December 2019 Plat Review

Ray Bai and Danny Mittal

January 2020

1 Greedy Pie Eaters

Problem Statement: $1 \leq N \leq 300, 1 \leq M \leq \frac{N(N+1)}{2}$

You are given M intervals, each of which is a triple of integers (l, r, w) satisfying $1 \leq l \leq r \leq n$ and $1 \leq w \leq 10^9$, indicating that the interval covers all of the integers from l to r inclusive. A set of intervals is valid if the intervals in the set can be ordered such that no interval in the set is fully contained in the union of the intervals before it in the order. Find the maximum possible total weight of a valid set.

Solution: Dynamic Programming

We will create a DP array indexed by l and r. $dp[l][r]$ will be the maximum weight of any valid set all of whose intervals lie between l and r. The transition comes from the idea that any valid set will have an interval that comes last in its order, and this interval should not be fully contained in the union of all the other intervals in the valid set, which means that it should contain at least one integer that is not contained by the rest of the set.

Let this integer be k. To update $dp[l][r]$, we will loop over all k such that $l \leq k \leq r$, and attempt to create a new valid set from the optimal valid set between l and $k-1$, the optimal valid set between $k+1$ and r, and an interval between l and r that contains k - this is important so that that interval isn't fully covered by the other two valid sets. We can get the optimal weightings of those two valid sets from $dp[l][k-1]$ and $dp[k+1][r]$ (remember that we don't need the optimal set, just the total weighting of the optimal). Getting the weighting of the new interval that we're adding will take a bit more work.

There are a couple ways to do this, but the simplest approach is another DP. Let $dpMax[l][k][r]$ be the maximum weighting of any interval lying between l and r that contains k - you'll notice that this is exactly what we need. For every interval (l, r, w), loop through all k such that $l \leq k \leq r$ and set $dpMax[l][k][r]$ to $max(dpMax[l][k][r], w)$. Then expand all of these placements outward using DP: more precisely, for all l, k, r, set $dp[l][k][r]$ to $max(dp[l][k][r], dp[l+1][k][r], dp[l][k][r-])$, capturing the weighting of the optimal interval inside. That concludes this DP.

Returning to the original DP, our transition is now clear: to update $dp[l][r]$, loop through all k such that $l \leq k \leq r$ and set $dp[l][r]$ to $max(dp[l][r], dp[l][k-1] + dpMax[l][k][r] + dp[k+1][r])$. We now only need an additional transition to account for combining two valid sets that don't intersect without adding a new interval, which is similar: loop through all k such that $l \leq k < r$ and set $dp[l][r]$ to $max(dp[l][r], dp[l][k] + dp[k+1][r])$. This will also account for expanding one valid set outward, as the other "valid set" will simply contribute a weighting of 0. Now our DP is complete, and the answer is contained in $dp[1][n]$.

The first DP has N^2 values each of which is computed in an $O(N)$ transition, giving a complexity of $O(N^3)$, and the auxiliary DP has N^3 values each of which is computed in an $O(1)$ transition, giving a complexity of $O(N^3)$, so that the overall complexity is $O(N^3)$, which is fast enough given that $N \leq 300$.

2 Bessie's Snow Cow

Problem Statement: $1 \leq N, Q \leq 100000$

You are given a rooted tree with N vertices. A node in this tree can be splashed with a color c where c is an integer satisfying $1 \leq c \leq 100000$, which will also cause all nodes in that node's subtree to be splashed with c.

You are then given Q queries, which can take one of two types:

Type 1 queries consist of two integers x, c satisfying $1 \le x \le N$ and $1 \le c \le 100000$, indicating that the node x has been splashed with c.

Type 2 queries consist of a single integer x satisfying $1 \le x \le N$, indicating that you should output the sum over all nodes y in the subtree of y (including x itself) of the amount of distinct colors with which y has been splashed.

