EGOI 2026 Editorial - Biscuits

Task Author: Nemanja Majski

In this task, we are given a stack of N biscuits with known weights. Aurora first chooses an integer $X \geq 0$. Bianca then chooses an integer $Y \geq 0$ such that $Y \neq X$ and at least Y biscuits are still in the stack. Aurora eats the top Y biscuits. If any biscuits remain after that, Bianca eats the next one.

Both players play optimally to maximize the total weight of biscuits they eat. Our task is to find the total weight of biscuits that Bianca can eat for the initial stack, and also for the stack obtained after each of the Q updates, where one biscuit is replaced by another biscuit with a possibly different weight.

Subtask 1: $Q = 0$ and all $W_i = 1$

When every biscuit weighs the same, both players only care about how many biscuits they eat.

In this case, Bianca will always choose Y as small as possible, to leave as little biscuits as possible to Aurora. Therefore, Aurora will choose $X = 0$ in each turn, to guarantee that she will get to eat at least one biscuit, and then Bianca will choose $Y = 1$ to guarantee that Aurora gets to eat just one biscuit.

Therefore Bianca gets to eat every second biscuit, starting with the second topmost biscuit on stack - summing this together, she gets to eat biscuits with total weight of

$$\left\lfloor \frac{N}{2} \right\rfloor.$$

Note that there are no updates in this subtask, so we just print this value once and we are done.

Subtask 2: $N \leq 3, Q \leq 5$

For very small N, we can try all possible choices of X and Y at each step using bruteforce.

We make a function that is given an array of weights of biscuits in a stack and computes the optimal score of the game recursively.

When trying to find the score that Bianca will be able to obtain in this situation, we can simply try all possible X that Aurora can choose, then all possible Y that Bianca can choose, such that $Y \neq X$. We can then recursively find the

optimal score with which the game will be finished after the performed turn. To get the value of our turn, we add W_Y to previously calculated score.

Bianca will then choose the Y that maximizes this score, and Aurora will choose X that minimizes it, among all possible choices of Y.

There are at most four possible values of X and Y, so this is fast enough for $N \leq 3$ and $Q \leq 5$.

Subtask 3: Weights are non-increasing

More formally, in this subtask, we are guaranteed that at any point

$$W_0 \geq W_1 \geq \cdots \geq W_{N-1}.$$

Claim: Bianca can guarantee her score to be at least $W_1 + W_3 + W_5 + ... + W_{N-1-(N \bmod 2)}$.

We will prove this. We start by demonstrating a working strategy for Bianca to achieve at least as high score for all N and proving it by induction.

For $N = 1$, the claim is trivial as Bianca can guarantee her score to be non negative.

Now assume that the claim (score of Bianca can be at least the sum of weights of biscuits on the even position with respect to the top of the stack) was proven for stacks of size $M = 1...N-1$. When playing on a stack of biscuits of size N with the weights $W_0, W_1, ..., W_{N-1}$, Bianca will do the following strategy: If Aurora chooses $X = 1$, then Bianca chooses $Y = 0$ and eats the biscuits of weights at least $W_0 + (W_2 + W_4 + ...) \geq W_1 + W_3 + W_5 + ...$, which proves the claim. Otherwise, Bianca chooses $Y = 1$ and eats the biscuits of weights at least $W_1 + (W_3 + W_5 + ...)$, once again verifying the claim.

On the other hand, Aurora can ensure Bianca eats biscuits with total weight at most $W_1 + W_3 + W_5 + ...$ She can achive that by always choosing $X = 0$. That guarantees that in Kth turn (we number turns starting from zero) Bianca eats a biscuit of weight at most W_{2K+1}, as Aurora herself eats the biggest remaining biscuit in each turn.

Therefore the score we need to find is just the sum of the biscuits on the even positions with respect to the top of the stack.

We can easily find this sum in $O(N)$ for the initial stack, and update it after each of Q updates in $O(1)$, leading to time complexity $O(N + Q)$.

