B503 Homework 4

Due date: November 19 Lecturer: Qin Zhang

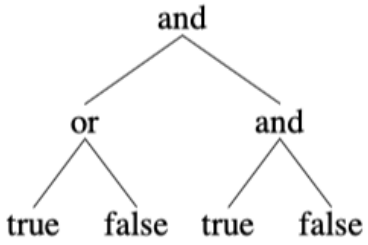
Instruction: Please submit your homework via Canvas before the deadline. Please add references to ALL the resources you have used for completing the assignment. You are allowed to discuss the assignment with other students, and if you do so, please list their names in the submission. Remember, you have to write all answers by yourself even if you have discussed with other students.

Unless mentioned otherwise, for each algorithm that you design, please **give a proof of its correctness**. If the question asks you to come up with an algorithm of a specified runtime complexity, please **give the running time analysis**. If the task doesn't specify the runtime complexity, you are allowed to present an algorithm of any time complexity. However, you still need to prove the correctness of your algorithm and analyze its running time.

Total points: 37

Late Policy: No extensions or late homeworks will be granted, unless a request is made to the course instructor before due date and written documents are provided to support the reason for late submission.

Problem 1 (5 points). At Aperture Bakeries, every cake comes with a binary boolean-valued tree indicating whether or not it is available. Each leaf in the tree has either a *true* or a *false* value. Each of the remaining nodes has exactly two children and is labeled either *and* or *or*; the value is the result of recursively applying the operator to the values of the children. One example is the following tree:



If the root of a tree evaluates to *false*, like the one above, the cake is a lie and you cannot have it. Any *true* cake is free for the taking. You may modify a tree to make it *true*; the only thing you can do to change a tree is to turn a *false* leaf into a *true* leaf, or vice versa.

This costs \$1 for each leaf you change. You can't alter the operators or the structure of the tree.

Describe an efficient $O(n)$ algorithm to determine the minimum cost of a cake whose tree has n nodes. Prove the correctness and analyze running time of your algorithm.

Solution: We can use dynamic programming to determine the cost. Define $OPT(v)$ as the minimum cost to make node v to be evaluated *true*. Our desired output is $OPT(root)$.

Boundary condition: If v is a leaf with value *true*, we make $OPT(v) = 0$. If v is a leaf with value *false*, we make $OPT(v) = 1$.

We have the following relationship:

$$OPT(v) = \begin{cases} \sum_{u \text{ as } v\text{'s children}} OPT(u) & \text{if v has operator } \textit{and} \\ \min_{u \text{ as } v\text{'s children}} OPT(u) & \text{if v has operator } \textit{or} \end{cases}$$

because both children need to be *true* in an *and* operator node and only one child to be *true* in an *or* operator node.

Since there are $O(n)$ subproblems, and the cost of each subproblem is constant, the total running time is $O(n)$.

Grading Scheme: 1 point for the base case, 2 points for the equation, 1 point for the implementation, 1 point for the running time.

Problem 2 (5 points). Consider two vertices, s and t, in a *directed acyclic graph* $G = (V, E)$. Give an efficient $O(|V| + |E|)$ algorithm to determine whether the number of paths in G from s to t is odd or even.

Solution: Let $n = |V|$ and $m = |E|$.

Define $OPT(v)$ as the number of paths in G from v to t. Our desired output is the evenness or oddness of $OPT(s)$.

As the graph is directed acyclic, we could produce a topological ordering $v_1, v_2, \ldots, v_n$ of the vertices in G. If t is in front of s, then there is no path between them. Otherwise, without loss of generality, let $v_1 = s$ and $v_n = t$ as we only care about nodes in $s \to t$ range.

Boundary condition: $OPT(t) = 1$.

We have the following relationship:

$$OPT(v) = \sum_{(u,v) \in E} OPT(u)$$

We compute the value of $OPT(v)$ from t to s as in the reversed topological order.

We need to solve $O(n)$ problem and each take $O(\text{degree of node})$ time, making the total running time $O(m)$. The topological sort takes $O(m + n)$ time. So the total running time is $O(m + n)$.

Grading Scheme: 1 point for the base case, 2 points for the equation, 1 point for the implementation, 1 point for the running time.

Problem 3 (5 points). You are given a sequence of integers. You can choose to remove a *single* number from the list, or not remove any at all. Your goal is to maximize the length

of the strictly increasing contiguous subsequence in the resulting array.

Create an algorithm to find the maximum length of such a sequence with a linear time complexity.

N.B. A contiguous subsequence $b_1, \ldots, b_m$ of sequence $a_1, \ldots, a_n$ is such a subsequence that its elements are next to each other in the original sequence, i.e. $b_i = a_{l-1+i}$ for some l and all $i \in [1..m]$.

Solution.

