Detecting Molecules

Author: **Shi-Chun Tsai**
National Chiao-Tnug University, `sctsai@cs.nctu.edu.tw`, country: Taiwan

Subtask 1 is special case, iterate all possible $k \leq n$, try to take exactly k molecules.

Subtask 2 is special case, iterate all possible k_1, k_2: $k_1 + k_2 \leq n$, try to take exactly k_1 molecules of minimal weight and exactly k_2 molecules of maximal weight.

Subtasks 3 and 4 may be solved using dynamic programming. The task is classic knapsack problem. Use $O(n \cdot u)$ of time, and $O(u)$ of space.

You may optimize the dynamic programming, use `bitset` and you'll pass **Subtasks 5**.

We suggested, contestant who solves **Subtasks 5 and 6** will invent greedy approach.
If you implement greedy in $O(n^2)$ you'll pass only **Subtask 5**.
Good time to pass **Subtask 6** is $O(n \log n)$.

There are 3 correct greedy solutions. All of them start with sorting the array of weights in nondecreasing order.

Greedy 1. Let fix k, number of molecules to take. We can choose set of size k with sum in $[l..r]$ iff $minSum[k] \leq u$ and $maxSum[k] \leq l$. Where $minSum[]$ is partial minimums on prefixes and $maxSum[]$ is partial maximums on suffixes. Both of them may be precalculated in $O(n)$.

Proof. Lets take minimal possible k molecules, its summary weight does not exceed l. Lets change the set smoothly from "k minimal molecules" to "k maximal molecules". One step: drop any one molecule, take any one another. Each step changes the sum by at most $u - l$. The last value of sum is at least l. So one of intermediate steps gives $l \leq sum \leq u$. ■

Greedy 2. There exists an answer which forms segment.
Use two pointers to find it in $O(n)$.

Proof. Lets fix k – number of molecules in the answer. The smallest k molecules form the leftest segment, the biggest k form the rightest segment. Lets change the set smoothly from "the leftest segment" to "the rightest segment". One step: drop the leftest molecule, add new one at the right. ■

Greedy 3. There exists an answer which forms union of prefix and suffix.
Use two pointers to find it in $O(n)$.

Proof. Lets fix k – number of molecules in the answer. The smallest k molecules form prefix, the biggest k form suffix. Lets change the set smoothly from "prefix" to "suffix".

One step: make prefix shorter by one, make suffix longer by k. ■

Roller Coaster Railroad

Author: **Kento Nikaido**
Keio University, snukent@gmail.com, Japan

Subtask 1. To solve the first subtask, one could iterate over all $n!$ possible permutations of the special sections. Once the permutation is fixed, the only thing left is to compute the required railway segment length between every two consecutive sections: if the i^{th} section is followed by the j^{th} one, then the required length is $\max(0, t[i] - s[j])$.

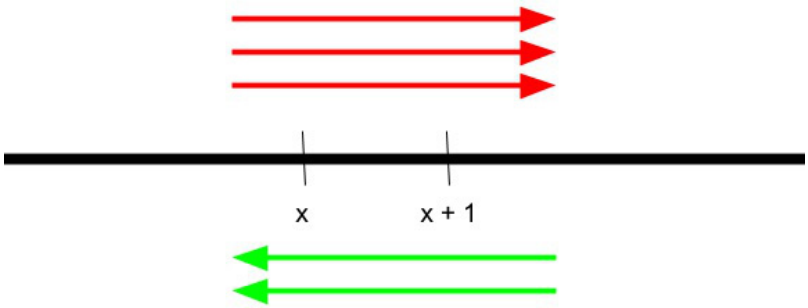
Subtask 2. One could use a standard dynamic programming approach to solve the second subtask. Every subset of the given sections set can be encoded as a bit mask (from 0 to $2^n - 1$). Let $ans[mask][i]$ denote the minimum total railway length, considering only the sections from the set encoded by $mask$ with an additional restriction that the i^{th} special section should go first.

