CS 180: Introduction to Algorithms and Complexity
Final Exam

June 9, 2020

Name	James Youn
UID	505399332
Section	Lec 1, Dis 1I

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.

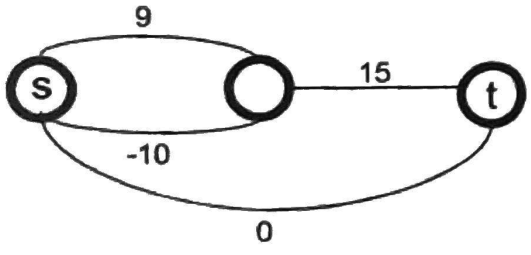
   (c) (4pt) If G is a weighted, connected graph with n nodes containing a negative weight cycle, then for every two nodes s, t, the shortest path from s to t containing $n+1$ edges is strictly shorter than the shortest path from s to t containing n edges. (Note that here we allow a path to have duplicate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

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

a) Suppose they produced different MST's, then there is an edge e in one tree (assume without loss of generality that it's in Prim's tree) that is not in the other. Call s1 and s2 the two cuts that are connected by edge e. There must be an edge e' that connects s1 and s2 in Kruskal's tree. The tree with the greater weight edge is not an MST because it could have replaced its edge with the lesser one, which is a contradiction. Therefore, **true**.

b) The max spanning tree problem can be turned into a minimum problem by multiplying every edge weight by -1 (call this new graph G'). The minimum weight edge added from G' would be the maximum one in G (original graph). Therefore, adding edges in decreasing order in G creates a tree with the same edges as adding them in increasing order in G'. Moreover, the Min Spanning Tree produced from G' would be the max one in G. Thus, **true**.

c) According to piazza question #335, the instructor allowed traversing back and forth on a single edge.



The shortest path containing at most 3 edges from s to t is traversing the -10 edge back and forth 2 times, followed by weight 0 edge, resulting in a path length of -20.
Shortest 4 edge path from s to t is traversing the -10 edge back and forth 3 times, followed by weight 15 edge, resulting in a path length of -15.

Therefore, **false**.

d) Definition of A <p B: If B can be solved in polynomial time, then A can be solved in polynomial time. Because A is in P, A can already be solved in polynomial time. If we assumed that B can be solved in polynomial time, we can have A make a polynomial number of unnecessary calls (does not change output) to B and A can still be solved; this works because there's already an algorithm unrelated to B that can solve A. Therefore, **true**.

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

1 • 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.

2 • 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.

3 • 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.

1)

$T(n) = 7T(n/7) + cn$

Suppose $T(n) = knlog_7(n)$, then $knlog_7(n) = 7k(n/7)log_7(n/7) + cn = knlog_7(n) - kn + cn$. Therefore, $T(n) = cnlog_7(n)$ and O(nlogn)

2)

$T(n) = 16T(n/4) + c$

Suppose $T(n) = kn^2 + a$, then $kn^2 + a = kn^2 + 16a + c$. Therefore, $T(n) = kn^2 - c/15$ and O(n^2)

3)

$T(n) = 2T(n/2) + cn^2$

Suppose $T(n) = kn^2$, then $kn^2 = \frac{1}{2}kn^2 + cn^2$. Therefore, $T(n) = 2cn^2$ and O(n^2)

Thus, **1 is the fastest**

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.

Iterate through the input array. In each iteration, we will finish that element (find all other elements that have the same value and make them a group). Mark each of the elements that we add to the group to avoid repeats.

