15-451/651 Algorithms, Fall 2018

Homework #3 Due: September 29, October 4, 2018

This HW has three regular problems, and one programming problem. All problems on this written assignment are to be done individually, no collaboration is allowed.

Solutions to the three written problems should be submitted as a single PDF file using gradescope, with the answer to each problem starting on a new page. The due date for this part is 11:59pm on Saturday September 29th. The due date for the programming part is Thursday October 4 at 11:59pm.

Unless otherwise stated, you should prove all of your answers. For example when presenting an algorithm you should prove correctness and the necessary running time bounds.

(25 pts) 1. **How Universal?**

In this problem we will modify the matrix method to construct a hash family that you will analyze for L-universality.

As in lecture, let the keys be $u$-bits long, and let the table size be $M = 2^m$. So an index is $m$-bits long. This time we pick two random $m \times u$ 0/1 matrices called $A_1$ and $A_2$. Our hash function is defined as follows:

$$h(x) = A_1 x \oplus A_2 \bar{x}.$$ 

The matrix-vector product uses modulo 2 arithmetic. The $\oplus$ indicates the exclusive or of two vectors of length $m$, and the symbol $\bar{x}$ indicates the bitwise complement of $x$. This defines a hash family $H$.

(a) Prove that $H$ is 3-Universal.

(b) Prove that $H$ is not 4-Universal.

(30 pts) 2. **Dynamic Programming 1**

For each part write a recurrence that can solve the problem. Make use of the given notation. Explain why the recurrence is correct. State the running time bound that memoization of your recurrence achieves, and justify the bound.

(a) **Obliterate 1.** You’re given a sequence $S = s_1, s_2, \ldots, s_n$ of $n$ bits. Your goal is to turn all the 1s to 0s in a way that achieves the maximum score. The way you do this is to pick a bit $s_i$ that is 1, and zero it. As a byproduct $s_{i-1}$ (if $i - 1 \geq 1$) and $s_{i+1}$ (if $i + 1 \leq n$) are also set to 0. The score of this move is $i$.

So for example if $S = [0, 1, 1, 0]$, there are only two ways to obliterate it. You can zero $s_2$ (which also zeros $s_3$ as a byproduct) for a score of 2, or you can zero $s_3$ (which also zeros $s_2$) for a score of 3.

Set up a recurrence for this DP algorithm, and analyze its memoized running time. The running time bound should be $O(n)$. 

(b) **Triangulations.** You’re given a convex polygon $P$ with $n$ vertices labeled $1, 2, \ldots, n$ in clockwise order. A triangulation is obtained by adding $n - 3$ chords across the polygon in such a way that they do not cross. (These chords are allowed to meet at the vertices of the polygon.) A triangle of the triangulation with vertices $\{i, j, k\}$ has cost that is specified, and denoted $T_{i,j,k}$. The goal is to find the cost of a minimum cost triangulation of $P$.

Let $P_{i,j}$ with $i < j$ be a polygon whose boundary consists of points $i, i + 1, \ldots, j$.

Let $C_{i,j}$ be the minimum cost of triangulating $P_{i,j}$.

Write a recurrence for $C_{i,j}$. The resulting running time bound should be $O(n^3)$.

(c) **Derivations.** You’re given a string $S$ of length $n$ consisting of $a$s and $b$s. You’re also given a list of eight transition rules, each with a cost. Rule 0 is $a \rightarrow aa$; rule 1 is $a \rightarrow ab$; rule 2 is $a \rightarrow ba$; …; rule 7 is $b \rightarrow bb$. The cost of applying rule $i$ is given, and denoted $C_i$.

Starting from a single letter $a$, $S$ can be derived by applying the rules one at a time. To apply a rule $x \rightarrow yz$, simply find a letter $x$ in the current string and replace it by $yz$. For example if $S=aaab$ it can be derived by applying rule 0 twice, followed by applying rule 1 to the rightmost $a$.

Let $A_{i,j}$ with $i \leq j$ be the minimum cost way to derive the range $S[i \ldots j]$ from a single $a$, and define $B_{i,j}$ analogously (start with a single $b$). Write a pair of mutual recurrences for these variables. Show that memoizing your recurrence results in an $O(n^3)$ algorithm for computing the desired answer.

(d) **Zig-Zag Subsequences.**

The original version of this problem has a non-DP $O(n)$ solution that we were not aware of. I’ve put the original here as well as a corrected one below. Feel free to do the original one, since some people already did it that way. Do the corrected one if you want to get more practice doing DP problems.

Original version:

A *zig-zag* sequence of numbers is one in which the comparison between successive numbers alternates between “bigger” and “smaller”. For example $[7, 2, 3, 1, 8, 7]$ is zig-zag and $[1, 7, 2, 3, 2]$ is also, but $[1, 2, 3]$ and $[1, 2, 2]$ are not.

