Solutions to homework 5. Due May 29

CS180: Algorithms and Complexity
Spring 2018

GUIDELINES:

- Upload your assignments to Gradescope by 5:59 PM.
- Follow the instructions mentioned on the course webpage for uploading to Gradescope very carefully (including starting each problem on a new page and matching the pages with the assignments); this makes it easy and smooth for everyone. As the guidelines are simple enough, bad uploads will not be graded.
- You may use results and algorithms from class without proofs or too many details as long as you state what you are using clearly. You may have to wait till Wednesday's class for Problems 3, 4.
- Additional Guidelines for this assignment
  - For any universe U and your choice of a range size r, you have access to random hash functions $h : U \to \{1, \ldots, r\}$ as we discussed in class. For any $u \in U$, computing $h(u)$ takes $O(1)$ time and you don't have to take into account the space for computing h.
  - You can use the running-times of the dictionary data structures we discussed in class. Make sure you explicitly state what your U is.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly. Homework is a stepping stone for exams; keep in mind that reasonable partial credit will be awarded and trying the problems will help you a lot for the exams.

1* Given two strings $X = x_1x_2 \ldots x_m$, $Y = y_1y_2 \ldots y_n$ of integers, a *common interlacing subsequence* is a sequence of indices $(i_1, i_2, \ldots, i_k)$ and $(j_1, j_2, \ldots, j_k)$, $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ and $1 \leq j_1 < j_2 < \cdots < j_k \leq n$ such that $x_{i_1} < y_{j_1} < x_{i_2} < y_{j_2} < \cdots < x_{i_k} < y_{j_k}$.

Give a polynomial-time algorithm that given X, Y as inputs finds a longest common interlacing subsequence between X and Y. You don't have to prove correctness or analyze the time-complexity of the algorithm.

For instance, if $X = [1, 3, 2, 5]$ and $Y = [2, 6, 4, 7]$, then if you take $i_1 = 1, i_2 = 2, i_3 = 4$ and $j_1 = 1, j_2 = 3, j_3 = 4$, you get an interlacing subsequence—$1 < 2 < 3 < 4 < 5 < 7$—of length 6. In fact, in this case 6 is the best possible answer. [.75 points]

[Hint: One approach is to sub-problems like we did for edit distance (with one modification) and then try to develop a recurrence relation based on a case-analysis of what the last two symbols of the interlacing sequence would be.]

Solution There are several ways to solve the problem revolving around the following sub-problem: For $1 \leq i \leq m$ and $1 \leq j \leq n$, let $A[i,j]$ be the longest common interlacing subsequence involving only the first i entries of x, the first j entries of y **and** ends at y_j (i.e., the last element of the subsequence is y_j). This set of sub-problems is similar to the sub-problems we had for edit-distance.

You can write down a recurrence relation between the $A[i,j]$ as above directly, but perhaps a cleaner way is to do the following. Let $B[i,j]$ be the longest common subsequence involving only the first i entries of x, the first j entries of y **and** ends at x_i (i.e., the last element of the subsequence is x_i). We can a design a recurrence between the two as follows:

$$A[i,j] = \max\{1 + B[t, j-1] : 1 \leq t \leq i, \text{ where } x_t < y_j.\}. \tag{1}$$

This is because, in the optimal solution to $A[i,j]$, we should have some element of $x_1, \ldots, x_i$ ahead of y_j, say it is x_t with $x_t < y_j$, then we want the longest common subsequence that involves $x_1, \ldots, x_t$, $y_1, \ldots, y_{j-1}$ and ends at x_t. Similarly, we get a recurrence

$$B[i,j] = \max(1, \max\{1 + A[i-1, t] : 1 \leq t \leq j, \text{ where } y_t < x_i.)\}. \tag{2}$$

(The additional max of 1 is account for the subsequence being just x_i - one that does not involve any y symbols.)

We can initialize the values as follows: $A[1,0] = A[0,1] = 0$, $B[1,0] = B[1,1] = 1$, $B[0,1] = 0$.

