CSE 101 Practice Final Fall 2019

True/False: For each of the following answer "True" or "False" and give a brief explanation (1 or 2 lines or sentences.)

1. DFS on a dense graph ($|E| = \Theta(|V|^2)$ has runtime $\Theta(|V|^2)$

   Solution: True

2. Suppose $f(n)$ and $g(n)$ are positive-valued increasing integral functions. If $2^{f(n)} \in O(2^{g(n)})$ then $f(n) \in O(g(n))$

   Solution: True

3. If a connected undirected graph has two edges that both have the minimum weight then the graph has 2 distinct MSTs.

   Solution: False

4. If a problem is in NP then there is no known polynomial time algorithm to solve it.

   Solution: False

5. If you perform explore on a DAG starting at a source vertex, then all vertices will be marked as visited.

   Solution: False

6. If an undirected connected graph has a cycle with a unique lightest edge e, then e is part of all MSTs

   Solution: True

7. $n + (n-2) + (n-4) + ...2 \in O(n)$.

   Solution: False

8. $2^n + 2^{n-1} + 2^{n-2} + ...1 \in O(2^n)$.

   Solution: True

9. It is always better to use a heap than an array as the data structure in Dijkstra's algorithm.

   Solution: False

10. If two problems are both NP-complete, and one has a polynomial-time algorithm, then the other also has a polynomial time algorithm.

    Solution: True

Paths in Graphs Give an $O(|E| + |V|)$ algorithm, possibly using known algorithms from class as subroutines, to tell, given a directed graph G and two vertices s and t, whether there is a not-necessarily simple path from s to t whose length is a multiple of 3. You can use without proof the correctness and time analysis of algorithms covered in class, but need to relate the above problem to the algorithms in question.

Solution:

Create a new graph G' from G so that for each vertex v in G there are three copies v_1, v_2, v_3 in G'. For every edge (u, v) in G, G' will have edges $(u_1, v_2), (u_2, v_3)$ and (u_3, v_1). Then you run DFS from s_1 and if t_1 is visited then there is a path with length a multiple of 3 from s to t in G.

Justification of correctness idea:
If there is a path from s_1 to t_1 in G', then the sequence of vertices in the path must be of the form $(s_1, \_2, \_3, \_1, \_2, \_3, \ldots, \_1, \_2, \_3, t_1)$, in other words you must go through vertices in the order
1,2,3,1,2,3...1,2,3,1 and so if you start with 1 and end with 1, then the number of edges will be a multiple of 3.

Runtime Analysis: Creating G' will take $O(|V| + |E|)$ because for each vertex, you must create an adjacency list for G' that has 3 times as many edges then loop through the original adjacency list and for each edge, create 3 new edges in G' so you have to do 3 constant time operations for each vertex and each edge so the runtime is $O(|V| + |E|)$. Then DFS on this new graph is $O(3|V| + 3|E|) = O(|E| + |V|)$.

Shortest Paths in Graphs Suppose you have a directed graph G with vertices s and t and edge lengths $\ell(e) \in \{0, 1\}$. Design a $O(|V| + |E|)$ time algorithm that returns the length of the shortest path from s to t. (Note that running Dijkstra's on this graph will not be $O(|V| + |E|)$.

Solution:
Running Dijkstra's using a binary heap or an array will not achieve the $O(|V| + |E|)$ runtime. We can essentially run Dijkstra's using a specialized priority queue in the following way:

