridges of the original graph are computed with a single depth-first search (Tarjan, $O(n+m)$), and each 2-edge-connected component is contracted into a single node. What remains is a tree: the **bridge tree**. Each tree node is assigned a weight equal to the number of original vertices it contains (the size of its component), and each tree edge is exactly one of the original bridges. Each station of the statement maps to a unique node of the bridge tree (that of its component).

The key observation is that adding a new edge between two vertices u and v of the original graph has, on 2-edge-connectivity, exactly the same effect as merging in the bridge tree all the nodes of the path between $\text{comp}(u)$ and $\text{comp}(v)$ into a single component: the new edge, together with that path, forms a cycle, and every cycle is 2-edge-connected.

So each query (a, b, c, d) translates into two paths in the bridge tree: $P_1 = \text{path}(\text{comp}(a), \text{comp}(b))$ and $P_2 = \text{path}(\text{comp}(c), \text{comp}(d))$. Let $W(P)$ be the sum of the weights of the nodes of a path P (that is, the number of original vertices it covers). There are two cases:

- If P_1 and P_2 **do not intersect** as sets of tree nodes, neither of the two new edges depends on the other to close its own cycle, so no new pair requires *both* edges at once: the answer is 0.
- If they intersect, the intersection $I = P_1 \cap P_2$ is itself a contiguous sub-path of the tree (a general property of paths in a tree). The answer is

$$(W(P_1) - W(I)) \cdot (W(P_2) - W(I)).$$

Intuitively: when both edges are added, the whole $P_1 \cup P_2$ ends up in a single giant 2-edge-connected component. A pair (x, y) becomes “newly” 2-edge-connected (that is, it was not already so before adding any edge) only if at least one of the two was not in the common part I . Moreover, if x is in the exclusive part of P_1 , it **needs** the edge of P_1 to connect (the edge of P_2 alone cannot close its cycle), and symmetrically for y in the exclusive part of P_2 . That is why the actual count, that of pairs requiring BOTH edges, is the product of the sizes of the two exclusive parts.

All that remains is to compute paths and path intersections quickly. With *binary lifting* on the bridge tree we get the LCA of two nodes in $O(\log n)$, and with that: the distance between two nodes, the weight of a path (using prefix weights accumulated from the root, analogous to prefix sums over the path to the root), and the *median* of three nodes x, y, z (the deepest of the three pairwise LCAs). The endpoints of the intersection of $P_1 = (s_1, e_1)$ and $P_2 = (s_2, e_2)$ are $\text{median}(s_1, e_1, s_2)$ and $\text{median}(s_1, e_1, e_2)$ (one must check that this segment is really contained in both paths; if it is not, the intersection is empty and the answer is 0).

```

1  int64 answer(int first_start, int first_end, int second_start, int
    second_end) const {
2      int first_intersection = median(first_start, first_end,
        second_start);
3      int last_intersection = median(first_start, first_end, second_end)
        ;
4
5      if (distance(second_start, first_intersection)
6          + distance(first_intersection, second_end) != distance(
            second_start, second_end)) {
7          return 0;
8      }
9
10     int64 common_weight = path_weight(first_intersection,
        last_intersection);
11     int64 first_exclusive = path_weight(first_start, first_end) -
        common_weight;
12     int64 second_exclusive = path_weight(second_start, second_end) -
        common_weight;
13     return first_exclusive * second_exclusive;
14 }

```

The bridges and the 2-edge-connected components are computed only once at the beginning ($O(n+m)$); afterwards, each of the q queries is answered in $O(\log n)$ with the bridge tree already built and the *binary lifting* already precomputed.

Complexity. Time: $O(n+m+(n+q)\log n)$ Space: $O(n+m)$

Problem I — Backup Ink

Estimated difficulty: Very hard (approx. 2650) • Topics: Strings: suffix automaton, small-to-large merging

Hints

Hint 1. Fix a length L and look only at the occurrences of a pattern of that length, sorted by position. If you cancel an interior occurrence, which other occurrences could be covering the same cells? Convince yourself that it is enough to look at the immediately previous and the immediately next occurrence: no farther occurrence can cover a cell that its neighbors do not already cover.

