The short shank; Redemption (prison)

BY NILS GUSTAFSSON, LOKE GUSTAFSSON, AND GUSTAV KALANDER (SWEDEN)

Subtask 1. $N \le 500$

For any $0 \le a \le n \le N$, $0 \le d \le D$ let $X_{n,d,a}$ denote the minimum number of prisoners that will rebel at time T if we only consider the first n prisoners and the wardens place at most d mattresses, the last of which is positioned immediately to the left of a (if $d = 0$, we ignore the value of a). In particular, the solution to the original problem is just $\min_{1 \le a \le N} X_{N,D,a}$.

We can compute all $X_{n,d,a}$ via dynamic programming as follows:

- Obviously, $X_{0,d,0} = 0$ for all d.
- If $0 < m < n$, then $X_{n,d,m}$ is either $X_{n-1,d,m}$ or $X_{n-1,d,m} + 1$; more precisely, the former case happens if and only if prisoner n will not rebel for some hence any placement of the mattresses such that the last mattress is at position m. This is in turn equivalent to $\min\{t_m + n - m, t_{m+1} + n - m - 1, \dots, t_n\} > T$.
- If $m = n > 0$ (i.e. there's a mattress immediately to the left of prisoner n), then $X_{n,d,m}$ can be computed as $\min_{m \le n-1} X_{n-1,d-1,m}$ or $\min_{m \le n-1} X_{n-1,d-1,m} + 1$, depending on whether prisoner n will rebel at time T. This is in turn equivalent to $t_n \le T$.

If we compute all these in order of increasing n and decreasing m, we can update the minimum in the second case in constant time for a total runtime of $O(N^2 D)$.

Subtask 2. $N \le 500\,000$, $D = 1$

If we want to place just a single mattress, then we have enough time to check all possibilities provided that we can compute the number of prisoners rebelling for a given placement in $O(1)$ after some precomputation.

For this, we consider prisoners to the left of our mattress and to the right separately. For the first case, it suffices to just sweep through the prisoners once from left to right to compute for any $n \le N$ how many of the first n prisoners will rebel if we never place a mattress.

The computation for the prisoners on the right of the mattress is slightly more involved as moving the mattress from left to right can affect basically any prisoner on the right. Instead, we sweep from right to left and keep a stack of those prisoners that will *not* rebel. When the mattress moves one step to the left, so that prisoner n is now to the right of it, all prisoners $n + 1, \dots, n + t_n - T$ will start to rebel unless they already did so. We can therefore just pop elements from our stack until we see the first prisoner to the right of this (and push n when necessary), which has constant amortized time.

Subtask 3. $N \le 4\,000$

This is another dynamic programming subtask, but we have to think backwards this time: instead of computing $X_{n,d,m}$ as above, we define $\tilde{X}_{n,k,m}$ as the minimal number of mattresses it takes to ensure that at most k of the first n prisoners rebel and such that the rightmost mattress is placed somewhere to the right of prisoner m. Of course, there are again $O(N^3)$ DP states, but we can make the following observation: $X_{n,k,\bullet}$ is obviously increasing in m, and moreover $X_{n,k,1}$ and $X_{n,k,m}$ differ by no more than 1; namely, take any placement of $X_{n,k,1}$ mattresses such that only k prisoners rebel and just add in an unnecessary mattress on your favourite place to the right of m.

Thus, instead of computing $\tilde{X}_{n,k,m}$ we can simply compute $Y_{n,k} := X_{n,k,1}$ together with the rightmost mattress $Z_{n,k}$ of any placement of $Y_{n,k}$ mattresses such that only k prisoners rebel. To compute these, we go through them in order of increasing n, making a case distinction whether we put a mattress immediately to the left of n and if not, whether prisoner n will rebel or not. The total runtime is $O(N^2)$.

Subtask 4. $N \leq 75\,000$, $D \leq 15$

This subtask can be solved in $O(ND \log N)$ by speeding up the solution to the very first subtask by putting its entries into a segment tree and doing all the updates for $n \leq m - 1$ at the same time when we move from $n - 1$ to n.

Alternatively, an inefficient implementation of the second half of the full solution sketched below takes $O(ND)$ time, which then of course also solves this subtask.