Solution:

First, we flatten the tree. What this means is that we find the pre-order traversal of the tree (using DFS) and assign each node x in the tree an interval (l_x, r_x) $(1 \le l_x \le r_x \le N)$ indicating that l_x is the index of x in the pre-order traversal, and r_x is the rightmost index in the pre-order traversal of a node in the subtree of x. Because of how pre-order traversal works, the nodes between l_x and r_x in the pre-order traversal will consist precisely of the subtree of x. This is useful, because to query the subtree of x, we can instead use a linear data structure such as a segment tree and query the interval (l_x, r_x).

We now do exactly that: create a lazy propagation sum segment tree corresponding to the nodes in the tree according to the pre-order traversal. The value at a single index will represent the amount of distinct colors with which the corresponding node has been splashed. Therefore, to answer type 2 queries for the node x, we simply query the lazy segment tree over the interval (l_x, r_x) and output the result.

All that's left now is to handle type 1 queries. For each possible color c (a.k.a. all integers from 1 to 100000) create a sorted set of integers (in Java/Kotlin, use TreeSet; in C++, use std::set). This set will contain l_x for all nodes x such that x has been splashed with c and no ancestor of x has been splashed with c.

Now, given a type 1 query x, c, we first want to determine if x has already been splashed, because in that case we shouldn't do anything. We can do this by querying c's sorted set in the following manner: if x has been splashed with c, then one of x's ancestors (possibly x itself) must be in the set. Because of pre-order traversal, any ancestor y of x would have to have l_y be the greatest integer in the set less than or equal to l_x. Therefore, we can simply query the *sorted* set to see what the greatest integer l_y less than or equal to l_x is, and then check if y is an ancestor of x (by checking if $l_y \le l_x \le r_y$). If y is an ancestor of x, then we do nothing and move on.

If not, then this query actually matters. We now have to update the segment tree, but before doing that, we need to un-update the segment tree for any descendants of x that were previously splashed with c. We can again do this by looking at c's sorted set: the descendants y of x must have l_y immediately follow l_x in the set. Therefore, we can simply continually query for the least integer l_y greater than l_x in the set, and while y is a descendant of x, remove l_y from the set and do a range update on the segment tree on the range (l_y, r_y) with a change of -1. Finally, add l_x to the set and do a range update on the segment tree on the range (l_x, r_x) with a change of 1.

Every type 1 query incurs at most one check for an ancestor of x, at most one addition of x into a set and update on the segment tree, and at most one removal from the set and un-update on the segment tree later on. This amount of operations is $O(1)$, and each operation is $O(\log N)$, so that each type 1 query incurs an overall complexity of $O(\log N)$.

Every type 2 query incurs one query on the segment tree, which is $O(\log N)$.

All queries are $O(\log N)$, yielding a complexity of $O(Q \log N)$ to process all of the queries. The DFS to determine (l_x, r_x) for all x is $O(N)$, and the initialization of the segment tree is $O(N)$, making the overall complexity $O(N + Q \log N)$.

3 Tree Depth

Problem Statement: $1 \le N \le 300, 1 \le K \le \frac{N(N-1)}{2}$

Consider an array p of length N, consisting of all of the integers from 1 to N in some order. Take as an example $p = \{4, 7, 3, 5, 1, 2, 6\}$.

A pair of indices (j, k) is an inversion if $1 \le j < k \le n$ and $p[j] > p[k]$. Conceptually, an inversion is when two elements in the array are in the wrong order. In the given example, $(2, 4)$ is an inversion because $p[2] = 7$ comes before $p[4] = 5$, but $7 > 5$.

The Cartesian tree of p is a tree constructed in the following manner: Find the index j such that $p[j]$ is the minimum value in the array. Set this as the root of the tree. Then find the Cartesian tree of the

subarray of p coming before j, and set that as the left subtree of j in the tree. Similarly, find the Cartesian tree of the subarray of p coming after j, and set that as the right subtree. The Cartesian tree of the given example is as follows:

```
      5
  3       6
1   4       7
 2
```

Note that the nodes in the tree are the indices, not the values themselves.