Hint 2. With that in mind, for an interior occurrence at distance a from its left neighbor and b from its right neighbor, compute explicitly how many cells of its own interval (of length L) are covered by each neighbor, and subtract from the total. At the ends there is only one neighbor on each side, so the analysis is even simpler.

Hint 3. You should arrive at the fact that the damage of cancelling an interior occurrence is $\max(0, \min(L, a) + \min(L, b) - L)$, and at the ends it is $\min(L, \text{distance to the only neighbor})$ (with damage L if the occurrence is unique). Note that if $a \leq k$ or $b \leq k$, the damage of that interior occurrence is already $\leq k$ automatically, regardless of the other side. Only the pairs of consecutive distances where **both** exceed k are dangerous.

Hint 4. For a dangerous pair (a, b) with $a, b > k$, find for which values of L the damage of that interior occurrence is $\leq k$. You will get a condition of the form $L \geq a + b - k$. Combining this with the condition at the ends (both border distances must be $\leq k$), each length $L > k$ of a fixed pattern is resilient if and only if it exceeds a certain threshold that depends only on the *set of positions* where that pattern occurs.

Hint 5. Many distinct substrings share exactly the same set of occurrence positions (the same **endpos**): that is exactly what a suffix automaton groups, one state per equivalence class. Process the states from largest to smallest **len**, keep per state a **std::set** of positions and a **std::multiset** with the sums $a + b$ of the dangerous pairs, and when you finish with a state merge its set into that of its suffix link using *small-to-large* (always inserting the smaller set into the larger one).

Solution

Let us fix a length L and call $p_1 < p_2 < \dots < p_m$ the (starting) positions of the occurrences of a pattern of that length. Each occurrence i paints the interval $[p_i, p_i + L - 1]$. Since two occurrences of the *same* string cannot “jump over” a third intermediate occurrence without the latter already covering the whole overlap zone, the only relevant overlap for occurrence i is with its immediate neighbors $i - 1$ and $i + 1$.

With $a = p_i - p_{i-1}$ and $b = p_{i+1} - p_i$, the left neighbor covers the first $\max(0, L - a)$ cells of the interval of i , and the right one the last $\max(0, L - b)$. The damage of cancelling i is what is

left uncovered by both, which after simplifying gives exactly

$$\text{damage}(i) = \max(0, \min(L, a) + \min(L, b) - L).$$

At the ends ($i = 1$ or $i = m$) there is only one neighbor, and the damage is $\min(L, \text{distance})$ (or L if $m = 1$, the occurrence is unique).

A pattern is resilient if and only if the damage of *all* its occurrences is $\leq k$. If $L \leq k$, the damage of any occurrence (at most L) already satisfies the bound, so **every length $\leq k$ is automatically resilient**. For $L > k$ at least two occurrences are needed, and the two extreme distances must be $\leq k$ (otherwise, that border occurrence alone already breaks the bound). For each pair of consecutive distances (a, b) with **both** greater than k , a little algebra shows that the damage of that interior occurrence drops to k or below exactly when $L \geq a + b - k$ (for any other pair, with $a \leq k$ or $b \leq k$, that occurrence is never a problem). So, for a fixed set of occurrences of a pattern, there is a threshold T (the maximum of $a + b - k$ over the dangerous pairs, or $k + 1$ if there are none) such that the pattern of length L is resilient exactly when $L \leq k$ or $L \geq T$ (plus the border conditions, which do not depend on L).

The key point is that, for *all* the substrings that share the same set of occurrence positions, this threshold T is the same – because the distances between consecutive positions do not depend on L , only on where they occur. And that grouping (same set of positions = same **endpos**) is exactly what a **suffix automaton** does: each state v represents all the substrings with lengths in $(\text{len}(\text{link}(v)), \text{len}(v)]$, all with the same **endpos**.