Subtask 5. $N \leq 75\,000$

For this subtask, one can further optimize the DP solution from the previous subtask to get an $O(N \log^2 N)$ solution. This requires one of several advanced, but standard techniques like Lagrange multipliers (also referred to as the "Alien trick" because of its use in the sample solution to IOI 2016's task Aliens) or using divide and conquer. However, we think that the full solution is actually easier than this.

Subtask 6. No further constraints.

To describe the full solution, let us call a prisoner n *passive* when they would not rebel if the wardens placed a mattress immediately to their left (i.e. $t_n > T$) but they would do so if the wardens didn't place any mattress.

If we have any optimal solution, we can move any mattress not to the left of an passive prisoner to the right until we hit a passive prisoner without changing the number of rebels: namely, if we move the mattress past a non-passive prisoner n, this does not affect their own state at time T by definition, and for any prisoner to the right of n the time where he will start rebelling can only increase.

Thus, we can restrict to placements of (at most) D mattresses where all mattresses are immediately to the left of passive prisoners. Moreover, putting up mattresses will only ever affect the state at time T of passive prisoners, so we simply want to maximize the number of passive prisoners that do not rebel at time T.

The crucial insight then is the following: we obtain a rooted forest with nodes the passive prisoners by declaring the parent of an inactive prisoner n to be the next passive prisoner $\pi(n)$ to the right of n such that $\pi(n)$ would not rebel if the wardens put a mattress just to the left of n (if they exist), and we can easily describe the effect of putting up a mattress graph-theoretically:

Lemma 1. *Putting a mattress immediately to the left of a passive prisoner n results in prisoners on the unique path from n to the root of its component (i.e. prisoners $n, \pi(n), \pi(\pi(n)), \dots$) not rebelling at time T, and does not affect the state at time T of any other prisoner.*

Proof. Obviously, putting up a mattress to the left of n does not affect the behaviour of prisoners to the left of n. By definition, neither n nor $\pi(n)$ will rebel at time T if we put a mattress immediately to

the left of n, while the state of no prisoner strictly between n and $\pi(n)$ is affected by this. The same argument shows that if $\pi(n)$ is undefined, the state of no prisoner to the right of n is affected.

Now consider a passive prisoner m to the right of $\pi(n)$. We claim that m will not rebel at time T when placing a mattress immediately to the left of n if and only if they would not rebel when placing a mattress immediately to the left of $\pi(n)$. Indeed, the implication '$\Rightarrow$' is always true for trivial reasons. For '$\Leftarrow$' observe that if a passive prisoner to the right of $\pi(n)$ would rebel at time T when putting up a mattress immediately to the left of n but not when placing one immediately to the left of n, then the incentive would have had to come from some prisoner between n and $\pi(n)$ which would have lead to $\pi(n)$ rebelling at time T, contradicting the definition of π.

Thus, the lemma follows by inductively applying the above argument to $n, \pi(n), \pi^2(n), \ldots$ □

Thus we are left with the following problem: given a forest, select at most D nodes such that the union of the paths to their corresponding roots is as large as possible. In order to efficiently solve this, we now observe:

Lemma 2. *Assume $D > 0$ and let v be any node of maximum depth. Then there exists an optimal solution containing v.*

Proof. Fix any optimal solution. We start at v and we go upwards until we for the first time hit a node m that lies on a path from some node w of our solution to the root. We claim that replacing w by v produces a solution not worse than the original one (hence again optimal).

Indeed, this can only remove the nodes between w (inclusive) and m (exclusive) from the union of our paths, while definitely adding in all nodes between v (inclusive) and m (exclusive); note that none of the latter were part of the original union by choice of m. The claim follows as v is of maximum depth, so that there at most as many nodes between w and m than there are between v and m. □

So what should we do after picking the first node v? Well, if we have any set S of nodes containing v, then trimming the path from a node $w \in S - \{v\}$ to its root so that it already stops the first time it hits the path from w to its root, does not affect the union of the paths. Thus, (optimal) solutions with at most D nodes and containing v are in $1:1$ correspondence with (optimal) solutions with $D-1$ nodes in the forest obtained by removing the path from v to its root. Applying the previous lemma inductively we therefore see that we can construct an optimal solution by iteratively choosing a node of a maximum depth and removing all nodes on the path to the root (inclusive).