1. If $mask = 2^i$ (there is only one special section), the answer is clearly zero.
2. Otherwise, there must be a section j following the i^{th} one, so

$$ans[mask][i] = \min_{j \neq i,\ j \in mask} \left(ans[mask - 2^i] + \max(0, t[i] - s[j]) \right)$$

The answer to the problem is $\min_i (ans[2^n - 1][i])$. One could also notice that this subtask is an instance of the well-known TSP (travelling salesman problem), where the special sections represent the cities, and the distance function is the required railway segment length.

Subtask 3. Let's add an additional special section with $s = \infty$ and $t = 1$: now we can introduce a restriction that the train must have speed exactly 1 km/h at the end of the journey (here ∞ represents any number greater or equal than any of the numbers given in the input data — 10^9 would suffice).



Consider an infinite graph, with a vertex set $1, 2, 3, ...$ and there is an edge from $s[i]$ to $t[i]$ for every section i (including the added one). For every positive integer x consider the *balance* value: (number of edges going over the segment [x,x+1] from left to right) minus (number of edges going over the segment [x,x+1] from right to left). In the picture above one can see three edges going from left to right (red) and two going in the opposite direction (green), so the balance is 1.

If one aims to start from 1 km/h and end with the same speed, then the train must cross the segment $[x, x + 1]$ equal number of times in both directions. If the balance is positive then it's necessary to add an additional green edge to slow down the train, so at least one railway segment is required and the answer is not zero. If the balance is negative it just means that the train needs to be accelerated at some point, so one can add as many additional red edges as needed for free.

Once the balance equals zero for every x, it is sufficient to check whether the resulting graph is connected or not. If the graph is not connected, than the answer is clearly not zero: to go from one component to the other it's needed to slow down the train at least once, so an additional railway segment is required. If the graph is connected, then, since all the balances are zero, for every vertex x its in-degree equals its out-degree, and thus there exists an Euler cycle in this graph, from which one could construct a valid sections arrangement.

To do this efficiently, one need to consider only the "interesting" values of x, which are given in the input data, and instead of considering segments $[x, x + 1]$ one should consider $[interesting_i, interesting_{i+1}]$.

Subtask 4. The solution for the last subtask naturally emerges from the previous one. If the balance is positive for some $x = interesting_i$, it is *required* to add additional green edges until the balance is restored, and every green edge corresponds to a railway segment. Thus, for every $x = interesting_i$ one needs to add $\max(0, balance \cdot length)$ to the answer, where $length$ stands for the distance to the next interesting point $(interesting_{i+1} - interesting_i)$.

The last piece is to make the graph connected. In case the balance is zero we can connect $interesting_i$ and $interesting_{i+1}$ with two edges in both directions, paying the length of this segment. Now we need to solve an instance of the well-know MST (minimum spanning tree) problem.

Shortcut

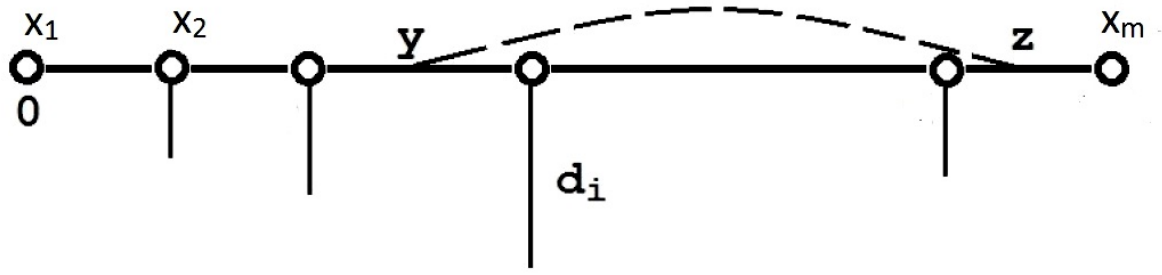
Author: **Gleb Evstropov**
National Research University High School of Economy, glebshp@yandex.ru, country: Russia

Subtasks 1 and 2: try to connect every pair of stations on the main line and then find the *diameter* by checking every pair of stations. In Subtask 2 you have to precompute the stations coordinates along the main line $x_i = \sum_{j=0}^{i-1} l_j$. These values will be used in all solutions for the other subtasks.

