CSE 101 Practice Final Questions: SOLUTIONS

For each problem, an example of a minimal acceptable answer is given, to give you an idea of how much detail is needed on the final. A more detailed explanation follows this for some problems. In some cases, there are different answers that would also be acceptable.

1. **(a)** 0. Explanation: The algorithm checks whether any of the numbers from 2 to the square root of x divide x evenly; if any do it returns 0 because the number is not prime. Otherwise the number is prime and it returns 1; so algorithm P is a primality checker (it gives the wrong answer for $x = 2$, but that is irrelevant to this problem). 15 is not prime, so P returns 0. In the following part, 17 is prime, so P returns 1.

   (b) 1

   (c) Let P_n be the largest prime number representable with n bits. The worst-case running time is then $\Theta(\lceil\sqrt{P_n}\rceil)$. Explanation: The worst-case values of x are prime, since those force the loop to go through all iterations without terminating prematurely. So the worst-case n bit number is the largest prime P_n using n bits. The loop on lines 2-3 does a constant amount of work (assuming mod takes constant time, which should have been stated explicitly in the problem, and would be stated explicitly if a similar problem came up on the final), for each of $\lceil\sqrt{P_n}\rceil$ iterations, giving the running time above. Note that P_n will be very close to $2^{n/2}$, so an answer like $\Theta(2^{n/2})$ would also be accepted for this problem.

   (d) No. Explanation: P_n is very close to 2^n, so the running time is exponential, not polynomial. More technically, there are an infinite number of primes of the form $2^n - 1$, giving an infinite number of values of n for which the running time actually is $\Omega(\lceil 2^{n/2}\rceil)$; exponential in $n/2$.

   (e) $\Theta(1)$. Explanation: The best-case n bit numbers are the even ones, for which the loop terminates after doing a constant amount of work in its first iteration, and finding that the number is divisible by 2.

2. In the pseudocode below, let the data structure (X, Y) with arrays X and Y of equal length indicate sort keys $X[i]$ with corresponding data $Y[i]$ that should be carried along with $X[i]$ during sorting (so if $X[i]$ gets put into position j of X with $j \neq i$ during sorting, $Y[i]$ also gets put into position j of Y).

   The algorithm returns an set A of activity indices, as was the case for GREEDY-ACTIVITY-SELECTOR.

   SCHEDULE-CLASSES(S,F)
   COUNTING-SORT((F, S),(F', S'),17)
   $A \leftarrow$ GREEDY-ACTIVITY-SELECTOR(S',F')
   return A

   Explanation: GREEDY-ACTIVITY-SELECTOR takes $O(n)$ time, but requires that its input be sorted by finishing time. Ordinarily this would take $\Omega(n \lg n)$ time, violating the constraint of $O(n)$ time for the overall algorithm, but in this special case the finishing times come from a very limited set of integers, and therefore can be sorted with counting sort in $O(n)$ time. 17 is the maximum value of the finishing times that are being sorted.

3. **(a)** $\Theta(n^2)$. Explanation: Let $T(n)$ be the time required for an array of n integers. SELECT and PARTITION on lines 2-3 require $\Theta(n)$ time (shown in class). Since x is the single smallest element of the array, $q = 1$ after line 3. So line 4 makes a recursive call on a single element (taking constant time), and line 5 makes a recursive call on $n-1$ elements (taking $T(n-1)$ time). The overall recurrence is $T(n) = T(n-1) + \Theta(n)$, which was shown in class (and in the textbook) to have solution $T(n) = \Theta(n^2)$.

   One difficulty here: the problem statement should have explicitly said that all elements in A are distinct, as we have usually assumed when talking about QUICKSORT. If all elements are identical, then there is no unique smallest element; x is simply equal to every element. In this case, PARTITION actually partitions the array into two equal-sized halves, yielding a running time of $\Theta(n \lg n)$. Such an answer wouldn't be required, since we didn't discuss QUICKSORT with equal keys in class - but if you gave this answer and understood that this best case would come up with all equal-sized keys, that is also fine.

   (b) The problem of sorting is in P, and is also in NP. Explanation: since we have algorithms (such as this one) to sort in polynomial time (n^2 is a polynomial, and $n \lg n$ is $O(n^2)$ so is also considered to be "polynomial time"), the problem is in P. Everything in P is also in NP, so the problem is also in NP.