It remains to demonstrate how we can (1) efficiently compute π and (2) efficiently perform the above greedy procedure.

First note that the naïve way to construct π is quadratic (or worse), which is too slow and would only solve Subtask 3 again. Instead, we sweep through the prisoners from left to right, keeping a stack of all passive prisoners n whose $\pi(n)$ we have not yet encountered. Whenever we encounter a new passive prisoner m, the prisoners n on the stack with $\pi(n) = m$ will form a contiguous (possibly empty) sequence starting on the top. This is because a prisoner still on the stack will have $\pi(n) > m$ (i.e. putting a mattress to the left of them will not prevent m from rebelling at time T) if and only if there is some prisoner k to the right of them (including themselves) with $k + T - t_k \geq m$.

Thus, we can find all n with $\pi(n) = m$ in constant amortized time by popping elements from our stack until we encounter a prisoner k with $k + T - t_k \geq m$, yielding a linear time implementation of the first part. Alternatively, this part can be implemented with a segment tree together with a stack in $O(N \log N)$ time which should still be fast enough.

For the second part, the naïve implementation takes time $O(ND)$ which suffices for Subtask 4. For a more clever implementation, we keep a priority queue Q of all roots sorted by the height of their component and we keep for each node a pointer to a child with maximum subtree height. Then, when we extract a node from Q (which we do at most D times), we use these pointers to traverse the path we are actually removing from our forest and delete all these nodes. Afterwards, we push all their children into the queue (as they become new roots). Since we do at most N pushes and traverse each node at most once, the total runtime is $O(N \log N)$.

In fact, one can also implement this step in linear time as well by implementing Q using an array of vectors (indexed by subtree height) and putting all nodes in there right at the beginning (which does not hurt since any non-root will come after their parent anyhow). Instead of actually removing a node from the queue, we mark it as deleted in a second array and just ignore it when we extract it. Since this only requires us to traverse the forest and the queue once, the total runtime is indeed $O(N)$.

The Collection Game (swaps)

BY ANTTI RÖYSKÖ (FINLAND)

Let $A_1, \ldots, A_N$ be such that room i has the A_i-th most aesthetic art collection. Note that computing $A_1, \ldots, A_N$ suffices to solve the task because *answer* should just output all rooms ordered according to increasing A_i. In this reformulation, a call to *schedule*(i, j) will then return whether $A_i < A_j$, potentially swapping A_i and A_j beforehand. So, we seek to determine $A_1, \ldots, A_N$ after our last call to *visit*.

Subtask 1. $V = 5\,000$ and the museum never swaps art collections.

Because the A_i will never get swapped in the first two subtasks, a call to *schedule*(i, j) will determine whether $A_i < A_j$ after the last call to *visit*, and so whether room i should appear before room j in the output. Therefore, we only have to sort all rooms using our calls to *schedule* and *visit*.

For Subtask 1, a solution calling *schedule* and directly afterwards *visit* every time can in total call these functions $N\lceil \log N \rceil$ times without exceeding V visits. So, we can use any sorting algorithm which needs only that many comparisons to sort all the rooms.

Note that `sort` in C++ will not suffice for this purpose. However, we can insert the rooms one-by-one into a `vector` using binary search, for example with `lower_bound`. Also, a custom implementation of, for example, merge sort will work.

Subtask 2. $V \geq 1\,000$ and the museum never swaps art collections.

One option here is to use that fact that merge sort splits the list to be sorted into two equal halves which can then be sorted independently before merging them to obtain a sorting of the whole list. Since these sublists can be sorted independently and because we can compare disjoint pairs of indices at the same time, we can sort the two halves simultaneously. On the highest level, we will then need $N - 1$ comparisons to merge the lists, on the second level we will need $\lceil N/2 \rceil - 1$ comparisons for the merging, then we will need $\lceil N/4 \rceil - 1$ comparisons and so on until 1 comparison on the lowest level. In total, this gives $(N - 1) + (\lceil N/2 \rceil - 1) + (\lceil N/4 \rceil - 1) + \cdots + 1 \leq N + N/2 + N/4 + \cdots + 1 \leq 2N$ comparisons, solving Subtask 2.