We find such elements in a way similar to a binary search. Suppose that the current iteration is trying to finish an element e, then we try to find the leftmost element e' to the right of e that has the same value. We then transfer the role of e to e' and repeat the process until we've added all elements corresponding to e into one group. We only have to consider elements to the right of e because if there were a same valued element to the left, then that element and those to its right (including e) would be part of a group already, and done would be 1.

query(i,j): returns # of unique elements from i to j inclusive
unk[]: stands for unknown. The original input array, indexed from 1 to n
groups: vector that stores sets, each containing the indices of same valued elements
done[i]: stores 1, if i has been added to a group. If not, stores 0

Functions:

```
groupMe(i) :
    add i to last set in groups; done[i] = 1
    if i == n OR query(i,n) != query(i+1,n) : return
    a = i+1; b = n
    while a != b :
        if query(i, (a+b)/2) == query(i+1, (a+b)/2) : b = (a+b)/2  //assuming division removes decimals
        else : a = (a+b)/2 + 1
    groupMe(a)
```

Algorithm:

```
Loop i:[1,n] :
    if done[i] == 0 :
        add an empty set to groups; groupMe(i)
```

Time complexity and proof:

The time it takes from initial call to groupMe(i), to right before recursively invoking groupMe(a), takes O(logn) in terms of queries: it simply finds the leftmost element that has the same value as unk[i] using binary search. Since we never call groupMe() more than once for every index of unk[], the algorithm runs in O(nlogn).

Every element is part of a group in the end with no element repeated due to the done array. It's clear from the groupMe function that the algorithm successively finds the next same valued element and adds it to a group. Therefore, in the end, all groups consist of the full set of the same elements.

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]

Name: James Youn UID: 505399332

Q4

a) If x is the number of cells in S1 and the rest are in S2, then the expected number of activations is x + (1 - x/n)(n-x) = n - x(n-x) / n. We want to maximize the term x(n-x). Since this is an upside-down parabola with roots at x = 0 and n, the maximum occurs at **x=n/2** (assuming the decimal portion is truncated).

b) I'm assuming "prefix" means a subgroup comprised of the first element(s) of the sorted cells. Define q1, q2,..., qn as the sorted probabilities in non-increasing order. If x number of elements belongs to S1, then S1 is not a prefix if and only if qi is in S1 such that i > x. Assume we have an optimal policy and that S1 is not a prefix, then there is a qj that is not in S1 such that j <= x. Exchanging qi with qj will result in less activation, because now it's more likely that S1 contains the user's cell and later slots have a decreased probability factor. This contradicts that we have an optimal policy, and thus S1 must be a prefix.

c) Piazza question #354 indicates that sets can be empty.
It's clear from part b that in an optimal policy, S1, S2,..., Sd consist of consecutive prefixes of the sorted cells. If even one of the slots did not consist of a prefix, then the incorrectly placed cell (lower probability) can be exchanged with the appropriate one (higher probability) so that the higher one comes earlier in the slot order. Now, it's more likely that the user's cell will be found earlier, and there is a decreased probability of having to activate later slots, which contradicts optimality. We can now use dynamic programming to partition S1,..., Sd to prefixes that result in an optimal policy.

q1, q2,..., qn = sort(p1, p2,..., pn) //make sure cells map to sorted probabilities
sum(i): indexed from 0 to n, array that stores sum from q1 to qi inclusive, sum(0) = 0
A(b,a): minimum number of activations for b cells and a slots
For all b and a such that a==1, A(b,a) = b
x: |Sa| (cardinality of last slot)

Loop i:[1,n] :
    sum(i) = sum(i-1) + qi
Loop a:[2,d] :
    Loop b:[1,n] :

$$A(b,a) = min_{x:[0,b]}(A(b-x,a-1) + (1 - sum(b-x))(x))$$

Time complexity and proof:

Time complexity is **O(dn^2)**: Sort is O(nlogn). Initializing array sum takes O(n). We're iterating over a 2d array (dimensions n x d), and each time we do, we iterate over x (0 to n), which takes O(dn^2) -- the dominating process.

Given that A(b,a) returns the optimal activations such that b<n and a<d, the algorithm gives the optimal solution to A(n,d) because 0 to n are all the possible cardinalities for the last slot, and it takes the minimum among these possibilities.

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.

