UCSD CSE 101 Section B00, Winter 2015 FINAL EXAM

March 19, 2015

NAME:

Student ID:

Question	Points	Score
1	25	
2	10	
3	10	
4	10	
TOTAL	/50	

INSTRUCTIONS. Be clear and concise. Write your answers in the space provided. Use the backs of pages, and/or the scratch page at the end, for your scratchwork. All graphs are assumed to be simple (no self-loops or multiple edges). For convenience, the scratch page at the end gives pseudocode of Next-Fit and also states the Interval Scheduling problem. Good luck!

You may freely use or cite the following subroutines from class:

- dfs(G). This returns three arrays of size $|V|$: *pre*, *post*, and *cc*. If the graph has k connected components, then the *cc* array assigns each node a number in the range 1 to k
- bfs(G, s), dijkstra(G, l, s), bellman-ford(G, l, s). Each of these returns two arrays of size $|V|$: *dist* and *prev*.

Question 1. Short Answer.

(a) BFS. *(2 points)* BFS is executed on the following undirected graph beginning at s, breaking ties in lexicographical order.
What is the order in which vertices are dequeued from the queue?

S, A, C, B, D, E

(b) MST. *(3 points)* Determine whether the following statement is correct or incorrect. If it is correct, provide a short proof of correctness. If it is incorrect, provide a counterexample to show that it is incorrect. Assume that the graph $G = (V, E)$ is undirected and connected. Do not assume that all the edge weights are distinct unless explicitly stated.

A lightest edge in a cut is present in all minimum spanning trees of G.

True by the cut property.

(c) MST. *(3 points)* Determine whether the following statement is correct or incorrect. If it is correct, provide a short proof of correctness. If it is incorrect, provide a counterexample to show that it is incorrect. Assume that the graph $G = (V, E)$ is undirected and connected. Do not assume that all the edge weights are distinct unless explicitly stated.

If G has more than $|V| - 1$ edges, and there is a unique heaviest edge e, then e cannot be part of any minimum spanning tree of G.

False. That edge may be the only edge connecting some vertex to the graph.

(d) DFS. *(1 point)* A directed graph G contains an edge $u \to v$. When you run DFS on the graph, you find that u has a *pre* value of 4 and a *post* value of 5, and v has a *pre* value of 2 and a *post* value of 7. What type of edge is $u \to v$?

Back edge

1 pt.

(e) Dijkstra. *(2 points)* Dijkstra's algorithm is executed on the following directed graph, starting from vertex A and breaking all ties in lexicographic order. What is the weight of the first finite path calculated by the algorithm from vertex A to vertex G? (i.e., when Dijkstra's algorithm first updates the value of the estimated shortest $A - G$ path distance from ∞ to a new finite value, what is the new value?)

7 is the length one first path found ($A \to B \to G$).

$7 \Rightarrow 2$

$5 \Rightarrow 1$

o.w. $\underline{\underline{0}}$

(f) Reduction. *(3 points)* Consider the following two problems.

HAMILTONIAN CYCLE (HC)

Input: A graph $G = (V, E)$.
Decision Problem: Does G contain a simple cycle that visits each vertex in V exactly once?

TRAVELLING SALESMAN PROBLEM (TSP)

Input: A set of n cities, (symmetric) pairwise distances between these cities $d(i, j) = d(j, i)$, $1 \leq i < j \leq n$, and an integer $k > 0$.
Decision Problem: Is there a tour over the n cities (i.e., a simple cycle that visits each city exactly once) with length $\leq k$?

Show how to make an HC decider, given the existence of a TSP decider.

(1) Describe the transformation of an arbitrary instance of HC into one or more instances of TSP.

Make each vertex in the input graph G be a city in the TSP problem formulation. For every pair of cities, if there is an edge between their respective vertices in the HC problem, make the distance between them be the length of that edge. If not, make the distance between them ∞. Then, we can set k to be the sum of the edges in G. That way, if there is a tour with length $\leq k$, then there must be a Hamiltonian Cycle.

(2) Describe the transformation of one or more yes/no outputs by the TSP "decider" into the correct yes/no output from the Hamiltonian Cycle "decider"

If formulated as stated above, if the TSP returns yes, then there is a Hamiltonian cycle in G.

(g) Approximation. *(3 points)* Suppose you are given an unlimited number of bins of capacity C and a set of items $s_1, ..., s_n$ with weights $w_1, ..., w_n$, and you wish to find the minimum number of bins needed to contain all of the items such that no bin contains a total weight greater than C. Recall from discussion that the Next-Fit algorithm takes the items in their given order and puts each item into the current bin if it fits within the weight limit. If the item does not fit, then the algorithm closes off the current bin (and never touches that bin again) and moves on to the next bin. We know that Next-Fit is guaranteed to produce a solution that uses less than twice the optimal number of bins. Give an example of an ordered series of item weights such that if the capacity of each bin is $C = 400$, the Next-Fit algorithm will return a solution that uses nine (9) bins whereas the optimal solution uses five (5) bins.