The states are processed in decreasing order of len (which is a topological order of the suffix-link tree, from the leaves towards the root). Each “leaf” state starts with its own position; when a state has been counted, its set of positions is merged into that of its suffix link, so when it is a state’s turn, its set is already complete (it includes everything contributed by its descendants). To avoid paying $O(n)$ per merge, *small-to-large* is used: the smaller set is always inserted into the larger one, which gives $O(n \log n)$ insertions in total. Together with a `std::multiset` that incrementally maintains the sums $a + b$ of the dangerous pairs (each insertion of a new position changes at most 3 consecutive pairs), the threshold of a state is computed in $O(\log n)$:

```

1  int threshold(int n) const {
2      if (positions.size() < 2) return n + 1;
3      if (*next(positions.begin()) - *positions.begin() > k ||
4          *positions.rbegin() - *next(positions.rbegin()) > k) return n
5          + 1;
6      return bad.empty() ? k + 1 : max(k + 1, *bad.rbegin() - k);
7  }
```

With the threshold T of a state v (range of lengths $[low, high]$), its contribution to the answer is $\max(0, \min(high, k) - low + 1) + \max(0, high - \max(low, T) + 1)$: the automatically resilient lengths ($\leq k$) plus those that exceed the threshold. Adding this over all states gives the answer. The cases $k = 0$ (answer 0, because the leftmost occurrence of any pattern always has at least one cell covered only by it) and $k = n$ (every substring is resilient, the answer is the total number of distinct substrings) are handled separately, without building the position sets.

Complexity. Time: $O(n \log^2 n)$ Space: $O(n)$

Problem J — The Single-Cut Cross

Estimated difficulty: Hard (approx. 2250) • Topics: Computational geometry (exact angular sweep)

Hints

Hint 1. For a fixed direction d , think about how to know whether the infinite line through the origin with direction d cuts the interior of a cable AB . No trigonometry is needed: it is enough to compare on which side of that line A and B fall, and that is exactly the sign of a cross product.

Hint 2. The second laser is perpendicular to the first. If you already know how to test “does the line with direction d cut me?” with a cross product, what similar operation lets you test the perpendicular line *without* explicitly rotating the vector d ? (Think about it in terms of a dot product instead of a cross product.)

Hint 3. There is no need to try all possible orientations (there are infinitely many). The set of cables cut by each line can only change when some line passes exactly through an endpoint of some cable, and those orientations are already explicitly forbidden by the statement. How many “critical” directions are there in total, and what happens to the cut configuration in the rest of the orientations?

Hint 4. Rotating the whole cross by 90° leaves it exactly the same as a *pair of lines* (the two simply swap which one is “the first” and which one is “the second”). That means it is enough to sweep a quarter turn to see all the possible configurations.

Hint 5. Fold each endpoint of each cable, rotating it by 90° at a time, until it falls in the first quadrant; the parity of rotations tells you whether that endpoint is an event for the first or the second line. Sort all those directions with *exact* cross products (no angles and no **double**), group the events that fall in exactly the same direction, and maintain, while sweeping, the weight cut by exactly one line and the number of cables cut by both. You only evaluate the answer in the open gaps between groups of events.

Solution

Let us fix a direction $d = (d_x, d_y)$ and the infinite line L_d through the origin with that direction. For a point P , the sign of $\text{cross}(d, P) = d_x P_y - d_y P_x$ indicates on which side of L_d the point P falls (zero exactly on the line). A cable with endpoints A, B has its interior cut by L_d if and only if $\text{cross}(d, A)$ and $\text{cross}(d, B)$ have strictly opposite signs.