We can run BFS using a double-sided queue Q. We start with s in Q and set $dist(s) = 0$. We pop vertices only from the left. So if the next vertex we pop is v then we insert all of v's neighbors in the following way: if the edge (v, u) has length 0 then push the vertex u into the left side of the queue and set $dist(u) = dist(v)$ and if the edge (v, u) has length 1 push the vertex u into the right side of the queue and set $dist(u) = dist(v) + 1$. Now, it may be the case that a vertex is pushed into the queue more than once. We want to set its dist value the *first* time it is popped so we can keep an *already popped* array so that we can check to see if a vertex has already been popped.
Justification of correctness:
Essentially, this double-sided queue is acting as a priority queue. So, we must show that at all times, the dist values of all vertices in the queue are sorted from left to right. Therefore, when we pop a vertex from the left, it is guaranteed to be the smallest dist in the queue. We will actually prove something stronger, that at all times the queue will only hold vertices with values d and $d + 1$ where d is the dist of the left-most vertex.
proof by induction:
Base Case: The queue starts with just s and one element is trivially sorted.
Inductive Hypothesis: Suppose that all vertices in the queue are sorted by dist values and v is the left-most vertex with dist value d and assume that the dist values of all vertices is either d or $d + 1$.
Inductive Step: Then we pop v and add all of v's neighbors:
If $\ell(v, u) = 0$, then we add u to the left of the list with $dist(u) = d$ and since all vertices are sorted in the queue with dist values either d or $d+1$, this maintains the inductive hypothesis.
If $\ell(v, u) = 0$, then we add u to the right of the list with $dist(u) = d+1$ and since all vertices are sorted in the queue with dist values either d or $d + 1$, this maintains the inductive hypothesis.
Runtime Analysis: Popping and pushing vertices each takes constant time. You will pop each vertex at most once so the time for popping is $O(|V|)$. You will push a vertex at most $|E|$ times, so the time for pushing is $O(|E|)$. So the runtime is $O(|V| + |E|)$.

Divide and Conquer: Consider the following problem: You are given a pointer to the root r of a binary tree, where each vertex v has pointers $v.lc$ and $v.rc$ to the left and right child, a positive weight $w(v)$, and each edge points from parent to child. The value NIL represents a null pointer, showing that v has no child of that type. You wish to find the total weight of the heaviest path starting from the root that uses an even number of edges

Design a divide and conquer algorithm to do this (you may assume that the graph is complete and the bottom level is filled.

Algorithm Idea: Define the algorithm EvenPath(r) to return the maximum weight path starting from r that uses an even number of edges. If r is null, then return 0. If r is a leaf then return w(r).

Otherwise, compute LL = EvenPath(r.lc.lc), LR = EvenPath(r.lc.rc), RL = EvenPath(r.rc.lc), RR = EvenPath(r.rc.rc).

Then return w(r) + max(w(r.lc) + max(LL,LR),w(r.rc) + max(RL,RR)).

EvenPath(r)
  if r is null:
    return 0
  if r is a leaf:
    return $w(r)$
  if $r.rc$ and $r.lc$ are both leaves:
    return $w(r)$
  $LL = EvenPath(r.lc.lc)$
  $LR = EvenPath(r.lc.rc)$
  $RL = EvenPath(r.rc.lc)$
  $RR = EvenPath(r.rc.rc)$
  return $w(r) + \max[w(r.lc) + \max(LL, LR), w(r.rc) + \max(RL, RR)]$

Runtime Analysis: There are four subproblems each of size $(n-3)/4$ and a constant non-recursive time so the runtime is $T(n) = 4T(n/4) + O(1)$ so $T(n) = O(n)$

Divide And Conquer 2: Consider the following weighted selection problem: You are given a list of ordered pairs $(x_1, f_1), \ldots, (x_n, f_n)$ where x_i is an integer and f_i is the frequency that integer appears and an integer k such that $1 \leq k \leq \sum f_i$. You wish to find the kth smallest element in the multiset of integers that the sequence represents.

For example: $(10, 4), (2, 7), (3, 5), (9, 3), (6, 1)$ means that there are 4 tens, 7 twos, 5 threes, 3 nines, and 1 six. You can think of this as representing a multiset of integers such that there may be repeats of integers. The example above describes the following sorted sequence of integers:

$$[2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 6, 9, 9, 9, 10, 10, 10]$$

If $k = 15$ then you would output 9 because 9 is the 15th element in the expanded sequence for the example above.