Let's define $OPT_L(l)$ and $OPT_R(r)$. $OPT_L(l)$ is the longest strictly increasing subsequence that ends at l. Symmetrically, OPT_R is the longest strictly inrceasing subsequence that starts at r.

$$OPT_L(l) = \begin{cases} OPT_L(l-1) + 1, & l > 1 \wedge a_{l-1} < a_l \\ 1, & \text{otherwise} \end{cases}$$

The first case is when a_l is greater than the previous element and thus the desired subsequence is just a continuation of the subsequence that ends at $l-1$. $OPT_R(r)$ is defined symmetrically.

The result x will be the maximum of two cases: when we don't remove a letter (x_0) and when we do (x_1).

Naturally, $x_0 = \max_{l \in [1..n]} OPT_L(l) = \max_{r \in [1..n]} OPT_R(r)$.

To find x_1, suppose that we remove the ith number, which might form new strictly inrceasing subsequence. Let's find the maximum length of all such subsequences. All of them must include some numbers to the left of i and to the right of i, otherwise they are not new. That is only possible if $a_{i-1} < a_{i+1}$. If it is, then the longest new subsequence would be formed with the longest prefix ending at $i-1$ and the longest suffix starting at $i+1$. In other words, $OPT_L(i-1) + OPT_R(i+1)$.

Thus, we can compute x_1 as the maximum of all possible removals, which are $O(n)$.

We have $O(n)$ subproblems, each one takes $O(1)$ time to solve. Thus, the total running time is $O(n)$.

Grading Scheme: 1 point for the base case, 2 points for the equation, 1 point for the implementation, 1 point for the running time.

Problem 4 (5 points).

During an all too common electricity outage, Jane and Joe decide to pass the time by playing a game. They use a set of n cards, each labeled with a distinct number from 1 to n. They shuffle the cards and place them in a line. The resulting order of labels is denoted as $a_1, a_2, \ldots, a_n$.

To start the game, Jane and Joe take an old Monopoly figure (the iron) and put it on one of the cards. They take turns moving the iron, with Jane taking the first turn. The game has two rules for moving the iron:

1. When the iron is on the ith card, it can only be moved forward, to a card with a greater label, i.e. to a card j where $a_j > a_i$.

2. The iron can only be moved by the number of cards divisible by the label on the current card, i.e. $(j - i) \bmod a_i = 0$.

If a player can't make a move, they lose the game.

Create an $O(n \log n)$ algorithm that, for each position $i \in [1..n]$, determines who will win the game starting with this position: Jane or Joe.

Solution.

Let $OPT(i)$ be true if the 2nd player (Joe) wins when starting at the ith card. Then,

$$OPT(i) = \bigvee_{q=0}^{\lfloor \frac{n-i \bmod a_i}{a_i} \rfloor} a_{qa_i+i \bmod a_i} > a_i \wedge OPT(qa_i + i \bmod a_i).$$

First, let's note that there is a hierarchy of subproblems induced by the label size. That is, to compute $OPT(i)$ we only need to have computed $OPT(j)$ for j such that $a_j > a_i$. That means we have a *valid separation into subproblems*.

The update stems from the fact that we win iff we are able to make such a move that we end up in a position where the second player. If we make such a move, we become the second player and are guaranteed a win. If we can't make such a move, all positions lead to the win of the first player, i.e. the opposite party.

Thus, we are iterating over all positions $j \coloneqq qa_i + i \bmod a_i$ where both of the rules are satisfied and see if any of them lead to the win of the second player.

The first rule is satisfied by checking $a_j > a_i$ in each disjunct.

The second rule is satisfied because we only iterate over positions j where it is satisfied. Indeed, if $|j - i| \bmod a_i = 0$, it applies precisely that $\exists q.\, j = qa_i + i \bmod a_i$:

$$\begin{aligned}
&(j - i) \bmod q = 0 \iff \\
\iff &\exists q'.\, j - i = q'a_i \iff \\
\iff &\exists q'.\, j = q'a_i + i \iff \\
\iff &\exists q', q_i.\, j = q'a_i + q_i a_i + i \bmod a_i \iff \\
\iff &\exists q \coloneqq q' + q_i.\, j = qa_i + i \bmod a_i
\end{aligned}$$

The boundaries of q stem from $j \in [1..n]$.

Let's estimate the runtime of updates we do:

$$T = \sum_{i=1}^{n} O(1) \cdot \lfloor \frac{n - i \bmod a_i}{a_i} \rfloor \leq O(1) \cdot \sum_{i=1}^{n} \frac{n}{a_i} = \cdots$$

Remember that $\{a_i \mid i \in [1..n]\} = [1..n]$.

$$\cdots = O(1) \cdot \sum_{a=1}^{n} \frac{n}{a} = O(1) \cdot n \sum_{a=1}^{n} \frac{1}{a} \leq O(1) \cdot n \ln n = O(n \ln n)$$

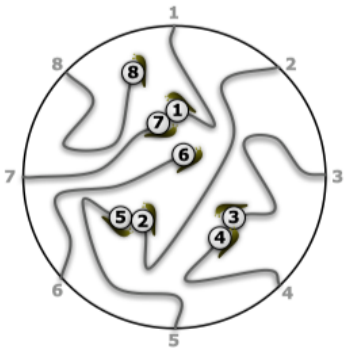
We also need $O(n \ln n)$ to sort the indices to compute $OPT(i)$ in the appropriate order.