Subtask 4: $N, Q \leq 100$

We can notice that the final score of Bianca after some turns depends only on the sum of the weights of all the biscuts she previously ate, and on the weights of the biscuits that remain in the stack (they form a suffix on the original array).

We can see the repetitive subproblem structure and therefore it is natural to try to introduce dynamic programming.

Define $dp[i]$ as the score Bianca will get if she and Aurora played the game on the stack of cookies with weights $W_i, W_{i+1}, \ldots W_{N-1}$.

Additionally, we will define $dp[N] = 0$, because playing the game on the empty array results in the score of zero.

We also know that $dp[N-1] = 0$, as Aurora can chose $X = 0$, forcing Bianca to chose a bigger value of Y and getting to eat the cookie at index $N-1$.

Now we will start computing the values of dp in decreasing order of indices.

Assume we have already computed $dp[i+1], dp[i+2], \ldots, dp[N]$ and are now computing $dp[i]$.

We now want to iterate on the possible values of X and Y to find the answer using the precomputed dp values.

First, assume that Aurora has already picked the value for X. Let us try to find the value Y that Bianca should pick.

When Bianca picks a particular Y, she will eat the biscuit at position $i+Y$, and then the game will continue on stack of biscuits with weights $W_{i+Y+1}, W_{i+Y+2}, \ldots, W_{N-1}$. So her score will be $W_{i+Y} + dp[i+Y+1]$.

Among all the values of Y Bianca can pick, she will pick the one that maximizes the value of $W_{i+Y} + dp[i+Y+1]$, under the constraint $0 \leq Y \leq N-i-1$ and $Y \neq X$.

Define a function $f(X)$ as the score Bianca gets under optimal gameplay, if Aurora picks this X. We know that

$$f(X) = \max_{0 \leq Y \leq N-i-1, Y \neq X} W_{i+Y} + dp[i+Y+1].$$

Among all possible choices for X, Aurora will pick the one that minimizes $f(X)$.

More formally, the $dp[i]$ can be calculated as

$$dp[i] = \min_X(\max_{Y \neq X}(W_{i+Y} + dp[i+Y+1])).$$

In each iteration,we can compute $dp[i]$ by iterating over all the possible values of X that Aurora can pick and compute $f(X)$ by iterating over all the possible values Y that Bianca can pick and finding the largest value of $W_{i+Y} + dp[i+Y+1]$ under the constratints. Then the smallest of the $f(X)$ values will be the value of $dp[i]$. For a single stack, this takes $O(N^3)$ time, and we will compute this from scratch for each update, so the total time complexity is $O(N^3Q)$, which is fast enough for $N, Q \leq 100$.

Subtask 5: $N \leq 10,000$, $Q \leq 100$

We will optimize the solution of the previous subtask. The number of states is small (there are N values of $dp[i]$, so let us focus on optimizing the process of computation of $dp[i]$.

Intuitively, if Bianca picks a large value Y, it will cause her to give up many biscuits to Aurora, which makes some sense if she is trying to get to a very big biscuit, but since the weights of the biscuits are pretty small, it does not make sense for her to give up too many biscuits by picking a very large Y.

Take the following example. Stack consists of 200 biscuits of weight 1, followed by a single biscuit of weight 50. Should Bianca set Y such that she takes the biscuit of weight 50?

If she does, her score will be 50. But if she instead always chooses Y as small as possible, she will end up eating at least half of the biscuits (it is the same strategy as in subtask 1 but applied to a general array) and then her score will be at least 100, which is much better.

Notice that this would not hold if the weight of the big biscuit would be, say 1000. The reason why this is happening is that the weights of the biscuits are pretty small. Let M be the maximum weight a biscuit can have.

If we could somehow prove that Bianca will always pick a small value Y, the number of possible $f(X)$ and corresponding Y we need to iterate through would decrease and therefore the dynamic programming would be faster.