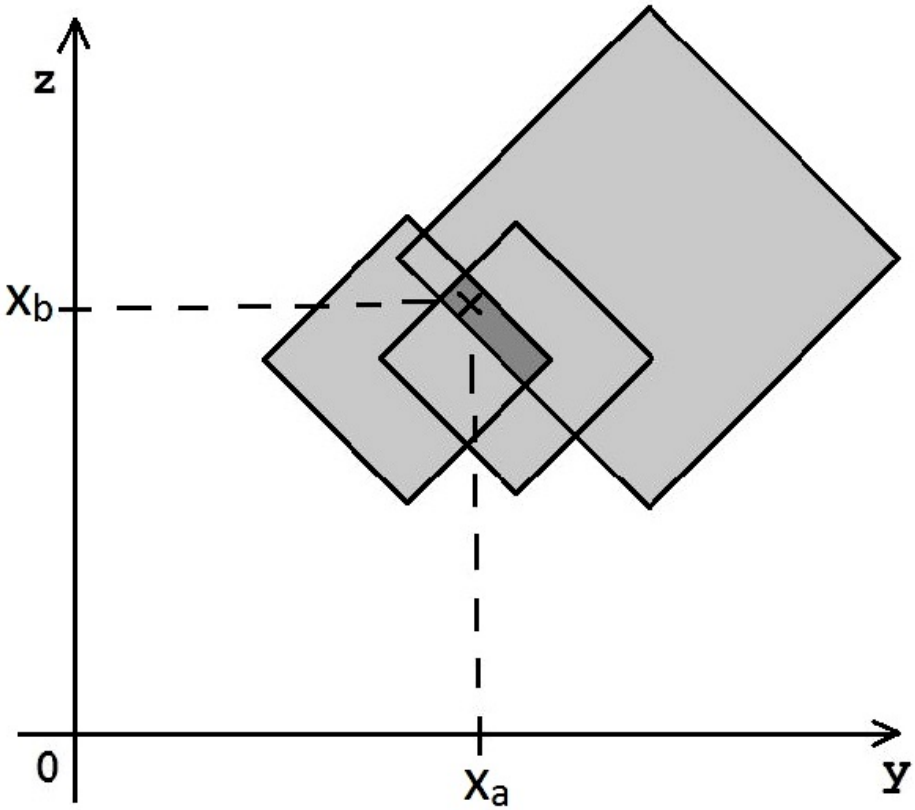
Subtasks 3 and 4: we can try to connect every pair of stations on the main line and then find the resulting diameter. To find the diameter, we can write down the resulting cycle and then iterate through it, keeping a pointer to the opposite position on the cycle (opposite in terms of distance). We also can maintain two queues for each of the halves, keeping the furthest stations in terms of cycle distance plus secondary line length. This will help us to compute the diameter in $O(n)$ time, making the solution run in $O(n^3)$ time in total. In subtask 3, we can use some data structures instead of queues, making the solution run in $O(n^3 \log(n))$. However, both solutions are technically complex.

Subtask 5: This subtask requires some clever ideas. First, we can do binary search on the answer. Then, we have to somehow check if it's possible to make the diameter less or equal to some fixed value k. Let's write some inequations.

Let's allow the express line start and end not only on the stations, but also on arbitrary points y and z on the main line. We use the same coordinate system that was used to compute x_i.



If we take a look at some pair i and j we can see that if $d_i + d_j + |x_i - x_j| \leqslant k$ then all pairs of y and z work. Otherwise, only y and z such that $d_i + d_j + |x_i - y| + |x_j - z| \leqslant k$ are valid. This formula actually describes a square rotated 45°. What we need to do is to try all pairs of i and j, cross corresponding squares and then check if there is some point where $y = x_a$ and $z = x_b$ inside the resulting area.



