Problem A. Problem Statement

Each press of the Enter key increases the number of parts by 1.

At the beginning, there was 1 part, and at the end — there are n parts, which means $(n-1)$ presses were made.

Problem B. Ant Hill

Let's consider the prefix sums of the array a:

- $p_0 = 0$;
- $p_1 = a_1$;
- $p_2 = a_1 + a_2$;
- ...
- $p_n = a_1 + a_2 + \cdots + a_n$.

It is clear that p_i represents the change in the number of ants inside the anthill after the first i observations (hence $p_0 = 0$).

There could not be a negative number of ants in the anthill at any moment in time, which can be expressed as:

$ans + p_i \geq 0$ for all $i = 0 \dots n$.

It is easy to notice that everything actually depends only on the minimum value of p_i:

$ans + \min(p) \geq 0$, from which it follows that $ans \geq -\min(p)$.

We need to choose the minimal answer, so the inequality turns into an equality.

Note that to solve the problem, it was not even necessary to create the array p — it was sufficient to have a single variable "total change after the current observation":

```
delta = 0;
min_delta = delta;

for (value in a) {
    delta += value;
    min_delta = min(min_delta, delta)
}

ans = -min_delta
```

It was also important to note that the maximum possible answer was on the order of $10^5 \cdot 10^6 = 10^{11}$, which could not fit into a 32-bit integer type, so it was necessary to use a 64-bit type (`long long` or `int64_t` in C++, `long` in Java).

Problem C. Linear Maze

We will sequentially compute the answer for the prefix of rooms $1 \dots i$.

- Entering any room increases the current answer by 1.
- If the room is a junction, then we will double the current answer.

Why does this method of calculation work?

When we first enter a junction room (let's denote it as x), we have to start the entire path to it from the beginning.

Suppose we have traveled the path from 1 to x a second time (and it could be different from the first time).

The most important thing is that after exiting room x for the second time, its state will return to its original position.

Accordingly, if we later enter room x again, trying to travel for the second time to the junction room $y > x$, we will repeat all the same steps as before.

From this, it follows that for each junction room, the length of the path simply doubles.

Problem D. Grouping

In this problem, a "greedy" approach is used.

First, let's note that the initial order of elements in the array does not affect anything, so we will sort it in ascending order.

It is claimed that there exists an optimal solution consisting only of subsegments of the sorted array a.

We will prove this constructively.

Suppose there exists an optimal solution such that a_i and a_{i+2} belong to the first group, while a_{i-1} and a_{i+1} belong to the second group.

Consider a distribution that differs from the optimal one in only one way: a_{i+1} and a_{i+2} are now in the first group, while a_{i-1} and a_i are in the second.

Thus:

- The number of groups has not changed.
- The size of the groups has not changed.
- The difference of elements in the first group was at least $a_{i+2} - a_i$, and has now become at least $a_{i+2} - a_{i+1}$.
- The difference of elements in the first group was at least $a_{i+1} - a_{i-1}$, and has now become at least $a_i - a_{i-1}$.

Since $a_{i+1} \geq a_i$, both differences could only have decreased (or remained unchanged), which means the new distribution is also valid.

Thus, from any optimal distribution, we can obtain an optimal one where the groups form subsegments of the array.

To solve the problem, it was sufficient to process the elements of the array in ascending order, maintaining the position and value of the first element in the current segment:

```
ans_cnt = 0

first = neg_inf
first_pos = -1

for (i = 0; i < n; i += 1) {
  if (a[i] - first > M or i - first_pos >= k) {
    ans_cnt += 1
```

```
        first = a[i]
        first_pos = i
    }
}
```

Problem E. Mountain Ranges

It is obvious that if a suitable k exists, it lies in the interval $[1; \max(h) - \min(h)]$ — let us denote the length of the interval as H.

First approach to the solution

We will construct the function $F(k)$ — the number of mountain chains for a fixed similarity parameter k.

The function itself can be implemented in $O(n)$ in a fairly straightforward way:

- We maintain a counter for mountain chains, initially set to 1.
- We process the mountains from left to right, starting from the second mountain.
- If the difference between the mountains exceeds the similarity parameter, we increase the counter by 1.

However, the solution with $O(H \cdot n)$ is too inefficient.

For further optimization, we note that as the parameter k increases, the value of the function $F(k)$ can either remain the same or decrease: $F(1) \geq \cdots \geq F(H-1) \geq F(H)$.

Accordingly, if $m < F(H)$ or $F(1) < m$ — there is no answer in principle.

Otherwise, we introduce an additional function $G(k) = (F(k) \leq m)$.

- If $G(k)$ is true, then $G(k+1)$ is also true.
- If $G(k)$ is false, then $G(k-1)$ is also false.
- We take $G(0)$ to be false, and $G(H+1)$ to be true.

It is clear that the function $G(k)$ is monotonically decreasing, so we can perform a **binary search** on the answer.

Note that after performing the binary search for the answer, we need to check the found value k_{opt} — there may be a situation where $F(k_{opt}) < m < F(k_{opt}+1)$. In such a case, there is also no answer.

The final asymptotic complexity will be $O(\log H \cdot n)$.

Second approach to the solution

We will compute the array $D_i = |H_{i+1} - H_i|$ and sort it in non-decreasing order.

We will also augment the array with $D_0 = -1$ and $D_n = H + 1$.

Let for the similarity parameter k the inequality $D_i \leq k < D_{i+1}$ hold.

In this case, we can confidently assert that for the given parameter k, exactly $n - i$ mountain chains will be formed.

Why is that? Let us call the condition "adjacent mountains are in the same chain" a **connection**. In this case, a new mountain chain is formed every time a connection is broken.