4. **(a)** EAT-CAKE(n,Y,W,K)

```
1  t ← 0
2  for i ← 1 to n
3      do if t + W[i] < K
4          then A[i] ← 1
5               t ← t + W[i]
6          else A[i] ← (K − t)/W[i]
7               t ← K
8  return A
```

Explanation: Our greedy algorithm eats cakes in order of yumminess value per kilogram (as suggested in the problem statement, since they are given to us sorted in this order). For each successive cake, if there is room for the entire cake in our stomach (checked on line 3) then we eat the entire cake (line 4) and increment our total weight eaten by the entire weight of this cake (line 5). If there is not room for the whole cake, then we eat as much of it as we can (line 6; note that we need to compute the *fraction* of the cake we are eating, not the *weight* we are eating). Note that, on line 6, we might be eating zero of the cake if our stomach is already full. Line 7 updates our total weight to record that our stomach is full. Line 8 returns the array of fractions of each cake eaten.

Note that, since the cakes are already sorted by yumminess per kilogram, we never need to access Y.

(b) First, show the greedy algorithm's first choice is part of some optimal solution. The greedy algorithm eats as much as possible (say a fraction f) of cake 1, occupying a total weight $f \cdot W[1]$ and yielding yumminess $f \cdot W[1] \cdot Y[1]$. Assume there is an optimal solution S that only has $f' < f$ of cake 1. Assume (without loss of generality, since S is optimal), that S uses up its entire weight allotment of K kilograms: since $K \geq fW[1]$, S includes at least $(f - f')W[1]$ kilograms of cakes other than 1. Modify S to get S' by removing $(f - f')W[1]$ kilograms of cakes other than 1 from S, and adding $(f - f')W[1]$ kilograms of cake 1. If the weighted average yumminess per kilogram of the removed cake is Y_a, $Y_a \leq Y[1]$ since cake 1 has maximal yumminess per kilogram. If S has total yumminess Y_S, then S' has yumminess

$$Y_{S'} = Y_S - (f - f')W[1]Y_a + (f - f')W[1]Y[1] = Y_S + (f - f')W[1](Y[1] - Y_a) \geq Y_S$$

so S' must also be optimal since it has total yumminess at least that of S. Since S' includes a fraction f of cake 1, the initial greedy choice is a part of an optimal solution.

Now, show that an optimal solution S, that includes a fraction f of cake 1, also includes an optimal solution to the subproblem solved by the greedy algorithm: that of maximizing yumminess consumed from cakes 2 through n, given the remaining weight allotment of $K - fW[1]$ kilograms. Assume that $S[2 \ldots n]$ is not an optimal solution to this problem; that there is a $T[2 \ldots n]$ such that

$$\sum_{i=2}^{n} T[i]W[i]Y[i] > \sum_{i=2}^{n} S[i]W[i]Y[i].$$

Then consider setting $T[1] = f$ to obtain a complete solution to the original problem. Since $\sum_{i=2}^{n} T[i]W[i] \leq K - fW[1]$, $\sum_{i=1}^{n} T[i]W[i] \leq K$, so the weight limit is respected. Say that S has total yumminess Y_S. Then

$$Y_S = \sum_{i=1}^{n} S[i]W[i]Y[i]$$

$$Y_S = fW[1]Y[1] + \sum_{i=2}^{n} S[i]W[i]Y[i]$$

$$Y_S < fW[1]Y[1] + \sum_{i=2}^{n} T[i]W[i]Y[i]$$

$$Y_S < \sum_{i=1}^{n} T[i]W[i]Y[i].$$

So T has total yumminess greater than that of S: S is not optimal, contradicting the original assumption. Therefore any optimal solution S that includes a fraction f of cake 1, may be completed by finding an optimal solution $S[2 \ldots n]$ with a total weight allotment of $k - fW[1]$.

By induction on the number of cakes processed, each step of EAT-CAKE makes a choice that is part of an optimal solution, and then solves a subproblem which will correctly complete the solution. Once all cakes have been processed, a complete optimal solution is returned.

5. Replace line 6 with the following lines:

   6a **if** $k \neq i$ **and** $k \neq j$
   6b **then** $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} \cdot d_{kj}^{(k-1)})$
   6c **else** $d_{ij}^{(k)} \leftarrow d_{ij}^{(k-1)}$

   where $\cdot$ denotes multiplication.