The second line is $L_{d^\perp}$ with $d^\perp = (-d_y, d_x)$. Since $\text{cross}(d^\perp, P) = -\text{dot}(d, P)$, testing $L_{d^\perp}$ is exactly comparing the signs of $\text{dot}(d, A)$ and $\text{dot}(d, B)$: there is never any need to actually rotate d . Each cable then has a 2-bit *state*: bit 1 if L_d cuts it, bit 2 if $L_{d^\perp}$ cuts it. State 3 (both bits) invalidates the whole orientation; state 1 or 2 contributes the weight of the cable; state 0 contributes nothing.

90° symmetry. As a pair of lines (without orientation), $\{\theta, \theta+90^\circ\}$ and $\{\theta+90^\circ, \theta+180^\circ \equiv \theta\}$ are literally the same pair: rotating the cross by 90° leaves it unchanged, only swapping which line is “the first”. Therefore it is enough to sweep $\theta \in [0^\circ, 90^\circ)$.

Critical directions. For an endpoint P , the sign of $\text{cross}(d(\theta), P)$ is a continuous function of θ that only vanishes when $\theta \equiv \arg(P) \pmod{180^\circ}$; outside those points it cannot change sign. The same holds for dot with $\theta \equiv \arg(P) + 90^\circ \pmod{180^\circ}$. Reduced modulo 90° , both values fall into the same residue $\arg(P) \pmod{90^\circ}$: each endpoint contributes a single event in the quarter turn, labeled with a bit (1 or 2) according to the parity of the 90° rotations needed to fold it into the first quadrant. Between two consecutive events the state of all cables is constant, so it is enough to evaluate one candidate per gap between events.

The algorithm is then:

1. Fold each of the $2n$ endpoints into the first quadrant, generating an event (**direction**, **cable**, **bit**).
2. Sort the events by increasing direction with an *exact* comparator: for vectors in the first quadrant, the sign of $\text{cross}(u, v)$ already determines the order unambiguously (the angular difference never exceeds 90°).
3. Sweep the events grouping them into blocks of *exactly* equal direction (not event by event): within a block there is no real “intermediate” orientation, so the whole block must be applied before evaluating the answer.
4. Maintain the weight cut by exactly one line and the number of cables in state 3; after each block, if that number is 0, the cell is valid and the maximum is updated.

```

1  i128 cross(Point a, Point b) { return (i128)a.x*b.y - (i128)a.y*b.x; }
2  i128 dot(Point a, Point b)   { return (i128)a.x*b.x + (i128)a.y*b.y; }
3  bool opposite(i128 a, i128 b) { return (a < 0) != (b < 0); }
4  // ...
5  while (r < events.size() && cross(events[l].direction, events[r].
6      direction) == 0) {
7      int i = events[r].cable;
8      if (state[i] == 3) --invalid; else if (state[i]) score -= weight[i];
9      state[i] ^= events[r].bit;
10     if (state[i] == 3) ++invalid; else if (state[i]) score += weight[i];
11     ++r;
12 }
13 if (!invalid) answer = max(answer, score);

```

All the arithmetic (**cross**, **dot**, ordering, ties) is done in `__int128_t`, never in `double`: products of coordinates up to 10^9 reach magnitude 10^{18} , which no longer fits in the 53-bit mantissa of a `double`, and this problem is designed precisely with cases of almost (or exactly) coincident directions, where that rounding error can flip a sign. With 128-bit integers the computed sign is always the true one.

Complexity. Time: $O(n \log n)$ per test case (dominated by the **sort** of events) Space: $O(n)$

Problem K — Signals That Don't Wait

Estimated difficulty: Very hard (approx. 2500) • Topics: Dynamic programming over marginal differences (slope trick)

Hints

Hint 1. If the capacity c_i were always infinite (that is, if the battery never lost energy to overflow), this problem would be exactly a classic one: each day some energy is received and you must decide whether to “buy” that day’s transmission by paying a_i , maximizing the number of purchases. Think about that simplified version first.