So far we can just check for such point by trying all possible pairs of a and b. However, there is a nice way to perform this check in linear time and not care about its impact to overall complexity in future. Proceed with a sweep line and keep two pointers: on the lowest point above the square and highest point below the square. Since functions of both positions from the x-coordinate of the sweep line are unimodal the overall complexity will be $O(n)$.

The total complexity of such solution is $O(n^2 \log M)$ where M is the maximal possible answer.

Subtask 6: Now, two more observations are needed. Let's fix $j > i$ and $y < z$. Now the pair produces some square iff $x_j - x_i + d_j + d_i > k$. Let's fix j and find the intersection of the squares for all possible i. The bounds of the square produced can be rewritten as follows:

$$\begin{cases} z + y \leqslant k + (x_j - d_j) + (x_i - d_i); \\ z + y \geqslant (x_j + d_j) + (x_i + d_i) - k; \\ z - y \leqslant k + (x_j - d_j) - (x_i + d_i); \\ z - y \geqslant (x_j + d_j) - (x_i - d_i) - k. \end{cases}$$

We can see that the bounds only depend on $x_i - d_i$ and $x_i + d_i$. We need only maximal and minimal values of $x_i - d_i$ and $x_i + d_i$, so we can use some data structure, for example, segment tree, to find them in $O(n \log(n))$ time. The total complexity of this solution is $O(n \log(n) \log(M))$.

Subtasks 7 and 8: Notice that $x_j - x_i + d_j + d_i > k$ is equal to $(x_j + d_j) - (x_i - d_i) > k$. It follows that if we iterate through j in the order of increasing $(x_j + d_j)$ then the set of i that is used to take minimum and maximum values is only expanding, i.e. once some particular i is in the set, it remains there for all remaining values of j. Thus, we can keep current maximum and minimum values of $x_i - d_i$ and $x_i + d_i$ without any data structure with the use of the two pointers technique and two sorted arrays. The total complexity of the full-score solution is $O(n \log M)$. Note, that sorting is done at the very beginning and there is no need to resort these two arrays in each iteration of binary search.

The another approach to achieve this time bound that doesn't use sort (and thus solves the decision problem itself in linear time) is to get rid of useless elements. We say that station i dominates station j if $d_i > d_j + |x_i - x_j|$. Now, if one station is dominated by the other we can consider only dominator as the endpoint of the express line (unless the position being dominated is considered for the other end). The set of positions that are not dominated by any other index can be found in linear time by computing prefix and suffix maximums. Now when we use the same solution as for subtask 6, but we query maximum and minimum values for inequality only when considering "good" positions (not dominated by any other positions). This allows to achieve the situation that the query bound moves only right and we can process each of them in $O(1)$.

Paint By Numbers

Author: **Michal Forišek**
Comenius University Bratislava, misof@ksp.sk, country: Slovakia

Subtask 1 is as simple as it gets: just generate and intersect all possible locations of the clue.

Subtask 2: if you really have to, you can bruteforce the 2^{20} configurations and check whether they match the clues. Your loss (of time), directly solving subtask C is much simpler.

Subtask 3 has a special case, helpfully shown in Example 2.

Except for that special case, we can only deduce black cells. This can be done greedily: for each clue, find the leftmost and the rightmost valid position of the corresponding block. If their intersection is non-empty, those cells are guaranteed to be black. (Proof of a more general statement is given below.)

The same approach works for subtask 4 as well. Additionally, we are able to deduce some white cells. One case is shown in Example 3. The same logic has to be applied at the beginning and at the end of a row. (In fact, the easiest solution is to prepend and append a white cell as a sentinel.)

Here's a tricky test case for subtask D: $n = 13$, $k = 4$, $c = (3, 1, 1, 3)$, $s =$ "`....._...._....`". Correct output: `?XX?_X_X_?XX?`. The tricky part here is that we can deduce the white cell at index 6. (Whenever two consecutive blocks have a fixed position, all cells between those positions have to be white.)

Subtask 4 was as far as we could get with an easy greedy approach. In order to solve the general version we will use dynamic programming.