There are many possible solutions. For example,
$[200, 1, 200, 1, 200, 1, 200, 1, 200, 1, 200, 1, 200, 1, 200, 1, 200, 1]$

(h) DP. *(2 points)* Given a sequence of n real numbers, $a_1, a_2, \dots, a_n$, a DP algorithm that finds the maximum sum of any contiguous (i.e., consecutively occurring) subsequence of elements in the given sequence is:

$$M(j) = \begin{cases} a_j. & \text{if } j = 1 \\ \max\{M(j-1) + a_j, a_j\} & \text{otherwise} \end{cases} \tag{1}$$

(In this recurrence, $M(j)$ is the maximum sum of any contiguous subsequence of elements found within the first j elements of the given sequence.) What is the worst-case running time of the given algorithm?
$O(n)$ because there are n subproblems, each with a constant amount of work required to solve it.

+2

O(?) +1

(i) Recurrence. *(2 points)* Consider the following pseudocode for a program that takes as an input a list of positive integers $[a_1, ..., a_n]$.

```
define foo([$a_1, ..., a_n$])
    print 'Hello'
    if $n \le 1$:
        return
    foo([$a_1, ..., a_{n-1}$])
    foo([$a_2, ..., a_n$]))
```

Write out the recurrence relation for the number of times $T(n)$ that this program prints "Hello", as a function of n. Then, give the closed-form solution for $T(n)$.

$2^{n-1} \to +1$

$2^n + 1 \to +2$

$2T(n-1)+1$ $\quad +1$ if missing, check $\underline{T(n)}$

$$T(n) = 2T(n-1) + O(1)$$

And the closed form is :

$$2^n - 1/$$

~~+4~~ $+2$,

(j) Greed Counterexample. *(2 points)* Give an example for which the "earliest start time first" greedy approach to interval scheduling does not return an optimal solution.

There are multiple possible solutions. For example, for jobs with starting and ending times [1, 7],[2, 4], and [5,6]. it would choose to schedule [1,5] instead of [2,4] and [5,6].

(k) SCC-Finding. *(2 points)* To find a *sink* strongly-connected component in a given directed graph G, we should perform *explore(v)* in G, starting from vertex v that satisfies: v has the { highest, lowest } { pre, post } label in a DFS traversal of { G , G^R }. (Please circle the three correct choices.)

HIGHEST, POST, G^R

−1 if wrong

Question 2. Dynamic Programming.
(10 points) You work for a charity and are trying to collect bread rolls to give to homeless people. There are two bakeries that each produce bread rolls at a constant rate. Both of these bakeries have a policy that dictates that all unused bread rolls must be thrown out at the end of every hour. Each bakery is happy to donate to your charity the rolls that would have been thrown out, if you are there to collect them at the beginning of the hour.

You must be at a bakery in order to collect rolls (otherwise they will be thrown out), and it takes exactly 90 minutes to travel between the two bakeries (so, if you choose to travel from one bakery to the other, you will miss the opportunity to collect one hour's unused bread rolls from either bakery).

You are given information on how many rolls each bakery will have left over for each of the hours 1 to n. Bakery A will have a_i rolls left over at the end of hour i, and bakery B will have b_i rolls left over at the end of hour i, for $1 \leq i \leq n$.

Design an efficient algorithm to determine the maximum number of rolls that you can obtain from the two bakeries over the n-hour period. You may start hour 1 at either bakery.

(a) Description of subproblems in English
For any hour, you may choose to either stay at your current bakery and receive the number of rolls that it will give out, or you can use your time to move to the other bakery, in which case you will miss an hour's worth of bread rolls but end up at the other bakery, subsequently free to stay there for as long as you would like. Since each bakery will have a different number of bread rolls left over each hour, we need to know the hour number in order to make our decision about which action to take. In addition, we need to know which bakery we are currently at. Therefore, those will be our two parameters.
For ease of readability, I will say that $A = 0$ and $B = 1$, so that we can pass in A and B as arguments instead of 0 and 1. We will store our answers in a $n \times 2$ matrix, with each column corresponding to one bakery. Alternately, we can just store solutions in a hashmap and directly set A and B as parts of the key. Thus, our recurrence for when we start at bakery A is

$$maxRolls(i, A) = max(a_i + maxRolls(i+1, A), maxRolls(i+2, B))$$

. Our recurrence for when we start at bakery B at time i is

$$maxRolls(i, B) = max(b_i + maxRolls(i+1, B), maxRolls(i+2, A))$$

. Since we are allowed to start at either A or B, the overall problem we are trying to solve is $max(maxRolls(1, A), maxRolls(1,$
The base cases are that it is always better to stay at the same bakery for hour n because there is no next hour from which to get rolls, so you just take the rolls from whichever bakery you are current situated at.