The algorithm is:

(a) Initialize: $A[1,0] = A[0,1] = 0$, $B[1,0] = B[1,1] = 1$, $B[0,1] = 0$.

(b) For $i = 1, \ldots, m$:

- For $j = 1, \ldots, n$:
  - Compute $A[i,j]$ using Equation 1 and $B[i,j]$ using Equation 2.

(c) RETURN $\max(A[m,j] : 1 \leq j \leq n)$.

The run-time of the algorithm is $O(mn(m+n))$ as we have at most $m \cdot n$ iterations and in each iteration, evaluationg Equations 1, 2 takes $O(m+n)$ time.

We can compute the optimal subsequence as usual from the recurrence relations by keeping track of which of the cases in the recurrences gives the best solution and updating the solutions accordingly.

2 Consider the complete rooted *quaternary* tree of depth n. That is take a rooted tree where the root has four children, each child has exactly 4 children and so on for n levels. So for example for $n = 1$, you have the root connected to its four children (with 5 nodes in total); for $n = 2$ you have 21 nodes in total; and more generally, the total number of nodes in the tree of depth n is exactly $1 + 4 + 4^2 + \cdots + 4^n = (4^{n+1} - 1)/3$.

Now, suppose that you delete each edge of the tree independently with probability 1/2. Let X be the number of nodes which are still connected to the root after the deletion of the edges. Compute the expectation of X (with proof). [.75 points]

[Hint: One clean approach is to write X as a sum of 'simpler' random variables and use linearity of expectation to compute the expectation.]

Solution. For each node v in the graph, let X_v be the random variable that is 1 if v is connected to the root and 0 otherwise. Then, $X = \sum_{v \in \text{ tree}} X_v$. Therefore, by linearity of expectation, $E[X] = \sum_{v \in \text{ tree}} E[X_v]$.

Now, for any node v at level k for $0 \leq k \leq n$, $E[X_v] = Pr[X_v = 1] = 1/2^k$ as v is connected to the root if and only if all the k edges on the path from the root to v do not disappear. Let L_k be all nodes at level k for $0 \leq k \leq n$ (so in particular, L_0 is just the root itself). Then,

$$\begin{aligned} E[X] &= \sum_{v \in \text{ tree}} E[X_v] = \sum_{k=0}^{n} \sum_{v \in L_k} E[X_v] \\ &= \sum_{k=0}^{n} \frac{|L_k|}{2^k} = \sum_{k=0}^{n} \frac{4^k}{2^k} \\ &= 2^{k+1} - 1. \end{aligned}$$

3 Suppose you are given n apples and n oranges. The apples are all of different weights and all the oranges have different weight. However, for each apple there is a corresponding orange of the same weight and vice versa. You are also given a weighing machine that (counter to common intuition) will **only** compare apples to oranges: that is, if you feed an apple and an orange to the machine it will tell you whether the apple or the orange is heavier or if they have the same weight. Your goal is to use this machine to pair up each apple with the orange of the same weight with as few uses of the machine as possible.

Give a randomized algorithm to determining the pairing. For full-credit, your algorithm should only use the machine $O(n \log n)$ times in expectation. You do not need to prove correctness or analyze the time-complexity of the algorithm. [.75 points]

(Remember that you cannot compare an apple to an apple or an orange to an orange.)

Solution: The algorithm is similar to quicksort. We pick a random orange, say o, from the set of oranges to serve as a pivot. By comparing this orange with the apples, we can split the apples into those apples less than o, A_-, and those that are more than o, A_+. In doing so, we also find the apple, say a, whose weight is equal to that of the orange o. We then, compare the oranges with a to split the oranges into those oranges that are less than a, O_-, and those that are more than a, O_+. We then return the pair (a, o) along with the outcomes of recursive calls to (A_-, O_-) and (A_+, O_+).

FindPairs(A,O): (Input: *List* of apples A, and *List* of oranges O.) (Output: Equal weight (apple,orange) pairs.)

(a) If $A = \emptyset$, RETURN.

(b) If $length(A) = 1$, RETURN $(A[1], O[1])$.