There is also another solution which just compares all pairs of rooms. This can be done in parallel using a total of $2N$ calls to *visit*, and it is easy to reconstruct the order all rooms from these comparisons, for example by a topological sort. This also solves Subtask 2.

Subtask 3. $N \leq 100$, $V = 5\,000$

In this subtask, we can iteratively single out the smallest element A_j and recurse on the remaining elements.* To do so, we iterate $i = 1, \ldots, N$ and keep the index j of the currently smallest element A_j among $A_1, \ldots, A_i$. For $i = 1$, this is simply $j = 1$. Then, when moving from i to $i + 1$, we compare the elements at indices j and $i + 1$ (calling *schedule* and *visit* each once), and set j to the smaller of the two elements. Note that even if A_j and A_{i+1} are swapped beforehand, this will yield the smallest element among $A_1, \ldots, A_{i+1}$.

* Also called “selection sort.”

This approach needs $N - 1$ calls to *visit* to determine the smallest element of the list and afterwards reduces N by one. So, the total number of call to *visit* will be $(N-1)+(N-2)+\cdots+1 = N(N-1)/2 \leq N^2/2$ which suffices for Subtask 3.

Subtask 4. $V = 5\,000$

In fact, it is possible to determine the smallest element of the list using only $\lceil \log N \rceil$ many calls to *visit*. One way to do so is by creating a single-elimination tournament amongst the elements with $\lceil \log N \rceil$ many rounds.[†] That is, with one call to *visit* (and $N/2$ calls to *schedule*) we determine the smaller element of A_1 and A_2, of A_3 and A_4, of A_5 and A_6, and so on. Afterwards, there will only remain $\lceil N/2 \rceil$ many elements on which we can repeat this procedure until we are with left with only the smallest element of the whole list.

By proceeding as above on the remaining $N - 1$ elements, we can solve the problem using at most $N\lceil \log N \rceil$ calls to *visit*, solving Subtask 4.

Subtask 5. $V \geq 500$

Essentially, the two previous subtasks used selection sort and heap sort. What other sorting algorithm is there?

Well, there is bubble sort, but that needs N^2 many bubble steps. However, note that we can parallelize those bubble steps. Namely, in a single phase we could bubble the elements A_1 and A_2, the elements A_3 and A_4, and so on simultaneously. The same applies to A_2 and A_3, A_4 and A_5, and so on. By alternating between these two phases, this would eventually sort the list.[‡]

Two problems remain. First, how many phases does this process need? It turns out that it needs only N phases[§], as can be proven by a submission during the contest. Therefore, this would suffice for Subtask 5.

However, we still have to implement these phase using the functions *schedule* and *visit*. If *schedule*(i, j) always puts the smaller element of A_i and A_j into A_i, this works out of the box, earning 60% of the points for this subtask.

If not, we can use the following observation: Any solution relying on the fact that the smaller element gets moved into A_i for every call to *schedule*(i, j) can actually be adapted to a full solution as follows. If *visit* returns that the smaller element is in A_j, we simply exchange the indices i and j in every query from now on, and so we may assume that the smaller element will be A_i after a call to *schedule*. Note that many implementations of the above idea will do this implicitly anyway.

Subtask 6. $V \geq 100$

From this subtask on, it gets considerable harder to solve the problem. It turns out that the special case where the smaller element is always moved into A_i for each call to *schedule*(i, j) is the problem of designing sorting networks, and as we know from the previous subtask, this special case suffices to solve the problem completely.

[†] Also called “heap.”
[‡] Also called “odd-even sort.”
[§] See Wikipedia.

In Subtask 6, any sorting network of depth $\leq \lceil \log N \rceil (\lceil \log N \rceil + 1)$ will suffice to solve the problem. Here, it is possible to come up with your own solutions of a sorting network of such a depth, but which does not quite solve the next subtask.