Design a divide and conquer algorithm to solve this problem in $O(n)$ time where n is the number of ordered pairs. The pairs are not necessarily in sorted order. (Hint: you can use the select algorithm from class to find the median of a list in $O(n)$ time.)

Algorithm Description: This algorithm will take the list of ordered pairs and an integer k.

If $n = 1$ then return the only integer.

Use the selection algorithm from class to find the median of all the *integers* (first entries) and call it m and let's call its frequency $f(m)$. Partition the list of ordered pairs by separating them into L, the integers less than m and G, the integers greater than m. Let S_L be the sum of all the frequencies of L and let S_G be the sum of all the frequencies of G.

If $k \leq S_L$ then recurse on (L, k)

If $S_L < k \leq S_L + f(m)$ then return m.

If $k < S_L + f(m)$ then recurse on $(R, k - f(m) - S_L)$

Runtime Analysis: There is one recursive call half the size and the non-recursive part is $O(n)$, so the recursion for the runtime is $T(n) = T(n/2) + O(n)$ and by the master theorem, $T(n) = O(n)$.

Greedy Algorithms and data structures You have a long corridor in your palace. Unfortunately, there are n unsightly marks on the floor of the corridor, at $x_1, ...x_n$ yards from the door. You will use some very expensive rugs whose width fills the corridor and are each one yard long to cover the marks. For simplicity, we will allow rugs to overlap, and allow part of the rug to extend past the end of the corridor. (Formally, we identify rugs with intervals of length 1 , and it hides the mark at x_i if x_i is in the interval.)

Part 1 Below is a greedy strategy that does not always find the minimum number of rugs. Give a counter-example where it fails to produce the optimal solution.

Candidate Strategy A : Until there are no more unhidden marks, place a rug at the interval containing the most unhidden marks.

Solution: Suppose the marks are: $x_1 = 0.5, x_2 = 1.1, x_3 = 1.4, x_4 = 1.6, x_5 = 1.9, x_6 = 2.5$ then you can cover 4 spots by putting a rug from 1 to 2. But then you would need a rug for x_1 and for x_6 totaling 3 rugs. Alternatively you could put a rug from 0.5 to 1.5 and from 1.5 to 2.5 to cover all the spots.

Part 2 Below is a greedy solution that does always find the minimum number of rugs.

Candidate Strategy B: Until there are no more unhidden marks, let x_i be the first unhidden mark (i.e., the smallest x_i not yet in an interval.) Place a rug from x_i to $x_i + 1$.

Illustrate the above strategy on the counter-example in part 1.

Solution: Put the first rug starting at $x_1 = 0.5$. Then x_1, x_2, x_3 are all covered. Put the second rug starting at $x_4 = 1.6$ then x_4, x_5, x_6 are all covered.

Part 3 Prove that this greedy solution is optimal.

Solution: Exchange: Suppose that OS is a valid non-empty solution that does not include putting the first rug starting at the smallest mark x_1. Create OS' by replacing the first rug of OS with a rug that starts at x_1.
Proof that OS' is valid. We must show that all marks that were covered by the first rug of OS are also covered by the greedy rug. Let x_j be an arbitrary mark covered by the first rug of OS and assume this rug spans from a to $a + 1$. Then $a \leq x_j \leq a + 1$.
We also know that x_1 must be covered by this rug as well, so $a \leq x_1 < x_j \leq a + 1$.
Since $a \leq x_1$, we also have that $a + 1 \leq x_1 + 1$. So this implies that $x_1 < x_j \leq x_1 + 1$ which means that x_j is also covered by the greedy rug.
$|OS| = |OS'|$

Induction part: Claim: The greedy strategy is optimal for all inputs of size n where $n \geq 0$.
Base Case: $n = 0$, then the greedy strategy is the empty set which is optimal.
Inductive Hypothesis: Suppose the greedy strategy is optimal for all inputs of size k with $0 \leq k < n$.
Inductive Step: Suppose I is an input of size n and OS is some valid solution of I. Then:

$$|OS| = |OS'| = |\{g\} \cup S(I')| \geq |\{g\} \cup GS(I')| = |GS(I)|$$

where I' is the set of marks that are not covered by g.