For an index j (so $1 \leq j \leq n$), define $d_p(j)$ as the depth of j in the Cartesian tree of p. Define S as the set of all p with exactly K inversions. For each j, find $\sum_{p \in S} d_p(j)$.

Solution: To solve this problem, we will need to make use of generating functions. A generating function is essentially a polynomial in which a term of the form x^a is seen as representing an object with a certain value equal to a, so that a term of the form cx^a is seen as represent c objects, all of which have that certain value equal to a. In our case, the objects will be the arrays p and the certain value will be the amount of inversions in the array. You can see where this is headed: we will create some kind of generating function and then look at the x^K term.

Consider constructing an array p. We'll begin with the index j. We're not going to actually say what value we're putting at index j, just that there is something there. Next, we'll move on to the index $j+1$. Like before, we won't actually say what value we're putting at index $j+1$, but what we will say is what that values is relative to $p[j]$. $p[j+1]$ can either be greater than or less than $p[j]$, and we'll choose which one it is. Notice something important: if we say that $p[j+1] > p[j]$, then $(j, j+1)$ is not an inversion, but if we say that $p[j+1] < p[j]$, then $(j, j+1)$ *is* an inversion. Therefore, at this step, we can introduce either 0 or 1 inversions.

When we're at index $j+2$, we make a similar choice: $p[j+2]$ will either be greater than the previous two values, in between them, or less than both, which will introduce either 0, 1, or 2 inversions. In the next step, we can introduce between 0 and 3 inversions, and so on. Eventually we will reach the end of the array and need start defining the values before j, starting with the index $j-1$, but this doesn't actually make a difference: if we've defined an ordering of q values and are now "choosing" a value that comes before or after all of those values, we can add between 0 and q inversions. To summarize: in the first step we add 0 inversions, in the second we add 0 to 1 inversions, in the third step we add 0 to 2 inversions ... up until the final step, where we add 0 to $N-1$ inversions.

What we can now do is make a generating function out of this: we start with the term x^0, representing an array that has 0 inversions. The next step is represented by the multiplying factor (x^0+x^1), because for each of our current possibilities for the array, we get a new possible array with 0 additional inversions and a new possible array with 1 additional inversion. So if we were to stop it right there and have an array of length 2, our generating function would be $x^0(x^0+x^1) = x^0+x^1$, which makes sense - there are two possibilities for the array, one of which, $\{1,2\}$, has 0 inversions, and one of which, $\{2,1\}$, has 1 inversion. To get the arrays of length 3, we multiply this by $(x^0+x^1+x^2)$, which gives $(x^0+x^1)(x^0+x^1+x^2) = x^0+2x^1+2x^2+x^3$, which is correct: The 1 array with 0 inversions is $\{1,2,3\}$. The 2 arrays with 1 inversion are $\{1,3,2\}$ and $\{2,1,3\}$. The 2 arrays with 2 inversions are $\{2,3,1\}$ and $\{3,1,2\}$. The 1 array with 3 inversions is $\{3,2,1\}$.

Of course, we want a length N array, so we continue up until the factor $(x^0+x^1+x^2+\cdots+x^{N-1})$, so that our overall generating function becomes the following product:

$$x^0(x^0+x^1)(x^0+x^1+x^2)(x^0+x^1+x^2+x^3)\cdots(x^0+x^1+x^2+\cdots+x^{N-1})$$

This isn't actually the final generating function that we'll need, but we will need to compute this generating function as part of our algorithm, so let's do that now. The highest power term that can be in our generating function is $x^{\frac{N(N+1)}{2}}$, so we'll represent the generating function as a 0-indexed array of length $\frac{N(N+1)}{2}+1$) named *poly*. We start with $poly[0] = 1$, and $poly[j] = 0$ for all $j \neq 0$. In the a-th step, we multiply by the factor $(x^0+x^1+x^2+\cdots+x^{a-1})$. What this means is that the coefficient of the x^j term in our new generating function is the sum of the coefficients of the $x^j, x^{j-1}, x^{j-2}, \ldots x^{j-(a-1)}$ terms in our old generating function. Therefore, we can use prefix sums of the coefficients of the old generating function to efficiently calculate the new generating function. Specifically, in the a-th step, we first perform prefix