(c) Pick a uniformly random index p from $1, \ldots, length(O)$. Let $o = O[p]$.

(d) Initialize $A_-, A_+, O_-, O_+ = \emptyset$. Set $a = \perp$.

(e) For $j = 1, \ldots, length(A)$,

i. Use the machine to compare $A[j], o$.

ii. If $A[j] < o$, add $A[j]$ to A_-.

iii. Elseif $A[j] > o$, add $A[j]$ to A_+.

iv. Else set $a = A[j]$.

(f) For $j = 1, \ldots, length(O)$,

i. Use the machine to compare $a, O[j]$.

ii. If $O[j] < a$, add $O[j]$ to O_-.

iii. Elseif $O[j] > a$, add $O[j]$ to O_+.

(g) RETURN (a,o) + Findpairs(A_-, O_-) + Findpairs(A_+, O_+).

4 Suppose you are writing a plagiarism detector. Students submit documents as part of a homework and each document is an (ordered) sequence of words. For some parameter m decided by the provost, we say two documents are copies of each other if one of them uses a sequences of m words (in that given order) from the other. Give an algorithm which given an integer m and N documents $D_1, \ldots, D_N$ as input, flags all submissions which are copies of some other submission. Your algorithm should run in expected $O(m + N +$ total-length of documents) time (i.e., expected $O(m + N + n_1 + \ldots + n_N)$ time where n_j is the length of j'th document) and be always correct. [.75 points]

Solution. Create a dictionary $\mathcal{D}$ which stores $(Key, value)$ pairs where Key ranges over sequences of m words and $value$ ranges over integers. Here we will implement the dictionary again with hashing with chaining where the hash-table has size $r = \sum_i n_i$ and the universe U being all possible sequences of m words.

The algorithm is the following:

(a) Initialize an empty array $Copy$ of length N with all elements set to 0.

(b) For $i = 1, \ldots, N$:

i. For $j = 1, \ldots n_i - m$:

A. Compute $value = Lookup(D_i[j, j+m])$, where $D_i[j, j+m]$ denotes the sequence of words between the j'th and $(j+m)$'th position of document D_i.

B. If $value = NULL$ (meaning the key was not found in the dictionary), $Insert(D_i[j, j+m], i)$ into the dictionary $\mathcal{D}$.

C. Elseif $value = i$, do nothing.

D. Elseif $value \neq i$, set $Copy[i] = 1$ and $Copy[value] = 1$.

Let us first analyze the correctness of the algorithm. The algorithm is walking through every document and then through every sequence of m words based on the starting index of the sequence in the inner 'For' loop. For each such sequence $s = D_i[j, j+m]$, the algorithm

checks if s is already present in the dictionary in Step B. If it is not present, it adds it to the dictionary and doesn't flag the document as this sequence of words has not appeared before. If s was already present in the dictionary (in case *value* was not NULL), in step C we check if the first time s was seen was in D_i itself. In this case, there is nothing to do (a document cannot plagiarize itself). On the other hand, if s was seen in other document, meaning *value* is not equal to i then we flag both D_i and D_{value} as copies in Step D - which is what we should do.

To analyze the time-complexity, observe that for each index i, $1 \leq i \leq N$, the inner For loop does at most n_i many Insertions or Lookups. So the total number of Insertions and Lookups done through out the algorithm is at most $2(n_1 + \cdots + n_N)$. As we chose our hash-table size r to be $n_1 + \cdots + n_N$ (the maximum number of insertions) it follows from what we did in class that each of these dictionary operations take $O(1)$ expected time. So the total time spent in the For loops is $O(n_1 + \cdots + n_N)$ in expectation. Finally, we also spend $O(N)$ time to initialize the Copy array used to flag plagiarized documents. So the total time is $O(N + n_1 + \cdots + n_N)$.

Additional Problems. DO NOT turn in answers for the following problems - they are meant for your curiosity and understanding.

1. There are four types of brackets: (,), <, and >. We define what it means for a string made up of these four characters to be *well-nested* in the following way:

   (a) The empty string is well-nested.

