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 $\bar{X}_{n,k,m}$ 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 $\bar{X}_{n,k,m}$ 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 $n, \pi(n), \pi(\pi(n)), \ldots$) 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 $O(N)$.