sums so that $sum[j]$ is $poly[0] + \cdots + poly[j]$. Then, we simply set $poly[j]$ to $sum[j] - sum[j-a]$. Since the generating function has $O(N^2)$ terms, both of these steps are $O(N^2)$, and since there are N factors to multiply, the overall complexity of calculating this generating function is $O(N^3)$.

Now, how do we use this to calculate the sum of $d_p(j)$? First consider again a single array p. $d_p(j)$, the depth of j in the Cartesian tree of p, is equal to the amount of ancestors of j in the Cartesian tree. Therefore, we can compute $d_p(j)$ by counting the amount of indices k such that k is an ancestor of j in the Cartesian tree. However, k being an ancestor of j is equivalent to $p[k]$ being the minimum value of all of the values at the indices between j and k; i.e. if $j < k$, then $p[k] = min(p[j], p[j+1], \ldots, p[k-1], p[k])$, otherwise $p[k] = min(p[k], p[k+1], \ldots, p[j-1], p[j])$.

Now that we have this condition that is equivalent to k being an ancestor of j, what we can now do is, for each k, compute the amount of $p \in S$ that have k as an ancestor of j by computing a generating function of p that have k as an ancestor of j, extracting the coefficient of x^K, and summing those results to get $d_p(j)$. For simplicity, assume that $k > j$ - the case of $k < j$ is different only in an unimportant way. When we were constructing the generating function above, since we started by "choosing" $p[j]$, we chose $p[k]$ in the $k - j + 1$th step, meaning that we could add between 0 and $k - j$ inversions. However, note that, since we started from $p[j]$, the elements that we had already chosen up to that step - a.k.a. the elements with respect to which we would order $p[k]$ - are precisely the elements that $p[k]$ must be less than to be an ancestor of j - the elements between indices j and k (excluding k itself, which is unimportant). This means that the modification that we have to make to the generating function in order to enforce that k be an ancestor of j is to always make $p[k]$ less than the elements that had already been chosen, which is equivalent to always adding $k - j$ inversions at the $k - j + 1$th step. This means that instead of multiplying by the factor $(x^0 + \cdots x^{k-j})$, we multiply by the factor x^{k-j}.

Therefore, in order to get the generating function that we need, we just need to divide our original generating function by $(x^0 + \cdots x^{k-j})$ and multiply it by x^{k-j}, which can be done in $O(N^2)$ using a similar prefix sums approach to before. This initially looks like a problem: we're doing this for each j and for each k, which would make this step overall $O(N^4)$; however, the generating function we calculate only depends on $k - j$, which means that we only need to actually calculate $N - 1$ of these generating functions and then reuse them, making this step only $O(N^3)$.

This is all identical in the case of $k < j$, except that we look at $j - k$ instead of $k - j$ (think about choosing the values below index j before you choose the values above index j), and instead of dividing our original generating function by $(x^0 + \cdots x^{k-j})$ and multiplying it by x^{k-j}, we divide our original generating function by $(x^0 + \cdots x^{k-j})$ and multiply it by x^0 (because we must now add 0 inversions on the $j - k + 1$-th step).

Finally, if we store the coefficient of x^K in the generating function for $d = k - j$ in $res_1[d]$, and store the coefficient of x^K in the generating function for $d = j - k$ in $res_2[d]$, then we get $answer[j] = \sum_{k=1}^{j-1} res_2[j-k] + \sum_{k=j+1}^{N} res_1[k-j]$. The most expensive steps we used were $O(N^3)$, so the overall complexity is $O(N^3)$, which is fine for $N \leq 300$.