(b) Recurrence

$$maxRolls(i, A) = max(a_i + maxRolls(i+1, A), maxRolls(i+2, B))$$
$$maxRolls(i, B) = max(b_i + maxRolls(i+1, B), maxRolls(i+2, A))$$

.

(c) Proof of Correctness
At any hour, your only two options are staying at the same bakery or moving to the other bakery but missing an hour's worth of bread rolls. This algorithm does an exhaustive search on all of the possibilities, so the answer it returns must be optimal.

(d) Running Time Analysis
There are $2n$ problems, each requiring a max over 2 numbers. Thus, the overall running time is $O(n)$

Question 3. Greed.
(10 points) Suppose that you are given n real numbers $x_1, ..., x_n$ (i.e., these are n numbers on the real number line). You may *not* assume that the numbers are given to you in sorted order.
Design an efficient algorithm that returns the smallest set of *unit-length intervals* (for example, $[2.8, 3.8]$) that encloses every one of the n numbers. You may assume that an interval contains both of its endpoints.

(a) Algorithm Description in English
Sore the numbers (it doesn't matter if you sort them in ascending or descending order). Iterate through the sorted numbers. For every number, if it is not enclosed in an interval, create a unit interval starting (or ending, depending on how you sorted the numbers) at that number.

(b) Proof of Correctness
Intervals must contain all of the numbers, because if we find a number not contained in an interval we create a new interval containing it.
We will use the exchange method to prove the correctness of this algorithm. Assume for contradiction that there exists an 'optimal' solution that uses fewer intervals than our greedy solution. Consider the intervals of each solution in increasing order of starting point.
Since we assumed the optimal solution is strictly better than our greedy solution, then at some point it must have some interval not contained in the greedy solution. Consider the first interval of each solution. The greedy solution's first interval must either be the same as the optimal solution's first interval or it must start later than the optimal solution's first interval, since it starts exactly at the first number, which is the latest an interval can start while still containing that number. This also means that it is possible for the greedy solution's first interval to contain some x_i that the optimal solution's first interval does not, because we know the greedy solution's first interval starts at the first number (meaning it does not miss any earlier numbers) and ends later than the optimal solution's first interval. This means that gap could contain one of the numbers. This logic also applies if the first interval in the optimal solution but not in the greedy solution is not the first interval.
After the first different interval, the greedy solution can only have the same number of numbers covered as the optimal solution or more numbers covered. Furthermore, the optimal solution can never surpass the greedy solution, because it has either fallen behind (and thus is forced to start the next interval earlier than the greedy solution's next interval) or it can at best match the greedy solution's next interval. This is because the greedy solution will choose the starting point of the next interval to be the starting point. The same logic as before applies, and the optimal solution can only fall behind or choose an interval that encloses the same number of numbers. Therefore, it is impossible for the optimal solution to over surpass the greedy solution, so the greedy

(c) Running Time Analysis
Sorting the numbers takes $O(n\ log\ n)$ time. After that, you just need to iterate through each number and check if it is in the previous interval. That will take $O(n)$ time. Thus, the overall running time is $O(n\ log\ n)$.

Question 4. Maximum Flow / Linear Programming.
(10 points) Consider the following flow network, in which the edge labels indicate the edge capacities.

(a) *(4 points)* Run the Ford-Fulkerson algorithm on the above flow network to determine the maximum s-t flow. Draw the **final** residual graph below.

$-2 \rightarrow$ flow many error

-2

-3

wrong flow value -2

(b) *(2 points)* Is it possible to increase the maximum s-t flow to 18 by increasing the capacity of exactly one edge? If so, which edge? If not, why not?

(c) *(4 points)* Write down a linear program whose solution gives the value of the maximum s-t flow in this flow network.

Use this page for scratch work.

```
define nextFit([$a_1, ..., a_n$])
currentBin = first bin
numBins = 1
for item in [$a_1, ..., a_n$]:
  if item fits in currentBin:
    currentBin.add(item)
  else:
    currentBin = new Bin
    numBins += 1
return numBins
```

INTERVAL SCHEDULING PROBLEM

Input: A set of n intervals, $[[s_1, e_1], ..., [s_n, e_n]]$, each consisting of a starting time and an ending time.
Problem: What is the largest number of intervals that can be chosen (i.e., scheduled) without overlap?