CS 180: Introduction to Algorithms and Complexity
Final Exam

June 9, 2020

Name	
UID	
Section	

1	2	3	4	5	6	Total

★ **Print your name, UID and section number in the boxes above, and print your name at the top of every page**.

★ **Your Exams need to be uploaded in Gradescope. Use Dark pen or pencil. Handwriting should be clear and legible.**

- There are 6 problems.
- Do not write code using C or some programming language. Use English or clear and simple pseudo-code. Explain the idea of your algorithm and why it works.
- Your answers are supposed to be in a simple and understandable manner. Sloppy answers are expected to receive fewer points.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.

1. For each of the following problems answer True or False and briefly justify your answer.

   (a) (4pt) Prim's algorithm and Kruskal's algorithm will produce the same minimum spanning tree when the edge weights are distinct.

   (b) (4pt) Suppose we run Kruskal's algorithm but instead of following the increasing order of edge weights, we use the decreasing order of edge weights. This will return the spanning tree of maximum total cost.

   (c) (4pt) If G is a weighted, connected graph with n nodes containing a negative weight cycle, then for every two nodes s, t, the shortest path from s to t containing $n+1$ edges is strictly shorter than the shortest path from s to t containing n edges. (Note that here we allow a path to have duplicate nodes and edges.)

   (d) (4pt) If problem A is in P and problem B is in NP, then $A \leq_p B$.

   solution:

   (a) *True*. When the edge weights are distinct there is only one MST. Thus Prim's and Kruskal's return the same MST.

   (b) *True*. Suppose the weights are given by the weight function $w\colon E(G) \to \mathbb{R}$. Consider the negated weight function $w'(e) = -w(e)$. Clearly, the spanning tree returned by running Kruskal's in decreasing order of edge weight using w is the same as the spanning tree returned by running standard Kruskal's using w'. Let T denote this tree returned by both algorithms. By correctness of Kruskal's, T is minimum weight with respect to w', and thus T is maximum weight with respect to w.

   (c) *False*. Here's a counterexample. Let G be a triangle with $V(G) = \{s, t, u\}$ and edge weights $w(s,t) = -3$, $w(s,u) = 1$, and $w(u,t) = 1$. Clearly, G contains a negative weight cycle. The min-weight path on 3 edges from s to t is $(s,t),(t,s),(s,t)$ for a total weight of -9. The min-weight path on 4 edges from s to t is $(s,t),(t,s),(s,u),(u,t)$ for a total weight of -4.

   (d) *True*. Since A is in P, there is a trivial reduction from A to B. Given an instance x of A, run the poly-time algorithm for A on x to decide if x is a yes or no instance of A. If x is a yes instance then output a yes, and if x is a no instance, then output a no.

2. (4 pt) Assume we have the following three divide-and-conquer algorithms:

   - For problem with size n, solve 7 sub-problems of size $n/7$, and use $O(n)$ time to combine the results to get the solution of the original problem.
   - For problem with size n, solve 16 subproblems of size $n/4$, and then constant time to combine the results to get the solution of the original problem.
   - For problem with size n, solve 2 subproblems of size $n/2$, and the $O(n^2)$ time to combine the results to get the solution of the original problem.

   Calculate the time complexity for each algorithm and show which one is the fastest.

   solution: $O(n \log n)$, $O(n^2)$, $O(n^2)$, so the first algorithm is the fastest one.