More precisely, let us assume that biggest value of Y Bianka might pick in such scenario is Y_{max}. Then note that if Aurora were to pick any X such that $X > Y_{max}$, it is the same as if she allowed Bianca to pick any number, as Bianca would have not picked X anyway. Therefore there is always an optimal strategy where Aurora picks X such that $X \leq Y_{max}$.

Therefore, if we can somehow get a good bound on Y_{max}, we can reduce the number of X values we check.

Suppose we are computing $dp[i]$ and we are wondering what is the biggest index j of a biscuit Bianca might choose to eat next, by picking $Y = j - i$.

If there would be $j > i + 2 \cdot M$ such that $Y = j - i$ is the optimal choice for Bianca on this suffix, then her score in this case would be $W_j + dp[j + 1]$. But she can do another strategy (somewhat similar to subtask 1 and the example above): while she can pick $Y \neq X$ such that the remaining suffix will be of length at least $N - j + 1$, she will pick the smallest possible Y (so if $X = 0$, then $Y = 1$, otherwise $Y = 0$) and keep playing the game, collecting at least half rounded down of the biscuits in $i...j - 2$, in other words, eating biscuits with total weight at least $\lfloor \frac{j-i-1}{2} \rfloor$.

Once this is no longer possible, in response to Aurora's choice of X, Bianca will choose Y such that she either gets to eat the biscuit at position j or $j - 1$

in this turn. If she can choose the former, then with this strategy she clearly achieves better score using $Y = 0$ or $Y = 1$ than if she originally chose $Y = j - i$, which is a contradiction. Otherwise, her score with this strategy is at least $\lfloor \frac{j-i-1}{2} \rfloor + W_{j-1} + dp[j]$.

However we can prove that $dp[i]$ is non increasing (for a suffix starting with i, Bianca can always copy her strategy for suffix of length $i+1$ and pretend the ith biscuit does not exist and get at least as good score as $dp[i+1]$), therefore the value above is at least $\lfloor \frac{j-i-1}{2} \rfloor + W_{j-1} + dp[j+1] \geq M + 1 + dp[j+1] > W[j] + dp[j+1]$.

which is a contradiction once again, proving that the optimal Y is always at most $2 \cdot M$. This gives us an upper bound of $Y_{max} \leq 2 \cdot M$.

This gives us a method to compute the transitions in $O(M^2)$. We can further optimize this by noticing that computing $f(X)$ and $f(X+1)$ is very similar and reusing the information computed cleverly. For example, we can do the following - we will introduce two new arrays, $pref$ and suf. We define $pref[j]$ as

$$pref[j] = \max_{i+1 \leq k \leq j}(W[k+1] + dp[k+2]).$$

Similarly, we define $suf[j]$ as

$$suf[j] = \max_{j \leq k \leq i+Y_{max}}(W[k+1] + dp[k+2]).$$

We can compute these arrays in $O(M)$ just before computing $f(X)$ values for $dp[i]$. Then for any X, we can find $f(X) = \max(perf[X-1], suf[X+1])$ in O(1) time.

With this optimization the time complexity is $O(N^2Q)$, which is still not enough to pass this subtask.

Then the number of transitions in the dynamic programming for previous subtask goes down to $O(M)$ and the overall complexity can be brought down to $O(NQM)$.

Subtask 6: $N, Q \leq 10,000$

There is also another way to make the dynamic programming from subtask 4 faster.

Consider the stack of biscuits of weights $W_i, ..., W_{N-1}$. Depending on which Y Bianca picks, the score will be one of the values $W_j + dp[j+1]$ for $i \leq j \leq N$. To account for the possiblity of Bianca not eating any biscuit, we define additionally $W_N = 0$ and $dp[N+1] = 0$.

Aurora can block at most one of these possibilities by choosing proper X and then Bianca will pick the highest scoring remaining option. We can formalize this as

$$dp[i] = \text{secondmax}_{i \leq j \leq N}(W_j + dp[j+1])$$

Here secondmax(S) is the second biggest element of multiset S (or 0 if the multiset has less than two elements).