One possible solution looks as follows:

- Step 1: Compute prefix sums of given white cells and given black cells (each separately).
- Step 2: For each i and j, compute the answer $P(i, j)$ to the following question: "Is there a valid solution if we only consider the first i clues and the first j cells of the given puzzle (as if cutting away and discarding the rest)?"
- Base case of the recurrence: If $i = 0$, this is possible iff there is no given black cell among the first j cells.

- Recursive case: If cell $j-1$ is forced white, $P(i,j) = P(i,j-1)$. If cell $j-1$ is forced black, we verify that it is possible to place the last block there (the number of forced white cells it overlaps must be zero, the next cell to the left must be able to be white) and if that is the case, we make a recursive call $P(i-1, j-c-1)$ where c was the corresponding clue. Finally, if cell $j-1$ isn't forced, the answer is the logical or of both above cases.

- Step 3: The same in reverse, i.e., with suffixes of the puzzle and the list of clues.

- Step 4: For each cell, we verify whether it can be white. A cell cannot be white if it is forced to be black. If that is not the case, we try all possibilities for the number of black blocks to the left of the cell, and verify each possibility using the data we precomputed in steps 2+3.

- Step 5: For each clue, we mark all the cells where it can be located as cells that can be black. Suppose we are processing clue t and that its value is c_t. For each i we verify whether the clued block can be placed starting from cell i. This requires a few checks that can all be done in constant time:
  – cells $i-1$ and $i+c_t$ must not be forced black
  – cells i through $i+c_t-1$ must not be forced white
  – there must be a valid solution for the first t clues and the first $i-1$ cells
  – there must be a valid solution for the last $k-t-1$ clues and the last $n-i-c_t-1$ cells

The above solution can conveniently be implemented in $O(nk)$, where k is the number of clues. Less efficient implementations may run in $O(n^2)$, these should not solve subtask 7. Other implementations may run in something like $O(n^2|C|)$ or in $O(n|C|^2)$. These should solve subtask 5, but not subtasks 6 and 7.

Proof of the greedy solution (subtasks 1-4)

Theorem 1. Consider the version of our puzzle where initially each cell is either unknown or forced white.

Suppose that there is a cell x such that there are two valid solutions in which this cell belongs to different black blocks. Then there is also a valid solution in which this cell is white.

Proof. Suppose that we have $i < j$ such that in the first solution the block that covers cell x corresponds to clue i and in the second solution it corresponds to clue j.

We will now construct a new solution as follows: take the first $j-1$ black blocks from the second solution and the remaining blocks from the first solution.

This is a valid solution because in the second solution the first $j-1$ black blocks are all strictly to the left of cell x and in the first solution all the remaining blocks are strictly to the right of cell x. It is also obvious that for this reason cell x is white in the new solution. ■

Unscrambling Messy Bug

Author: **Shi-Chun Tsai**
National Chiao-Tnug University, `sctsai@cs.nctu.edu.tw`, country: Taiwan

Subtask 1 allows you to check all 2^n possible elements, so it can be solved by various solutions. For example, you can add 2^{n-1} random elements in your set, check all elements, try all $\frac{n(n-1)}{2}$ transpositions and check that it will give you the same result.

Subtask 2 may be solved by simple solution, using at most n operations `add_element` and at most $\frac{n(n-1)}{2}$ operations `check_element` operations.

- Lets add n elements into the set, i-th elements will have first i bits set to one.

```
add_element("10000000")
add_element("11000000")
add_element("11100000")
add_element("11110000")
add_element("11111000")
add_element("11111100")
add_element("11111110")
add_element("11111111")
```

- Now we will get positions of bits one by one. First, lets find the position of bit 0. This can be done using at most $n-1$ queries:

`check_element("10000000")` returned `false`
`check_element("01000000")` returned `false`
`check_element("00100000")` returned `false`
`check_element("00010000")` returned `false`
`check_element("00001000")` returned `false`
`check_element("00000100")` returned `true`

- Now we know the position of bit 0 and want to find the position of bit 1. This can be done using at most $n-2$ queries:

`check_element("10000100")` returned `false`
`check_element("01000100")` returned `false`
`check_element("00100100")` returned `false`
`check_element("00010100")` returned `true`

- And so on, we can find position of i-th bit using $n-i-1$ queries, so the total number of queries in the worst case is $\frac{n(n-1)}{2} = 496$.