   (b) If A is well-nested, then so are <A> and (A).

   (c) If S, T are both well-nested, then so is their concatenation ST.

   For example, (), < () >, ((< >)), () <> are all well-nested. Meanwhile, (, <>),)(, (<)>, and (<(> are not well-nested.

   Devise an algorithm that takes as input a string $s = (s_1, s_2, ..., s_n)$ of length n made up of these four types of characters. The output should be the length of the shortest well-nested string that contains s as a subsequence. For example, if (<(> is the input, then the answer is 6; a shortest string containing s as a subsequence is ()<()>. For full-credit, your algorithm should run in time $O(n^3)$ but you don't have to prove its correctness or analyze the time complexity.

   (Hint: For this problem, try to use sub-problems that solve the problem for substrings of the original s: $(s_i, s_{i+1}, \ldots, s_j)$ for all $i < j$.)

   Solution. We will solve the problem by a case-analysis and dynamic programming as above. For $1 \leq i \leq n$ and $1 \leq k \leq n$, let $s[i, k]$ denote the string of k symbols starting at position i. Concretely, $s[i, k] = s[i]s[i + 1] \cdots s[i + k - 1]$. Here, we adopt the convention that we always ignore indices which are greater than n (it simplifies notation and does not cost us anything).

   In a similar vein, let $OPT(i, k)$ be the length of the shortest well-nested string that contains $s[i, k]$ as a subsequence. Suppose we have computed $s[i, k]$ for all $1 \leq i \leq n - k + 1$. We next give a recurrence relation to compute $s[i, k + 1]$. As before, for this we will do a case-analysis as before. If $s[i + k]$ is a *left* symbol (meaning one of '(' or '¡')), then clearly $s[i + k]$ needs to

be matched to a newly inserted matching symbol. We then just have to solve for the string $s[i,k]$ and we can put the two together. Therefore, in this case $OPT(i,k+1) = OPT(i,k)+2$.

Now, suppose that $s[i+k]$ is a *right* symbol (meaning one of ')' or '¿'). Then, we have to high-level options: either we match $s[i+k]$ to a new matching symbol or an already existing matching symbol. In the first case, we might as well put the matching symbol right in front of $s[i+k]$. Once, we do this we have to solve the problem for $s[i,k]$. Thus, the cost in this case again is $OPT(i,k)+2$.

Let us now handle the more interesting possibility of matching $s[i+k]$ to an already existing symbol. For this, there are k options: we could conceivably match $s[i+k]$ to any of the appropriate symbols among $s[i], s[i+1], \ldots, s[i+k-1]$. If it is matched to the symbol at $s[i+j]$, then the problem splits into sub-problems. We have to find the optimal completion for the string $s[i]s[i+1]\cdots s[i+j-1]$ (the ones occurring before $s[i+j]$) and for the string $s[i+j+1]s[i+j+2]\cdots s[i+k-1]$ (the ones occurring between $s[i+j]$ and $s[i+k]$). Therefore, the cost in this scenario is $OPT(i,j)+OPT(i+j+1,k-j-1)$. This leads us to the following recurrence relation for computing $OPT(i,k+1)$. For two symbols x, y, let $Cost(x,y) = \infty$ if (x,y) do not match, i.e., $(x,y) \notin \{('(', ')'), (' <', ' >')\}$ and 0 if they match each other. Then, the above arguments show the following recurrence relation for OPT:

$$OPT(i,k+1) = \min \begin{cases} OPT(i,k)+2 \\ \min_{0\le j\le k-1} OPT(i,j) + OPT(i+j+1,k-j-1) + Cost(s[i+j], s[i+k])). \end{cases}$$

For initialization, we know that $OPT(i,1) = 2$ for any $1 \le i \le n$. This leads us to the following algorithm for computing OPT:

(a) Set $OPT(i,1) = 2$ for $1 \le i \le n$.

(b) For $k = 1, \ldots, n-1$
For $i = 1, \le, n$ Compute $OPT(i,k+1)$ using the above recurrence relation.