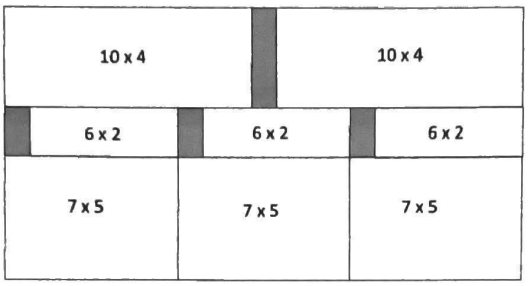


Figure 1

Name: James Youn UID: 505399332

Q5

R: the rectangle under consideration (deciding on whether to cut into two) (not necessarily original rectangle)
R is indexed from 0 to a (width) and 0 to b (length), where 0, a, and b are sides
S: the set of desired/pre-specified rectangles
r: a certain rectangle in S
A(a,b): wasted area from a rectangle that has dimensions a x b

First, check to see if r fits into R perfectly. If it does, then return 0 for the wasted area. If R becomes too small for any r, then return R's area. Else, cut R optimally, and the total wasted area from both cuts is the solution; find the optimal cut by simulating all possible horizontal and vertical cuts to see which results in the smallest wasted area.

Functions:

fit(w, l) : //determines whether w x l is large enough for one of the desired rectangles
    Loop through S : If there exists a rectangle that fits : return true
    return false
pFit(w, l): //determines whether w x l is the exact same dimension as one of the desired rectangles
    Loop through S : If there exists a rectangle that perfectly fits : return true
    return false

Algorithm:

Loop a:[1,W] :
    Loop b:[1,L] :
        if pFit(a,b) is true : A(a,b) = 0
        else if fit(a,b) is false : A(a,b) = a*b
        else : $A(a,b) = min(\ min_{i:[1,a-1]}(A(i,b) + A(a-i,b)),\ min_{j:[1,b-1]}(A(a,j) + A(a,b-j))\ )$

Time complexity and proof:

There are two nested loops, O(WL), with pFit and fit running in O(k); A(a,b) is computed in O(W+L). Therefore, the time complexity is **O(WL(k+W+L))**.

Given that A(a,b) returns the optimal solution such that a<W and b<L, the algorithm works. Given any single rectangle, the choices are to cut it or to stop either because it's a desired rectangle or it's too small for any. The algorithm considers each of these choices and for the one that requires a cut, it evaluates the cost of every possible cut and takes the minimum.

6. (20pt) Consider the problem of "Approx-3SAT": The input is the same as 3-SAT, which is a boolean expression $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ where each C_i is an "or" of three literals (each literal can be one of $x_1, \ldots, x_m, \neg x_1, \ldots, \neg x_m$). Note that we assume a clause cannot contain duplicate literals (e.g., $(x_1 \vee x_1 \vee x_2)$ is not allowed). Instead of determining whether there's a truth assignment on $\{x_i\}_{i=1}^m$ that satisfies this boolean expression (which means it satisfies all the clauses), now we want to determine whether there's an assignment that satisfies at least $n-1$ clauses. Prove that Approx-3SAT is NP-Complete.

Let's say that 3SAT and Approx-3SAT return true if and only if their clauses can be satisfied.

Claim: 3SAT <p Approx-3SAT.

Proof: Add another clause whose three literals are F, F, and F to the original problem. If Approx-3SAT returns true (there is a way to satisfy all the original clauses), then 3SAT on the original clauses is true. If Approx-3SAT returns false, then there is a clause from the original problem that can't be satisfied and 3SAT on the original clauses returns false. Given this biconditional relationship, 3SAT can be reduced to Approx-3SAT.

Textbook theorem 8.15 states that 3SAT is NP-complete, meaning if problem A is in NP, then A <p 3SAT. Since 3SAT <p Approx-3SAT, all problems in NP can be reduced to Approx-3SAT. Therefore, Approx-3SAT is NP-complete.