   - $T(n) = 7T(\frac{n}{7}) + O(n)$, so $T(n) = O(n) \cdot$ depth $= O(n) \cdot O(\log n) = O(n \log n)$
   - $T(n) = 16T(\frac{n}{4})) + O(1)$, $f(n) = O(n^0)$, $\log_b a = \log_4 16 = 2 > 0$, by the master theorem case 1, $T(n) = O(n^2)$.
   - $T(n) = 2T(\frac{n}{2})) + O(n^2)$, $f(n) = O(n^2) = \Omega(n^2)$, $\log_b a = \log_2 2 = 1 < 2$, by the master theorem case 3, $T(n) = O(n^2)$

3. (20pt) There is an array with n integers, but the values are hidden to us. Our goal is to partition the elements into groups based on their values — elements in the same group should have the same value, while elements in different groups have different values. The values are hidden to us, but we can probe the array in the following way: we can query a subset of these n elements, and get the number of unique integers in this subset. Design an algorithm to partition these n elements in $O(n \log n)$ queries.

   solution. Assume we have partitioned the first m elements into K groups such that elements in the same group have the same value and elements in different groups have different values. Now we consider the $(m+1)$-th element, we first query $\{1, \ldots, m, m+1\}$. If this outputs $K+1$, then we know that the $(m+1)$-th element is in a new group different from existing K groups. Otherwise, if the output of the query is K, then we know the $(m+1)$-th element must belong to one of the K groups.

   In the latter case, we need to determine which group the $(m+1)$-th element belongs to. To this end, we denote these groups as $G_1, \ldots, G_K$. We can first query the subset $\{m+1\} \cup G_1 \cup \ldots G_{K/2}$ which contains elements $m+1$ and all the elements from the first $K/2$ groups. Based on the output, we know whether element $m+1$ is in the first $K/2$ groups or the other half of groups. Once we decide which half it belongs to, we can again divide these groups into two halves and repeat the above process. Note that there are K groups in total, which means we will repeat this recursion at most $O(\log K)$ times before we know which of the K groups the $(m+1)$-th element belongs to.

   Iterate the whole process from $m = 0$ to $m = n-1$, we finish the grouping of these n elements. The time complexity (total number of queries) is $O(\sum_{m=0}^{m=n-1} \log K) = O(n \log n)$. The pseudo code is displayed as follows.

```
arr with n elements
Group[1]={1}; Group[i]={} for all i ≠ 1
K=1
for m = 1,...,n
  if query(arr[m+1], Group[1], ..., Group[K]) = K+1
    Group[K+1]={m+1}
    K ++
  else if query(arr[m+1], Group[1], ..., Group[K]) = K
    l = 1, r = K, mid = (r − l + 1)/2
    while(r − l > 0)
      mid = (r − l + 1)/2
      if query(arr[m+1], Group[l], ..., Group[mid]) = mid − l + 1
        r = mid
      else if query(arr[m+1], Group[l], ..., Group[mid]) = mid − l + 2
        l = mid
    Group[l]={Group[l], m+1}
```

4. (20pt) A phone company divides a city into n cells $c_1, c_2, ..., c_n$. In each cell it has a tower. When a call comes to a mobile user the company has a set of probabilities $p_1, p_2, ..., p_n$ such that the user is now at cell c_i with p_i probability. The company wants to activate as few towers as possible on average to find the user. When it activates a tower in a cell where the user is, the search stops. Search time is divided into d slots. By the end of d slots the user must be found. The question the company faces is what are the towers(cells) to activate in slot $1, 2, ..., d$ as to minimize the expected number of activations. More specifically, the company wants a policy which is a collection of d sets $S_1, S_2, ..., S_d$ where S_i contains towers (cells) to be activated at slot i.

   For example, let $n = 5$ and $d = 3$, a policy might be $S_1 = \{c_1\}, S_2 = \{c_2, c_4\}, S_3 = \{c_3, c_5\}$. The expected number of activation for this policy is $1 + 2 \cdot (1 - p_1) + 2 \cdot (1 - p_1 - p_2 - p_4)$

   (a) What is the optimal policy if $p_i = \frac{1}{n}$ for all i and $d = 2$. Prove it to be optimal. [5 pts]

   (b) Prove that in an optimal policy, S_1 is a set of cells which is a prefix of the non-increasing sorted order of the cells according to their probability. [5 pts]

   (c) Design a polynomial time algorithm to calculate the optimal policy in general. Justify the correctness of your algorithm and it's time complexity. [10 pts]

   solution.