   Explanation: d_{ij}^{k} in this algorithm is the minimum path length from i to j using intermediate vertices numbered at most k. If $k = 0$, it is just the edge weight from i to j. For $k > 0$ under the new, multiplicative score, it is the minimum of the path score from i to j using intermediate vertices numbered at most $k-1$, and the product of the best path scores from i to k using intermediate vertices numbered at most $k-1$ and from k to j using intermediate vertices numbered at most $k-1$ (as computed by the modified pseudocode). If $k = i$ or $k = j$ though, we would be multiplying by 0 and we would get a path length of 0 for every path this way. This is not correct, so we need the **if** above to avoid zeroing d_{ij} when $k = i$ or $k = j$. Note that the original algorithm didn't have to address this special case, since *adding* zero doesn't change anything.

   To see why this is correct for multiplicative scoring (without giving a complete proof), note that, as with additive path score, every subpath of an optimal path P must also be optimal - if not, it could be replaced by a better subpath to get a better path than P, contradicting its optimality.

   Note that, if we allowed negative path weights, we would lose this optimal substructure. A poor path weight (very large) from i to k could get multiplied by a negative path weight from k to j to get an excellent, low (in fact, negative) path weight from i to j; so here a good path could include a very bad subpath (the one from i to k) and this algorithm would not work.

6. Show the problem is in NP by describing a verifier:

   (a) **Certificates:** let certificate y encode a subset V' of the vertices V in the instance.

   (b) **Verifier:** the verifier takes the instance $< G = (V, E), b, c >$ and the certificate's V'. It checks that V' contains b vertices, and that E contains exactly c edges that have both endpoints in V'. If both of these are satisfied, it outputs 1. Otherwise, it outputs 0.

   (c) **Running time:** the verifier loops through all edges in E, doing at most $O(V)$ work for each to check whether the endpoints are in V', for a total of $O(VE)$ time, which is polynomial.

   (d) **If the correct decision on an instance is yes:** the graph contains a subset V' of b vertices with exactly c edges between them. Using this V' as a certificate, the verifier finds b vertices in V' and c edges between them, so it outputs 1.

   (e) **If the verifier outputs** 1 **for some certificate:** say V' is the subset of vertices encoded by this certificate. Then V' has b vertices with c edges between them, so the correct decision is yes.

   Show the problem is NP-hard:

   (a) **Reduce from:** CLIQUE

   (b) **Reduction:** An instance A of CLIQUE consists of a graph G and a clique size k. From such an instance, let the reduction construct an instance B of EDGE-COUNT consisting of the same graph G, a vertex count $b = k$, and an edge count $c = k(k-1)/2$.

   (c) **Running time:** The reduction simply copies the graph, copies k into b, and computes c in constant time, for a total running time that is polynomial.

   (d) **If the correct decision on instance A of CLIQUE is yes:** then the graph G contains a clique V' with k vertices. Since $b = k$, V' has b vertices. There are $k(k-1)/2$ edges in a clique of size k (explanation: k possibilities for the first vertex, $k-1$ possibilities for second vertex, divided by two since direction of the edge does not matter), so there are $c = k(k-1)/2$ edges in G with both endpoints in V'. Thus, the correct decision on instance B of EDGE-COUNT is yes.

(e) **If the correct decision on instance B of EDGE-COUNT is yes:** then the graph G contains a set V' of $k = b$ vertices, with $c = k(k-1)/2$ edges between them. Since these edges are all distinct, and there are only $k(k-1)/2$ edges possible among k vertices, all edges between vertices in V' are present. V' is a clique of size k, so the correct decision on instance A of CLIQUE is yes.

(Explanatory note: most undirected graphs we have seen in this class have not used edges from a vertex back to itself, or multiple edges between the same pair of vertices. These properties are required for this reduction to work, so they should have been given in the problem statement. They would be stated explicitly if such a problem appeared on the final.)

7. Consider $n = 3$ activities. Activity one has start time $s_1 = 1$ and finishing time $f_1 = 5$. Activity two has start time $s_2 = 2$ and finishing time $f_2 = 3$. Activity three has start time $s_3 = 4$ and finishing time $f_3 = 5$.

   The first-starting strategy produces the schedule consisting of just activity one.

   A better schedule consists of activities two and three.