- When $k = (H+1)$, all mountains are in one chain — that is, there are $(n-1)$ connections.
- If $k < D_{i+1}$, then all connections from $i+1$ to $(n-1)$ are broken.

- In total, $n - (i + 1)$ connections are broken, which means the number of mountain chains becomes $1 + n - (i + 1) = n - i$.

We need to obtain exactly m mountain chains, which means k must be in the half-interval $[D_{n-m}; D_{n-m+1})$.

If this half-interval is empty — then there is no answer. Otherwise, we need to take the minimum value — that is, D_{n-m}.

The final asymptotic complexity turns out to be $O(n \cdot \log n)$ due to sorting.

Problem F. Traffic Lights

This problem is solved by carefully emulating the described process.

We will keep track of the current time since the start of the trip in the variable *time*.

When we approach the i-th intersection — we increase *time* by a_i.

Now we need to understand what color is currently lit at the i-th traffic light.

We know that d_i seconds before the start of the trip, the green light was on. Since that moment, a total of $total_i = (time + d_i)$ seconds have passed.

The process of changing colors is cyclical with a period of $period_i = (g_i + r_i)$.

We will calculate the remainder $rem_i = total_i \mod period_i$. This will represent the time that has passed since the start of the last lighting period.

If $rem_i < g_i$, then the bus will pass through the intersection immediately; otherwise, we need to add to the time the value $period_i - rem_i$ — exactly how long we need to wait at the red light.

Problem G. Need More Gold

To begin with, we will show that the main idea of solving this problem is greedy.

Let's assume we have found some optimal answer, where we consecutively defeated monster j first, and immediately after that, monster i, with the condition that $g_j < g_i$.

We will show under what circumstances we can swap these monsters (and only these) so that the answer remains optimal.

Let's denote the amount of gold before the battle with monster j as cw.

In this case, we can immediately establish two statements:

- $b_j \le cw$;
- $b_i \le cw + g_j$.

First, note that for the monsters fought after this pair, nothing will change—the total amount of gold after both victories will be $cw + g_i + g_j = c_w + g_j + g_i$.

Secondly, let's consider two cases:

- $cw < b_i$ — in this case, at the moment of the battle with monster j, we could not have defeated monster i.
- $b_i \le cw$ — in this case, we could swap the battles with monsters j and i:
  - We can defeat monster i by our assumption;
  - We can defeat monster j, since $b_j \le cw \le cw + g_i$ ($g_i \ge 0$ by the problem statement).

Thus, we have shown the following: if at the moment we **can defeat** monsters $m_1, m_2, \ldots, m_c$, then to obtain an optimal answer, we can always choose the monster with the highest value of g_{m_i}.

At this point, we can already implement a solution where we will choose the optimal monster each time in $O(n)$, resulting in an overall asymptotic complexity of $O(n^2)$—correct, but not efficient enough.

For further optimization, we will need a data structure (DS) that can efficiently perform the following operations:

- add a value;
- return the extremum (minimum or maximum);
- extract the corresponding extremum.

It is known that data structures like "binary heap" and "search tree" can perform all these operations in $O(\log N)$ or more efficiently:

- C++: `std::priority_queue` and `std::set`;
- Java: `PriorityQueue` and `TreeSet`;
- Python: the `heapq` package.

First, we will add all monsters to the DS *alive*, which will search for the minimum by the value of b_i. This structure will store monsters that are still alive, but we cannot defeat them yet.

We will also maintain the DS *ready*, which will search for the maximum by the value of g_i. This structure will store monsters that are still alive, but we are now capable of defeating them.

Each step will consist of several parts:

- If $cw \geq w$, then the processing is complete.
- Extract monsters from *alive* with the minimum value of b_i, while $b_i \leq cw$.
- All monsters extracted in this way are added to *ready*.
- If *ready* is empty, then the processing is complete.
- Otherwise, extract from *ready* the monster with the maximum value of g_i, after which we increase cw by g_i and the count of defeated monsters by 1.

Problem H. Hierarchy

The structure of each company forms a **tree** (and all companies together form a **forest**).

In this case, the CEO is the root of the corresponding tree, and the number of employees in his company is the number of vertices in this tree.

The number of vertices in each tree of the given forest can be found in two ways:

- a complete traversal of each tree using depth-first search / breadth-first search;
- using a disjoint set union (DSU) system.

Note that when solving through traversals, it is recommended to use **adjacency lists** to store the graph structure.

The final asymptotic complexity of the solution will be $O(n)$ for the traversal method and $O(n \cdot D)$ for the DSU method, where D is the complexity of DSU operations (depends on the chosen implementation).

Problem I. Study Day

The problem can be solved using dynamic programming.

Let dpO_i be the maximum total knowledge you will gain by attending lectures up to and including lecture i if you did not meditate before lecture i.

Similarly, let dpM_i be the maximum total knowledge you will gain by attending lectures up to and including lecture i if you meditated before lecture i.

The transitions will be as follows:

- $dpO_0 = dpM_0 = 0$ — there is no knowledge before attending the lectures;
- $dpO_i = \max(dpO_{i-1}, dpM_{i-1}) + a_i$ — attending a lecture without meditation can be done without restrictions;
- $dpM_i = dpO_{i-1} + 2 \cdot a_i$ — you cannot attend two lectures in a row with meditation.

The answer will be $\max(dpO_n, dpM_n)$.

To reconstruct the optimal order of actions, you can iterate through the lectures in reverse order, starting from the optimal state (O or M):

- If we are in the M state, then the previous action must be O;
- If we are in the O state, then the previous state is O if $dpO_{i-1} > dpM_{i-1}$, otherwise it is M.

In the end, we will obtain the sequence of actions in reverse order.