For example, we can go for a merge sort style approach which first sorts the two halves of the list recursively and then merges them. To do this merging, we can iterate over a gap size $g = 2^k, 2^{k-1}, \ldots, 1$ where k is the maximal value satisfying $2^k \leq N$ and bubble each element A_i with A_{i+g} (using two calls to *visit*). It can be proven that this sorting network correctly sorts any list.[¶]

Subtask 7. $V \geq 50$

For the last subtask, we need a sorting network of depth $\leq \lceil \log N \rceil (\lceil \log N \rceil + 1)/2$ to obtain all the points. Some standard sorting networks will work for that, for example bitonic mergesort, Batcher's odd-even mergesort or the pairwise sorting network. But it is also possible to come up with your own solutions.

[¶] Using the zero-one principle from Wikipedia.

The Xana coup (xanadu)

BY LUKAS MICHEL (GERMANY)

Abridged problem statement. You are given a tree with N vertices, where every node is initially either turned on or off. Every vertex has a button. Now you can execute the following operation several times: You press the button at vertex v which toggles vertex v and all of its direct neighbors. Toggling a vertex means turning it on if it's currently turned off and vice versa. Now you are to find the minimal number of button presses needed to turn off all vertices.

There are a few important insights in this problem.

- For every vertex, it's only important whether we toggled this vertex an even or an odd number off times. It's easy to see this: After toggling a vertex once, it changes its state, and if we toggle it a second time, it returns to its initial state.
- The order of the button presses doesn't matter.
- It doesn't make sense to press a button more than once. This would toggle the vertex and its neighbors an even number of times, causing no change.
- So we will press every button either 0 or 1 times. This means the solution will either be between 0 and N or it will be impossible to turn off all vertices.

Subtask 1. $N \le 20$

After observing that we have to press every button either 0 or 1 times, the first subtask is really straightforward. Just try all possible subsets of buttons to press. For every subset check if pressing those buttons turns off all nodes and then choose the subset with the minimal number of buttons that turns off all nodes. This yields an easy $O(N \cdot 2^N)$ solution that should easily fit into the time limit.

Subtask 2. $N \le 40$

There are several solutions for the second subtask. One possible solution is to first find a centroid. We root the tree at this centroid. Then, every subtree will have at most $N/2$ vertices. Let's first assume we don't press the centroid's button. We now run the brute force from subtask 1 on every subtree. This way, we determine the minimal number of button presses if we press the button of the root of this subtree and the minimal number of button presses if we don't press the button of the root of this subtree. Now, we have to combine the solutions of the subtrees. Obviously, if the centroid is turned on, we have to press an odd number of children's buttons, and an even number otherwise.

To combine the solutions efficiently, we can iterate over all children. We have two variables *even* and *odd* that represent the minimal number of button presses if we want to press an even or odd number of children's buttons. *even* is initially set to 0 and *odd* to ∞. Now let's assume we are currently processing a child. Let p_1 be the minimal number of button presses to turn off the subtree of the child if we want to press the child's button, and p_2 the same value if we don't want to press the child's button. Then we set $even = min(odd + p_1, even + p_2)$ and $odd = min(even + p_1, odd + p_2)$. The logic behind this: If we want to press an even number, we can either press an odd number before and then press the current child's button, or we press an even number before and we don't press the current child's button.

Of course, after having processed all children, our result is in *even* if the centroid was turned off and in *odd* if the centroid was turned on. Now, we press the button of the centroid and repeat the same procedure again, adding 1 at the end. This solution runs in $O(N \cdot 2^{N/2})$.

Another, perhaps more approachable solution starts by dividing the tree into its bipartite parts, i.e. dividing the set of vertices into two sets such that every edge has a vertex in each of them. Then, the smaller of these parts has at most $N/2$ vertices; let's call the set of vertices in this part V_1 and the other one V_2. Note that whether a vertex in V_2 is turned on only depends on this vertex's button and on buttons of vertices in V_1.

We can now brute force all possible button presses for V_1. With these fixed, the state of any vertex in V_2 only depends on its own button; this means that we know whether we have to press it so that the vertex is off in the end. Now that all buttons in V_2 are fixed, we can check whether the vertices in V_1 are all off. If they are, we have a possible solution. We then just take the minimal number of button presses over all solutions. This algorithm also runs in $O(N \cdot 2^{N/2})$.