Our final desired answer is $OPT(1,n)$.

The correctness of the algorithm follows easily by induction. The base-case of $k = 1$ is correct as we've hardcoded into the algorithm. Further, if $OPT(i,k)$ are correctly computed for all $1 \le i \le n$, then $OPT(i,k+1)$ would all be correctly computed as we use the correct recurrence relation.

Time-complexity: The algorithm has $n \cdot n$ iterations and in any iteration, the total amount of work is at most $O(n)$ (as we have to look at a minimum of a total of at most n numbers. Therefore, the total run-time is $O(n^3)$.

2. Call a sequence of coin tosses "monotone" if the sequence never changes from Heads to Tails when parsed left to right. For example, the sequences $TTTHHHH$, $TTTT$ are monotone whereas $TTTHHHT$ is not. Consider a sequence of n coin tosses of a fair coin. For integer $k > 0$, let Y_k be the number of monotone sub-sequences of length k in the n coin tosses. Compute the expectation of Y_k (with proof).

   Recall that a sequence x is a sub-sequence of y if x can be obtained from y by deleting some symbols of y without changing the order of the remaining elements.

Solution. Let us first compute the probability that a sequence of k coin tosses is monotone. Note that a sequence is monotone if and only if after the first occurrence of a H every subsequent toss is a H as well. This implies that there are only $k+1$ sequences which are monotone as there are exactly $k+1$ options for the occurrence of the position of the first H: none, or the i'th position for $1 \leq i \leq k$. As any sequence of k coin-tosses has probability $1/2^k$ of occurring, the probability that a sequence is monotone is exactly $(k+1)/2^k$.

We next use use linearity of expectation once again. Let x be the coin-toss sequence. For a subsequence $I = \{i_1 < i_2 < \cdots < i_k\}$, let Y_I be the indicator random variable that is 1 if the sub-sequence $x_{i_1} x_{i_2} \cdots x_{i_k}$ is monotone. Then, $Y_k = \sum_I Y_I$, where the summation ranges over all subsequence of size exactly k as each monotone subsequence adds exactly 1 to the summation as needed. From the previous paragraph, $Pr[Y_I = 1] = (k+1)/2^k$ and hence $E[Y_I] = (k+1)/2^k$ (by definition of expectation). Therefore, by linearity of expectation (I ranges over all subsequences of size k),

$$E[Y_k] = E\left[\sum_I Y_I\right] = \sum_I E[Y_I] = ((k+1)/2^k) \cdot |\{\text{number of subsequences of size } k\}| = ((k+1)/2^k) \cdot \binom{n}{k}.$$

3. Define a random graph $G = (V, E)$ as follows. The vertex set of G is $V = \{1, \ldots, n\}$. Now for each pair $\{i, j\}$ with $i \neq j \in [n]$, add the pair $\{i, j\}$ to E with probability $1/2$ independent of the choice for every other pair. Let X_5 be the number of independent sets of size 5 in the graph G. Compute the expectation of X_5 (with proof).

   Recall that an independent set is a collection of vertices I in G such that no two vertices in I have an edge between them.

   Solution. We will once again use linearity of expectation (surprise!). For a subset $I \subseteq [n]$ of size 5, let X_I be the indicator random variable that is 1 if I is an independent set and 0 otherwise. Note that $Pr[X_I = 1] = 1/2^{10}$ as X_I is 1 only if none of the $\binom{5}{2} = 10$ pairs involving elements of I is connected which happens with probability $1/2^{10}$ (because each pair is an edge independently with probability $1/2$).

   Now observe that $X_5 = \sum_I X_I$ where the summation ranges over all subsets of size exactly 5. This is because each independent set of size 5 adds exactly one to the summation as needed. Therefore, by linearity of expectation (I ranges over subsets of size 5),

$$E[X_5] = E\left[\sum_I X_I\right] = \sum_I E[X_I] = (1/2^{10}) \cdot |\{\text{number of subsets of size } 5\}| = \binom{n}{5}/2^{10}.$$