Part 4 Describe an efficient algorithm that carries out strategy B. Your description should specify which data structures you use, and any pre-processing steps.

Solution:
Marks$(x_1, \ldots, x_n)$
    if $n == 0$:
        return $\emptyset$
    $i = 1$
    while $i \leq n$ and $x_i \leq x_1 + 1$:
        $i = i + 1$
    if $i == n + 1$:
        return $\{x_1\}$
    else:
        return $\{x_1\} \cup$**Marks**$(x_i, \ldots, x_n)$

Greedy algorithms and dynamic programming:
In the *pretty printing* problem, you are given a list of lengths of words, positive real numbers $w_1, w_2, ..w_n$,

that must be printed in order, but can be broken up into lines in different ways with a maximum page width of M characters.

For Example: Suppose the words are "The lazy brown fox jumped over the fence". If all characters have the same size, and we don't count spaces, we'd have the list $3, 4, 5, 3, 6, 4, 3, 5$. Say $M = 10$. Then we could break them up as

T	h	e	L	a	z	y	_	_	_
B	r	o	w	n	F	o	x	_	_
J	u	m	p	e	d	O	v	e	r
T	h	e	F	e	n	c	e	_	_

The *slack* of a line is the amount of empty spaces left over. In the above example, the slacks are $3, 2, 0, 2$.

The penalty of a placement is the sum of all slacks. For the above example, the penalty would be $3 + 2 + 0 + 2 = 7$.

1. Fill out the rest of this solution specification: (4 points)
   - Instance: List of word lengths $w_1 ... w_n$ with $w_i > 0$ and $M \geq \max(w_i)$.
   - Solution Format: Sequence of indices: $(i_1, i_2, \ldots, i_k)$ so that Line 1 consists of the words $(w_1, \ldots, w_{i_1})$, Line 2 consists of the words $(w_{i_1+1}, \ldots, w_{i_2})$, ..., Line $k+1$ consists of the words $(w_{i_k+1}, \ldots, w_n)$.
   - Constraints:

     Solution:
     The total length of all words in a single line does not exceed M
   - Objective:

     Solution:
     Minimize the penalty.
2. **Candidate Greedy Strategy:** Place words on the first line while the total is less than M. Then move on to the second line, and repeat until all words are assigned lines.

   Prove that this greedy strategy is optimal

   Solution:
   exchange argument:
   Let OS be some solution that doesn't fill in the first line as much as possible. Then let w be the first word in the second line. Create OS' by moving w from line 2 to line 1.
   OS' is valid: Since the top line of OS was not filled entirely, we can add w to the top line and it is still less than M. Since OS was valid, all other lines are less than M.
   OS' is just as good as OS: Let's say that the penalty of OS is P. Then taking w away from line 2 will increase the penalty by $|w|$ and adding w to line 1 decreases the penalty by $|w|$. So the penalty of OS' is also P.

Dynamic Programming 1: Define the *quadratic* penalty function to be the sum of squares of all the slacks. For the above example, the quadratic penalty function would be $3^2 + 2^2 + 0^2 + 2^2 = 17$. Give a counter-example to the **Candidate Greedy Strategy** when the objective is to minimize the quadratic penalty function.

Give a dynamic programming algorithm for the pretty printing problem that minimizes the *quadratic penalty function* based on the following recursive algorithm: (This algorithm tests every possible way to fill in the last line and updates the MinPenalty as it goes.)