Subtask 3 can be solved by several optimizations of the previous algorithm. The simplest one is to shuffle the order of bytes. This will give us the average number of queries $\frac{n(n-1)}{4} = 248$, which was enough to pass the tests.

Subtasks 4 and 5 require $O(n \log n)$ solution, in **subtasks 4** you can make at most $2n \log n$ operations of each type, in **subtasks 5** you can make only $n \log n$ operations.

Subtask 5 may be solved using *divide and conquer* technique. Lets try to split all bits into two halves using n requests, and solve problem for each half independently. In this solution we will make at most $n \log_2 n$ operations of each type.

- To split group of bits into two halves, we will add into set $n/2$ elements, i-th element will have i-th bit set to one, all other set to zero:

  ```
  add_element("10000000")
  add_element("01000000")
  add_element("00100000")
  add_element("00010000")
  ```

- After this, we will check n elements with single bit set to one. For example:

  `check_element("10000000")` returned `false`
  `check_element("01000000")` returned `true`
  `check_element("00100000")` returned `false`
  `check_element("00010000")` returned `false`
  `check_element("00001000")` returned `true`
  `check_element("00000100")` returned `true`
  `check_element("00000010")` returned `false`
  `check_element("00000001")` returned `true`

- Now we know, which $n/2$ bits correspond to first $n/2$ bits. In this example we know, that bits 1, 4, 5, and 7 correspond to bits 0–3. So now we will solve same problem for them only, and after that solve problem for other $n/2$ bits.
- In order to solve problem for some subset of bits, we need to make sure that the elements we use in different subproblems are distinct. We can achieve this by setting all bits that are not in our subset to ones. For example, when we want to split bits 0–3 into halves, we will make the following operations.

  ```
  add_element("10001111")
  add_element("01001111")

  check_element("11110010")
  check_element("10111010")
  ```

```
check_element("10110110")
check_element("10110011")
```

Aliens

Author: **Chethiya Abeysinghe**
Forestpin (Pvt) Ltd, chethiya@gmail.com, country: Sri Lanka

Subtask 1. $k = n$ means that you can use a photo for each point of interest: the minimum photo containing point (r_i, c_i) is the square containing points (r_i, r_i) and (c_i, c_i). Constraints in this subtask were small enough to iterate over all cells in that square and mark them as photographed. After processing all photos, calculate and return the number of cells which have been marked at least once.

Subtask 2 required the participants to come up with a dynamic programming solution. Some important observations:

- If a photo covers two points (x, x) and (y, y), then it also covers all points between them.
- Each photo's boundary must be equal to some r_i.

Now we can treat this problem as a dynamic programming problem: cover n points on line using k segments such that sum of squares of their lengths is as small as possible. Start with preprocessing the input data: sort all points by r_i and remove duplicates. Notice that each photo should cover some contiguous set of points.

- Let $f_{i,j}$ be the minimum cost to cover first i points with at most j photos.
- $f_{0,j} = 0$ for all $0 \le j \le k$.
- $f_{i,j} = \min_{t<i} f_{t,j-1} + (r_{i-1} - l_t + 1)^2$.
- $f_{n,k}$ contains the answer.
- $O(nk)$ states, calculating transitions from each state takes $O(n)$ time.
- Overall running time: $O(n^2 k)$.

Subtask 3 dropped the $r_i = c_i$ restriction. We'll describe the similar DP solution for this subtask. It's possible to prove that photo containing points (x, x) and (y, y) covers point (r, c) if and only if segment $[\min(r, c), \max(r, c)]$ is fully contained in segment $[x, y]$ $(x \le y)$. So if we consider segments $[\min(r_i, c_i), \max(r_i, c_i)]$ instead of points (r_i, c_i), the problem is reduced to the following: cover all n segments with k larger segments such that their total area (considering intersections) is minimized.