   (a) $|S_1| = |S_2| = n/2$ if n is even; $|S_1| = (n-1)/2$ or $|S_1| = (n+1)/2$ if n is odd.

   (b) Simply by exchange argument.

   (c) Let $OPT_{i,j}$ denotes the minimal expected number of activations with i slots and j cells.
   Then $OPT_{1,j} = j$ and $OPT_{i,j} = min_{1 \le k < j}\{OPT_{i-1,k} + (j-k)(1 - \Sigma_{t=1}^{k} P_t)\}$.

5. (20pt) Given a large $W \times L$ rectangle, we want to cut it into smaller rectangles of specific shapes $(a_1 \times b_1), (a_2 \times b_2), \ldots, (a_K \times b_K)$. Note that all these numbers, including $W, L, a_1, \ldots, a_K, b_1, \ldots, b_k$ are integers, and each time we can only make a full horizontal or vertical cut on a rectangle at an integer point to split it into two. In the end we will get a collection of small rectangles, hopefully most of them have the shape matching one of the $a_i \times b_i$, but there could be pieces that don't match with any pre-specified shapes and those areas are wasted. For simplicity we assume the rectangles cannot be rotated (so $a_i \times b_i$ is different from $b_i \times a_i$). We don't care about how many of these smaller rectangles we get in the end, but our goal is to minimize the total wasted area. Design an algorithm that runs in polynomial time of k, W, L that computes the minimum possible wasted area.

   For example, assume $W = 21, L = 11$ and the desired rectangles are $(10\times4), (9\times8), (6\times2), (7\times5), (15\times10)$. The minimum possible wasted area is 10 (the gray area), as shown in Figure 1.

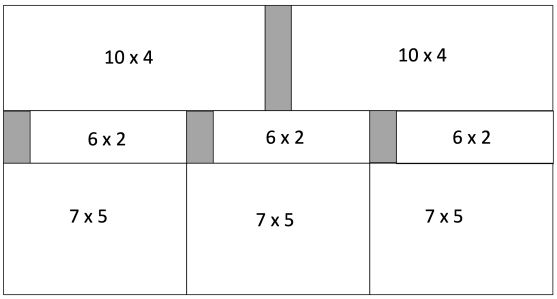

Figure 1

Solution: Dynamic programming: Define OPT(i, j) as the minimum wasted area for rectangle with width i and height j. Initially, we set every OPT(i, j) with a trivial solution OPT$(i, j) = i * j$. Also, to be noticed, when the rectangle matches the exact size of smaller rectangle candidates, we set that OPT$(i, j) = 0$. That is,
Initial:

$$\text{OPT}(i,j) = \begin{cases} 0, & \text{matches candidate } (a_i, b_i) \\ i * j, & \text{otherwise} \end{cases}$$

When cutting horizontally, we have OPT$(i, j) =$ OPT$(a, j) +$ OPT$(i - a, j)$. When cutting vertically, we have OPT$(i, j) =$ OPT$(i, b) +$ OPT$(i, j - b)$. Therefore, we need to find the minimum wasted area for $0 < a < i$ and $0 < b < j$. So we have the following recursive rule for computing the OPT values as follows:

$$\text{OPT}(i,j) = \min\{\text{OPT}(i,j), \min_{0<x<i} \text{OPT}(x,j) + \text{OPT}(i-x,j), \min_{0<y<j} \text{OPT}(i,y) + \text{OPT}(i,j-y)\} \quad (1)$$

Input: candidate set S, rectangle size W,L
for $i, j = 1, \ldots, n$
  OPT$(i, j) = i * j$
for i,j in S
  OPT$(i, j) = 0$
for $i, j = 1, \ldots, n$
  OPT$(i, j) = \min\{\text{OPT}(i,j), \min_{0<x<i} \text{OPT}(x,j) + \text{OPT}(i-x,j), \min_{0<y<j} \text{OPT}(i,y) + \text{OPT}(i,j-y)\}$
Return OPT(W, L)