Hint 2. For that simplified version, look at what happens if you define $dp[k]$ = the maximum possible energy at the end of day i having sent *exactly* k messages so far (among all ways of choosing which k to send). Since sending more messages always costs extra energy, $dp[k]$ is non-increasing in k . That is the function that must be updated day by day.

Hint 3. The function dp is not arbitrary: it is **concave**. That is, the “marginal loss” of sending the k -th message (the difference $dp[k-1] - dp[k]$) never decreases as k grows – if you choose optimally, it always pays to have bought the cheapest transmissions first. This means that dp is completely determined by the *sorted multiset* of those differences, without needing to store $dp[k]$ for each k separately.

Hint 4. With that representation, processing a day means “insert a_i as a new candidate difference, and if the total cost already bought exceeds the available base energy, discard the *most expensive* difference” (the least convenient transmission of all those bought so far). This is, literally, the same trick used by CF 1526C2 (<https://codeforces.com/problemset/problem/1526/C2>).

Hint 5. The capacity c_i is still missing. Each day, before deciding whether to send, the energy is truncated to $\min(c_i, E + r_i)$. That truncation does not affect all values of $dp[k]$ equally: it affects small k more (they represent more retained energy). It can be simulated exactly by discounting the “overflow” ($\max(0, dp[0] + r_i - c_i)$) starting from the **smallest** differences of the multiset (not the most expensive ones), until that overflow is exhausted.

Solution

Let $dp_i[k]$ be the maximum possible energy at the end of day i having sent exactly k messages among the first i days (over all ways of choosing which ones to send), and $-\infty$ if it is not reachable. As the hints argue, for a fixed i this function is concave and non-increasing in k , so instead of storing $dp_i[0], \dots, dp_i[i]$ explicitly it is enough to store:

- **base**: the value $dp_i[0]$ (the maximum possible energy without having sent anything), and
- a **multiset**<long long> with the differences $dp_i[k-1] - dp_i[k] > 0$ already “bought” (plus a separate counter of how many differences are exactly 0, so as not to re-scan them), such that $dp_i[k] = \text{base} - (\text{sum of the } k \text{ smallest differences})$.

Each day is processed in two steps, in this order:

1. Recharge and capacity truncation. First `base` becomes $\min(c_i, \text{base} + r_i)$ (the recharge raises all values equally, and the capacity truncation only affects the k for which $dp[k]$ exceeded c_i , which are precisely the small k). To simulate that truncation on the differences, we compute `overflow = max(0, baseprevious +  $r_i$  -  $c_i$ )` and discount that amount starting from the **smallest** differences of the multiset (taking out the minimum, subtracting as much as possible, reinserting it if something is left, and otherwise counting it as a zero difference), until `overflow` is exhausted.

2. Transmission attempt. a_i is inserted as a new difference into the multiset (today's message is tentatively "bought") and added to the accumulated total of differences. If that total exceeds `base`, it means the energy is no longer enough to sustain all the chosen transmissions, and the **largest** difference is discarded (the least profitable transmission of all those bought so far, which may be today's or one from an earlier day).

The final answer is the number of differences that survive at the end (including the zero ones), that is, the final size of the multiset plus the zero counter.

A fragment of the core of the algorithm (multiset version):

```

1  i64 overflow = max(0LL, base + recharge - capacity);
2  base = min(capacity, base + recharge);
3  while (overflow && !positive.empty()) {
4      auto it = positive.begin();
5      i64 value = *it;
6      positive.erase(it);
7      i64 removed = min(overflow, value);
8      total -= removed;
9      overflow -= removed;
10     value -= removed;
11     if (value) positive.insert(value);
12     else ++zeros;
13 }
14 positive.insert(cost);
15 total += cost;
16 if (total > base) {
17     auto it = prev(positive.end());
18     total -= *it;
19     positive.erase(it);
20 }

```

Complexity. Time: $O(n \log n)$ per test case (amortized) Space: $O(n)$

Each message enters and leaves the multiset $O(1)$ times amortized, and each insertion/deletion costs $O(\log n)$.