Using this formula, we can compute $dp[i]$ in $O(N)$. But we need to do it faster! To do so, notice that when moving from suffix $i+1$ to suffix i, the multisets passed to the secondmax stays the same, except one new possibility gets added, corresponding to Bianca choosing i, resulting in the score $W_i + dp[i+1]$. But recalculating secondmax for a multiset with one more new value is actually easy and can be done in $O(1)$ - we just need to mantain the two largest values in the multiset and update them with the new value.

This gives an $O(N)$ computation of dp for the original array and after each update, leading to a total time complexity of $O(NQ)$ which is fast enough for $N, Q \leq 10,000$ with a proper implementation.

Full solution

We will explore the structure of the $O(N)$ per stack dynamic programming to be able to efficiently process updates. First of all, define $W_N = 0$ in addition to $dp_N = 0$ so that we can state

$$dp[i] = \text{secondmax}(W_j + dp[j+1] \mid i \leq j \leq N).$$

without the 0, because that is just the $j = N$ case.

The most complex part of the formula for the computation of the dynamic programming is the secondmax formula, so we will focus on simplifying it. To achieve this, we start by introducing a new dynamic programming $t[i]$ such that $t[i] = W_i + dp[i+1]$. Then we have that

$$t[i] = dp[i+1] + W_i = \text{secondmax}(W_j + dp[j+1] \mid i+1 \leq j \leq N) + W_i$$

which we finally rewrite using the $b[i]$ definition as

$$t[i] = \text{secondmax}(t[j] \mid i+1 \leq j \leq N) + W_i$$

which is simpler since now we have the single dp (actually t) values in the secondmax function, not the sums.

But wait, we need $dp[0]$, how do we get that? We will do a little trick and say that we have $W_{-1} = 0$ and extend the computation to obtain $b[-1]$ as well:

$$t[-1] = \text{secondmax}(t[j] \mid 0 \leq j \leq N) + W_{-1} = dp[0]$$

Notice that when we are computing value of $t[i]$, we do not need the full information about the values $t[i+1], \ldots, t_{N-1}$, but rather, we are interested in

two special values, let us call them $A[i+1] = \max\{t[i+1], t[i+2], \dots t[N]\}$ and $B[i+1] = \text{secondmax}\{t[i+1], t[i+2], \dots t[N]\}$, where clearly $A[i+1] \geq B[i+1]$. These are closely related to $t[i]$ by $t[i] = B[i+1] + W_i$.

Using $A[i+1]$ and $B[i+1]$ we are now interested in computing $A[i]$ and $B[i]$. They are the two greatest values from the multiset $\{t[i], t[i+1], t[i+2], \dots t[N]\}$ which is the same as the two highest values in the multiset $\{B[i+1]+W_i, A[i], B[i]\}$, but since $B[i] \leq B[i+1]+W_i$ and $B[i] \leq A[i]$, these are just $A[i]$ and $B[i+1]+W[i]$ (possibly swapped, we do not know which one is bigger).

Well this does not seem too helpful so far - we are now keeping track of two values (A and B) instead of one (the dynamic programming). Let us try to simplify this to be able to keep track of just one value. As a thought experiment, what would the values of $A[0]$ and $B[0]$ be if at the values $A[N], B[N]$ were not both 0, but both 100. Let us denote such arrays as A' and B'. We can notice that $A'[i] = A[i] + 100$ and $B'[i] = B[i] + 100$ will hold not just for $i = N$, but also for the rest of the indices. Notice that whenever $B[i] + W_i > A[i]$, it will also hold that $B'[i] + W_i > A'[i]$

Inspired by that observation, instead of keeping track of the values of $A[i]$ and $B[i]$, we will keep track of the values $A[i] + B[i]$ and $A[i] - B[i]$. We will see that value of both can be computed separately. Computing final value of $A[i] + B[i]$ is easy: from the nature of updating $(A[i], B[i])$ using $(A[i+1], B[i+1])$ we can see that $A[i] + B[i] = A[i+1] + B[i+1] + W[i]$, therefore $A[i] + B[i] = \sum_{j=i}^{N-1} W_j$.