Given an arbitrary sequence of numbers $S$ of length $n$, your goal is to find the length of the longest zig-zag subsequence of $S$. (Elements of the subsequence need not be contiguous.) For example the longest zig-zag subsequence of $[1, 2, 3, 4, 3, 2, 1, 2, 3, 4]$ is four, for example $[1, 4, 1, 4]$.

Corrected version:

Define zig-zag sequences and subsequences as in the original version. You’re given a sequence of numbers $S = s_1, s_2, \ldots, s_n$, along with a sequence of costs $c_1, c_2, \ldots, c_n$. The cost of $s_i$ is $c_i$. The cost of a subsequence of $S$ is just the sum of the costs its elements.

Give an $O(n^2)$ dynamic programming algorithm that computes the cost of the maximum cost zig-zag subsequence of $S$. For example if the input is $[1, 3, 1, 2, 1]$ with costs $[1, -10, 0, 0, 1]$ the maximum zig-zag subsequence is $[1, 2, 1]$ with a cost of 2.
3. **Dynamic Programming 2**

Don’t do all four of the following problems. Just do problems (a) and (b), or do (c) and (d). The same groundrules apply as in Dynamic Programming 1.

(a) **Obliterate2.** You’re given a sequence $S = s_1, s_2, \ldots, s_n$ of $n$ numbers in the range $[1, k]$. This time the goal is to obliterate the sequence at the minimum cost. The allowed move is to pick any number of the sequence and delete it. If you’re deleting a number $s_i$, the cost is $|s_{i-1} - s_i| + |s_i - s_{i+1}|$ (except the term involving $s_{i-1}$ or $s_{i+1}$ is not counted when the index is out of the range.) After a number $s_i$ is removed the sequence is shortened and the elements of the sequence are renumbered.

For example, suppose $S = [1, 3, 8]$. If we remove 3, then 8, then 1, the cost of this solution is $(|1 - 3| + |3 - 8|) + (|1 - 8|) + (0) = 14$. On the other hand if we do it in left to right order the cost is $(|1 - 3|) + (|3 - 8|) + (0) = 7$.

As usual, set up a recurrence for this problem, and analyze its memoized running time. (Shoot for $O(n^3)$.)

(b) **Obliterate to Something.** This is the same as Obliterate2, except you’re also given a goal sequence $G$, which occurs (perhaps many times) as a (possibly non-contiguous) subsequence of $S$. You must delete elements of $S$ until it becomes $G$, and to do it at the minimum cost.

Set up the recurrence, and again shoot for $O(n^3)$.

(c) **Subtree Counting in Binary Trees.** The input to this problem is a rooted tree $T$ with $n$ nodes, along with a bound $1 \leq K \leq n$. The root of the tree is denoted $r$. Let $n_i$ be the number of children of node $i$. And let $c_1, \ldots c_{n_i}$ denote the children of node $i$. In this problem $n_i \leq 2$ for all $i$.

Let $C_{i,k}$ be the number of distinct connected subgraphs of $T$ that have $k$ nodes that include $i$ and no ancestor of $i$. The goal is to compute $C_{i,k}$ for all $1 \leq k \leq K$ and all $1 \leq i \leq n$.

For example, suppose the tree is just a chain of $n$ nodes. Then $C_{r,k} = 1$ if $k \leq n$.

Or suppose $T$ is the following nine-node tree. Then $C_{r,4} = 6$. The bound to achieve is $O(nK^2)$.

```
     o
    / \  \
  o   o
 / \  \
o   o
/ \  /\
 o o o o o
```

(d) **Subtree Counting.** This is the same as the previous problem except that the number of children of a node is not limited. It may be more than two. Again, the bound to achieve is $O(nK^2)$.
(25 pts) 4. **Palindromes with One Mistake**

As you certainly know, a palindrome is a string that is the same when reversed. A *palindrome with one mistake* is a string with the property that you can change one character of it to make it a palindrome. An example is “dodad”.

The input to the program is a string of lower and upper case letters. The output is the longest (contiguous) substring of the input string that is a palindrome or a palindrome with one mistake (whichever is longer). If there are two that are equally long, output the leftmost one.

The input size is at most $10^6$ characters, and the time limit is 10 seconds.

For example, if the input is:

```
dogaaaaTaacat
```

then the output is:

```
aaaaTaaca
```

Second sample input:

```
dogaaaTaaacat
```

then the output is:

```
gaaaTaaac
```

Your algorithm should be guaranteed to give the right answer, even for inputs that an adversary chooses after looking at your code. So, for example, if your algorithm is randomized, it should be a *Las Vegas* algorithm. (That is, an algorithm which is guaranteed to give the right answer, but whose running time is a random variable.)

Include a comment in your submission that explains the algorithm that you used, along with a rough analysis of its running time (or expected running time, if your algorithm is randomized).