Subproblems: Let $A[i]$ be the minimum quadratic penalty only using words $w_1, \ldots, w_i$.
Base Cases: $A[0] = 0$
Recursion: The case analysis will be on which words are in the last line.
Case 1: w_i is the only word on the last line:

$$A[i] = (M - |w_i|)^2 + A[i-1]$$

Case 2: w_{i-1}, w_i are the only words on the last line.

$$A[i] = (M - |w_i| - |w_{i-1}|)^2 + A[i-2]$$

...
Case k: $w_{i-k+1}, \ldots, w_i$ are the only words on the last line and You can't possibly fit any more words.

$$A[i] = \left(M - \sum_{\ell=i-k+1}^{i} |w_\ell|\right)^2 + A[i-k]$$

Then we take the minimum over all cases:

$$A[i] = \min_{j=1,\ldots,k} \left[\left(M - \sum_{\ell=i-j+1}^{i} |w_\ell|\right)^2 + A[i-j]\right]$$

where k is the maximum number of words you can fit in the bottom line.
Ordering the subproblems: Order them from 1 to n.
final form of the output: $A[n]$
pseudocode:
PrettyPrint$(w_1, \ldots, w_n; M)$

1. $A[0] = 0$
2. **for** $i = 2, \ldots, n$:
3. LastLine $= |w_i|$
4. $minPen = \infty$
5. **while** LastLine $< M$
6. $minPen = \min\left(minPen, (M - LastLine)^2 + A[i-j]\right)$
7. LastLine = LastLine $+ |w_{i-j}|$
8. **return** $A[n]$

Runtime: The outer loop runs $O(n)$ times and the while loop runs a maximum of n times. So the whole algorithm runs in $O(n^2)$ time.

Dynamic Programming 2 Suppose you have two sequences of positive integers $A[1, \ldots, n]$ and $B[1, \ldots, n]$. A common subsequence of A and B is sequence that is a subsequence of both A and B. (Note: subsequences do not have to be consecutive.) You wish to find the weight of the *heaviest* common subsequence. The common subsequence between A and B that has the maximum sum of entries. For example: If the two sequences are:

$$(1, 2, 1, 2, 30, 1, 2) \qquad \textit{and} \qquad (1, 30, 2, 1, 2, 1, 2)$$

Then the heaviest common subsequence is: $(1, 30, 1, 2)$ with a weight of 34. (Note that this is different than the *longest* common subsequence.)

Subproblems: Let $H[i, j]$ be the weight of the heaviest common subsequence between $A[1, \ldots, i]$ and $B[1, \ldots, j]$.

Base Cases: $H[0, j] = 0, H[i, 0] = 0$

Recursion: The case analysis will be on if $A[i] = B[j]$ and $A[i], B[j]$ are part of the heaviest common subsequence or not.

Case 1: If $A[i], B[j]$ are not part of the heaviest common subsequence then

$$H[i, j] = \max(H[i-1, j], H[i, j-1])$$

Case 2: If $A[i] = B[j]$ and $A[i], B[j]$ are part of the heaviest common subsequence then:

$$H[i, j] = A[i] + H[i-1, j-1]$$

Then if $A[i] \neq B[j]$ then

$$H[i, j] = \max(H[i-1, j], H[i, j-1])$$

If $A[i] = B[j]$ then

$$H[i, j] = \max(H[i-1, j], H[i, j-1], A[i] + H[i-1, j-1])$$

Ordering the subproblems: Order $i = 1, \ldots, n$ in the outer loop and $j = 1, \ldots, n$ in the inner loop.

final form of the output: $H[n, n]$

pseudocode:

HeaviestCS($A[1, \ldots, n], B[1, \ldots, n]$)

```
1. for i = 1, ..., n:
2.     H[i, 0] = 0
3. for j = 1, ..., n:
4.     H[0, j] = 0
5. for i = 1, ..., n:
6.     for j = 1, ..., n:
7.         if A[i] == B[j]:
8.             H[i, j] = max(H[i-1, j], H[i, j-1], A[i] + H[i-1, j-1])
9.         else:
10.            H[i, j] = max(H[i-1, j], H[i, j-1])
11. return H[n, n]
```

Runtime: Two nested for loops each running n times and body of loop is constant for $O(n^2)$ time total.