Meanwhile, the value of $A[i] - B[i]$ can be computed in a for loop. We will create a new array of values $D[i] = |A[i] - B[i]|$. Even with the absolute value, this still allows us to reconstruct $A[i]$ and $B[i]$ uniquely using the difference and sum, because we know that $A[i] \geq B[i]$. We start by setting $D[N] = 0$ and then iterate through the array from index $N-1$ to index 0. When we are at index i, we can see that

$$D[i] = |A[i] - B[i]| = |A[i+1] - B[i+1] - W[i]| = |D[i+1] - W[i]|$$

so we can calculate $D[i]$ values without explictly calculating A, B or anything else.

However, this computation is still $O(N)$ per query, so we need something faster.

Updates are local, but they force us to recompute the whole prefix of the array $D[i]$, starting with the point of change and going backwards all the way to 0.

What if all updates happen after, let us say, index K? We would like to maintain some sort of prefix structure that will let us handle updates without having to iterate over the first K indices each time. Notice that value of $D[i]$ depends only on value of $D[i+1]$ and W_i. We can extend this observation - the value $D[i]$ depends only on the values W_i, W_{i+1} and $D[i+2]$ and so on. And so we can see that, after each update, the new value of $D[0]$ can be determined from the new

value of $D[K]$ and the values W_i for $0 \leq i \leq K$, but the only thing that changes is $D[K]$.

Notice the constraint $W_i \leq 50$: it also implies that $D[i] \leq M$ (we can prove this by induction, as $D[i] \leq M$). Therefore, $D[K]$ can take at most $M + 1$ different values.

The idea is to precompute value of $D[0]$ for all possible values of $D[K]$. As there are at most $M + 1$ possible values $D[K]$ can take, this can be done in $O(KM)$. Then, when we need to compute the difference after some update, we can compute $D[K]$ and then look up the corresponding precomputed value to find $D[0]$.

But in the original problem, the updates can happen anywhere in the array and therefore using a simple prefix sum like this is not sufficient. Instead, we will use a segment tree based on a similar idea.

We define function $f_{L,R}(x)$ (for $0 \leq L < R \leq N - 1$) as follows. Assuming that value $D[R + 1] = x$, what will be the value of $D[L]$? If for some j such that $L \leq j \leq R$ we know values of $f_{L,j}$ and $f_{j,R}$ for all $x \in [0, ..., M]$, we can calculate $f_{L,R}$ for all $x \in [0, ..., M]$ by observing that $f_{L,R}(x) = f_{L,j}(f_{j,R}(x))$.

When we build a segment tree over the array, we will have the node responsible for interval $[L, R]$ maintain the value of function $f_{L,R}$ for all $x \in [0, ..., M]$.

Computing the leaf nodes, $f_{L,L}$, is easy when we know W_L: we simply get $f_{L,L}(x) = |x - W_L|$ and we can do this in $O(M)$ whenever needed.

For the other nodes, we can compute $f_{L,R}$ using the precomputed f for the two children - this can be simply done using the formula above, in $O(M)$.

To find the optimal score, we need to maintain $S = \sum_{i=0}^{N-1} W_i$ and find $D[0] = f_{0,N-1}(0)$. Then we calculate $B[0] = \frac{S-D[0]}{2}$ and observe that $dp[0] = t[-1] = B[0]$, so we will print $\frac{S-D[0]}{2}$ in the beginning and after processing each update.

This segment tree can be set up in $O(N \cdot M)$ time. Every update can be processed in $O(M \log N)$ time and the score can be foud in $O(1)$ time. The total time complexity will be $O(N \cdot M + Q \cdot M \cdot \log N)$, which is fast enough for 100 points.