If segment S_i is included in some other segment S_j, then any photo that covers S_j also covers S_i, so we can safely remove S_i. Removing all such segments can be done in $O(n \log n)$ time:

- First, sort all the segments in order of increasing left endpoint.
- In case of equality, sort them in order of decreasing right endpoint.
- Iterate over all segments in this order.
- If the current segment is included in the last non-removed segment, remove it. Otherwise keep it.

Now, since all left endpoints are increasing and no segment is included in the other, then for all $i < j$: $r_i < r_j$ and $c_i < c_j$. Observations and definition of $f_{i,j}$ are almost identical to previous solution:

- $f_{0,j} = 0$ for all $0 \le j \le k$
- $f_{i,j} = \min\limits_{t<i} f_{t,j-1} + (r_{i-1} - l_t + 1)^2 - \max(0, r_{t-1} - l_t + 1)^2$ (1)
- Last term in this formula accounts for the intersection of segments $t-1$ and t ($t > 0$).
- $f_{n,k}$ contains the answer.
- Overall running time: $O(n^2 k)$.

Subtasks 4 and 5. Here you were required to come up with an optimization of the DP solution described above. **Subtask 4** allowed $O(n^2)$ solutions. One possible solution uses the *Knuth's optimization*.

Define $A_{i,j}$ as the optimal t in (1) and $\text{cost}(t, i) = (r_{i-1} - l_t + 1)^2 - \max(0, r_{t-1} - l_t + 1)^2$.

Lemma: $A_{i,j-1} \le A_{i,j} \le A_{i+1,j}$

This allows us to prune the search space on each step, reducing the running time to $O(n^2)$. If you calculate $f_{i,j}$ in order of increasing j and decreasing i, then at the moment of calculating $f_{i,j}$, values of $A_{i,j-1}$ and $A_{i+1,j}$ are already known, so you can only check $t \in [A_{i,j-1}, A_{i+1,j}]$.

It can be rather difficult to prove the correctness formally, but it's easy to be convinced this is true. A possible strategy for the competition would be to implement this solution without a formal proof, then test the hypothesis on smaller inputs using the solution for

subtask 3. It is known that this optimization results in $O(n^2)$ running time. Also, don't forget about 64 bit integers.

Subtask 5 required a different kind of optimization, running in $O(nk)$ or $O(nk \log n)$ time. Implementing any of the two following optimizations was enough to pass all the tests from this subgroup.

Divide and Conquer optimization ($O(nk \log n)$)

Using the fact that $A_{i-1,j} \leq A_{i,j}$ and that all $f(*, j)$ can be calculated from all $f(*, j-1)$, we can apply divide and conquer optimization.

Consider this recursive function `Calculate`(j, I_{min}, I_{max}, T_{min}, T_{max}) that calculates all $f(i, j)$ for all $i \in [I_{min}, I_{max}]$ and a given j using known $f(*, j-1)$.

function CALCULATE(j, I_{min}, I_{max}, T_{min}, T_{max})
  if $I_{min} > I_{max}$ **then**
    return
  $I_{mid} \leftarrow \frac{1}{2}(I_{min} + I_{max})$
  calculate $f_{I_{mid},j}$ naively, let T_{opt} be the optimal $t \in [T_{min}, T_{max}]$
  CALCULATE(j, I_{min}, $I_{mid} - 1$, T_{min}, T_{opt})
  CALCULATE(j, $I_{mid} + 1$, I_{max}, T_{opt}, T_{max})

The initial call to this function will be `Calculate`(j, 1, n, 0, n) for all j from 1 to k. The time speedup comes up from the fact that all naive calculations of $f_{I_{mid},j}$ on each level of recursion take $O(n)$ in total, because each recursive call splits the segment $[T_{min}, T_{max}]$ into 2 (almost) disjoint segments. The depth of recursion is $O(\log n)$, so the running time of each `Calculate` call is $O(n \log n)$. After calculating all k layers in $O(kn \log n)$ time, we get the answer.