Subtask 3. The graph is a line.

Vertex 1 will only have one neighbor, vertex 2. Imagine we press the button at vertex 1. Then, we already know whether we have to press the button at vertex 2 or not: If vertex 1 was previously turned off, we will have to press button 2, and if vertex 1 was previously turned on, we should not press button 2. The same holds for all other buttons.

So, deciding whether we press the first button already determines what buttons we press. This yields an $O(N)$ solution for this subtask: Try both options (pressing or not pressing the first button) and choose the option that turns off all vertices in the minimal number of button presses.

Subtask 4. The graph is a binary tree.

If every vertex is directly connected to at most 3 other vertices, we can choose a root r such that every vertex has at most 2 direct children.

We can observe that any vertex v only depends on its parent and its ≤ 2 children. Now we can do dynamic programming on the tree. For every vertex v we compute 4 values:

- $dp[0][0][v]$: Assuming we didn't press the button of v's parent and we don't want to press button v, what is the minimal number of button presses to turn off all nodes in v's subtree? This should be set to ∞ if it's impossible to turn off all vertices under these constraints.
- $dp[0][1][v]$: Now we assume that we didn't press the button of v's parent, but we want to press button v.
- $dp[1][0][v]$: Now we did press the button of v's parent and we don't want to press button v.
- $dp[1][1][v]$: Now we assume that we press both the button of v's parent and button v.

Now let's assume we have calculated dp for all children and now we want to calculate $dp[i][j][v]$. First we check whether after pressing the button of v's parent i times and button v j times (after toggling v $i+j$ times) vertex v is turned on or off. If v is already turned off, we have to press either both or none of the children's buttons. Otherwise we should press exactly one of the children's buttons. So, we simply use the dp-values that we calculated for the children. The overall solution to the problem then is $min(dp[0][0][r], dp[0][1][r])$ because we can either push button r or not push button r. This solution runs in $O(N)$.

Subtask 5. No further constraints.

For the full solution we can use the same dynamic programming that we used for subtask 4. We just have to change how we calculate $dp[i][j][v]$. Assume that v is turned off after toggling it $i + j$ times. Then, we have to press an even number of children's buttons. Otherwise, we should press an odd number of children's buttons.

There are multiple ways to calculate that efficiently, here is one: We use an idea similar to the one we used for solving subtask 2. For every node, we temporarily calculate 4 values:

- $tmp[0][0]$: This should be set to the minimal number of button presses to turn off all vertices in the children's subtrees if v is not pressed and we press an even number of children's buttons.
- $tmp[0][1]$: Same as before, but we press an odd number of children's buttons.
- $tmp[1][0]$: Now we want to press v and an even number of children's buttons.
- $tmp[1][1]$: Here we want to press v and an odd number of children's buttons.

How can we calculate those? We do almost the same as in subtask 2: We initialize the odd tmp-values with ∞ and the even ones with 0. Now let's iterate over all children and calculate their dp. Assume we are currently processing child c. Then the we change tmp in the following way:

- $tmp[0][0] = min(tmp[1][0] + dp[0][1][c], tmp[0][0] + dp[0][0][c])$
  We don't press v, so we use $dp[0]$ in both cases. We want to press an even number of children's buttons, so we can either press an odd number before ($tmp[1][0]$) and press c or we press an even number ($tmp[0][0]$) and don't press c.
- $tmp[0][1] = min(tmp[1][1] + dp[1][1][c], tmp[0][1] + dp[1][0][c])$
- $tmp[1][0] = min(tmp[0][0] + dp[0][1][c], tmp[1][0] + dp[0][0][c])$
- $tmp[1][1] = min(tmp[0][1] + dp[1][1][c], tmp[1][1] + dp[1][0][c])$

Now if vertex v is turned on after being toggled $i + j$ times, then $dp[i][j][v] = j + tmp[1][j]$, because we need an odd number of children's buttons pressed. Else, $dp[i][j][v] = j + tmp[0][j]$. This solution runs in $O(N)$.