Grading Scheme: 2 points for the equation, 0.5 point for the implementation, 2.5 points for the runtime.

Problem 5 (5 points).

Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

Describe and analyze an $O(n^3)$ algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

Solution: As the snails cannot cross a slime trail, if snail i and snail j meet during the race, all snails within the (i, j) range can only meet snails that are also within the range.

So we could define our sub problems in the following way. Define $OPT(i, j)$ the maximum total reward for snails $(i, i + 1, \ldots, j)$. Our desired output is $OPT(1, n)$.

Boundary condition: Self pair is set to be 0 $OPT(i, i) = 0$ and neighboring pair is set to be $OPT(i, i + 1) = M(i, i + 1)$.

Then we have the following relationship:

$$OPT(i,j) = \begin{cases} \max_{i \le k < j} \{OPT(i, k-1) + OPT(k+1, j-1) + M(k, j)\} \\ OPT(i, j-1) \end{cases}$$

The first relationship corresponds to the case where j is matched with some k and the second relationship corresponds to the case j is not matched with any other snail.

We fill the matrix by the increasing order of l which represents the interval length. The boundary condition solves the case where $l = 0$ and $l = 1$. Then for $l = \{2, 3 \ldots\}$, we solve

for all possible $OPT(i, i+l)$ until we reach $OPT(1, n)$.

We have $O(n^2)$ subproblems and it takes $O(n)$ time to solve each of them. The total running time is $O(n^3)$.

Grading Scheme: 1 point for the base case, 2 points for the equation, 1 point for the implementation, 1 point for the running time.

Problem 6 (6 points). You are given a weighted oriented graph $G = (V, E)$, $|E| \geq |V|$. You want to find a path in G, longest *by the number of edges*, such that each consequent edge is greater than the previous one.

Create an $O((|V| + |E|) \log |V|)$ algorithm to find the *length* of such path.

Solution.

Let $OPT(v, E')$ be the longest path ending at $v \in V$ in a graph $G' = (V, E' \subset E)$.

Let $E_w \subset E$ be the set of edges with weight w. Then, for some $E' \subset E$ where all weights are less than w:

$$OPT(v, E' \cup E_w) = \max\left(\{OPT(v, E')\} \cup \{OPT(u, E') + 1 \mid (u, v, w) \in E_w\}\right).$$

The longest path in a graph $E' \cup E_w$ is either the same as in E' or includes only one new edge from E_w, which should also be the last. The reason it should be the last is that there are no edges in $E' \cup E_w$ with a larger weight to put after it. Thus, it's enough to consider all penultimate vertices in such paths, i.e. such vertices u that $u, v, w \in E_w$.

The answer is going to be $\max_{v \in V} OPT(v, E)$.

While the number of subproblems is potentially $|V| \cdot 2^{|E|}$, we don't need all of them to compute the answer.

First, note that we can compute subproblems layer-by-layer, starting with the subset of edges with the lowest weight, then second lowest etc. The number of such layers is $O(E)$, which brings the number of computed subproblems to $O(|V||E|)$. Still not good enough!

For another insight, note that in each layer E_w we only need to update vertices that are end-vertices of E_w. The rest stay the same. Thus, for each layer we run for only $O(|E_w|)$. In total, it's $O(|E|)$.

Additionally, we need $O(|E| \log |E|)$ to sort the edges by their weight and $O(|V|)$ to compute the final answer. Thus, resulting in a total of

$$O(|V| + |E| \log |E|) = O((|V| + |E|) \log |V|).$$

Grading Scheme: 4 points for the equations, 1 point for the implementation, 1 point for the running time.

Problem 7 (6 points).

You have a string s with only lowercase English letters. You can take any contiguous substring of s such that it has identical symbols and delete them all together from s. For example, if $s = xyyzqqqzzq$, you can delete qqq and get $xyyzzzq$.

Devise an algorithm to find the minimum number of such subsequent deletions to make s empty. The running time should be $O(n^3 \log n)$ (or, naturally, better).

Solution.

Let $OPT(i, j) = R(i, j)$ be the optimal solution for a substring of s from i to j, with j exclusive.

We can formulate the recursive update for R as the minimum of two cases: one where we delete s_i as a separate action and one where we delete it together with some other substring:

$$R(i,j) = \begin{cases} 0, & i \geq j \\ \min\left(\{1 + R(i+1, j)\} \cup \{R(i+1, k) + R(k+1, j) \mid s_k = s_i, k > i\}\right), & i < j \end{cases}$$

In the first case, we add one to however much it takes to delete the rest of the string. In the second case, we add however much it takes to delete the whole substring between s_i and some other character $s_k = s_i$ and then we can delete s_i together with $R(k+1, j)$.

The number of subproblems is n^2, the time to compute one is $O(n)$, for a total of $O(n^3)$.

Grading Scheme: 4 points for the equations, 1 point for the implementation, 1 point for the running time.