Convex Hull Trick optimization ($O(nk)$)

Another possible optimization is called *Convex Hull Trick*. Let's expand (1):

$$\begin{aligned} f_{i,j} &= \min_{t<i} f_{t,j-1} + (r_{i-1} - l_t + 1)^2 - \max(0, r_{t-1} - l_t + 1)^2 \\ &= \min_{t<i} f_{t,j-1} + r_{i-1}^2 - 2(l_t - 1)r_{i-1} + (l_t - 1)^2 - \max(0, r_{t-1} - l_t + 1)^2 \\ &= C_i + \min_{t<i} M_t r_{i-1} + B_{t,j} \end{aligned}$$

where $C_i = r_{i-1}^2$, $M_t = -2(l_t - 1)$, $B_{t,j} = f_{t,j-1} + (l_t - 1)^2 - \max(0, r_{t-1} - l_t + 1)^2$.

We see that the formula can be expressed in terms of minimum of linear functions $M_t x + B_{t,j}$, evaluated at $x = r_{i-1}$. Notice that as i increases, M_i decreases and r_i increases. That allows

us to maintain the lower envelope of these linear functions using a stack and query the minimum value at given x. Adding a line and querying a point can be implemented in $O(1)$ amortized time, so the total running time is $O(nk)$. This technique is often referred to as the *Convex Hull Trick*. We will also use it to get the 100 point solution for this problem.

Subtask 6. Let's look at $f_{i,k}$ as a function of k and study the differences between two adjacent values. The following theorem states that these differences are *non-increasing*. We'll call such functions *convex*.

Theorem: $f_{i,j-1} - f_{i,j} \geq f_{i,j} - f_{i,j+1}$

Let's assign some constant penalty C for each photo. The new cost function $\widetilde{f}_{i,j} = f_{i,j} + jC$ is still convex, because $\widetilde{f}_{i,j-1} - \widetilde{f}_{i,j} = f_{i,j-1} - f_{i,j} - C$.

Let's introduce another optimization problem without the restriction on number of photos.

$$g_i = \min_{k=1}^{n} \widetilde{f}_{i,k} = \min_{k=1}^{n} (f_{i,k} + kC)$$

This equation for g_i can also be expressed only in terms of previous values of $g_j (j < i)$.

$$g_i = \min_{t<i} g_t + (r_{i-1} - l_t + 1)^2 - \max(0, r_{t-1} - l_t + 1)^2 + C$$

Using this formula, all g_i can be computed in $O(n)$ time using Convex Hull Trick optimization, if all l_i and r_i are sorted beforehand. The solution from subtask 5 can also be modified to find the minimum number of photos required to achieve the optimum, call it $p(C)$.

Since $\widetilde{f}$ is convex, $p(C)$ is also equal to the minimum x such that $\widetilde{f}_{n,x} - \widetilde{f}_{n,x+1} \leq 0$ which is equivalent to $f_{n,x} - f_{n,x+1} \leq C$, so $p(C)$ is monotone. Also if we set $C = 0$, optimum value of g_n is achieved with the n photos, and if we set $C = M^2$, then the optimal solution only contains one photo. Combining everything above, we can use binary search to find such C_{opt} that $p_1 = p(C_{opt}) \geq k$ and $p_2 = p(C_{opt} + 1) \leq k$.

This means that all differences $(f_{n,p_2} - f_{n,p_2+1}), (f_{n,p_2+1} - f_{n,p_2+2}), \ldots, (f_{n,p_1-1} - f_{n,p_1})$ are equal, and $f_{n,j}$ is a linear function of j on this interval. Since the desired value of $f_{n,k}$ is somewhere in this interval, it's possible to calculate it just by linear interpolation, because all slopes are equal.

This solution requires sorting the segments once and doing $O(\log m)$ iterations of binary search to find C_{opt}, each iteration running in linear time. Total running time: $O(n \log n + n \log m)$.