CS 161
Summer 2018
8/9/18

Tutorial 7: Advanced Algorithms
Final Review

Advanced Algorithms

1. **(Warmup with Advanced Algorithms) (Difficulty: Easy to Medium)**

   (a) **T/F** All NP-complete decision problems are also NP-hard, regardless of whether $P = NP$.

   (b) **T/F** The dynamic programming solution for the Traveling Salesperson Problem runs in $O(n^2)$-time.

   (c) **T/F** The approximation algorithm for vertex cover that selects an edge at random, adds its endpoints to the vertex cover, then deletes remaining incident edges to either endpoint is $2 \cdot OPT$.

   (d) 0/1 Knapsack is NP-hard, yet it runs in $O(nW)$-time, where n is the number of items available and W is the capacity of the knapsack. Why does this not prove $P = NP$ even though we have a seemingly polynomial-time solution to an NP-hard problem?

   (e) Give an example in which fixed-parameter tractability is useful.

   (f) How might you rephrase an optimization problem into a decision problem? Concretely, suppose you wanted to solve the single-source shortest path problem. Find a decision version of SSSP, such that a polynomial time solution to the decision version of SSSP can be transformed into a polynomial time solution to the original optimization problem in polynomial time.

Final Review

1. **(Farmville) (Graph Algorithms) (Difficulty: Medium)**

   Suppose there's group of farms that have entered into a cooperative agreement to ensure their collective crops do not perish from drought. They intend to build a network of irrigation pipes to connect the farms, such that each pipe has a direction of flow (i.e. water can only travel in one, fixed direction) and each farm can use water that passes via the pipes through the farm. Each farm would also possess a well that contributes water to any irrigation pipes leaving the farm. Think of this scheme as a means to distribute the risk that certain wells might go dry one year, and others go dry a different year—some years, Farmer Joe takes more water from the network that he contributes, and some years, he supplies more water from his well than he takes.

   (a) After designing the network of directed pipes, the farmers want to ensure that water from every farm can reach every other farm. For example, if there is only one farm whose well is still working, every farmer should still be able to receive a portion of that water via the irrigation pipe network. The only algorithm that the farmers know is BFS, which runs in $O(|V| + |E|)$-time and takes as input a directed graph $G = (V, E)$ and a source vertex $s \in V$, and outputs the set of vertices that can be reached from vertex s. Design an algorithm that can help the farmers check whether the desired condition holds for a proposed pipe network, and uses their BFS algorithm only a *constant* number of times. What's the runtime of this algorithm?

   [We are expecting: An English description of the algorithm and a brief justification of its runtime.]

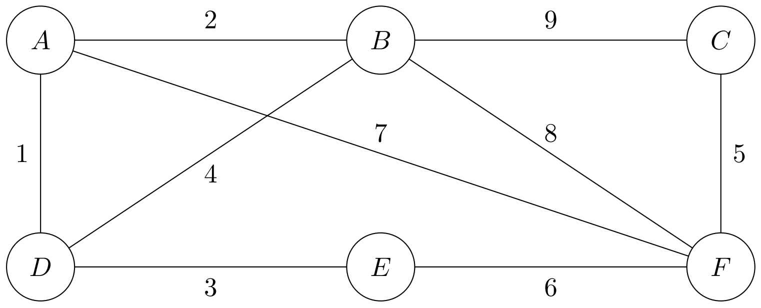
(b) Thanks to your solution, the farmers built a successful network, and enjoy many years of steady crops. One year, however, a turnip gets stuck in an irrigation pipe, clogging that pipe; in the week it takes to fix that pipe, Farmer Joe's potatoes all perish due to lack of water. The farming cooperative forms a new emergency planning committee to design a new, improved irrigation plan. This time, they want a network such that the guarantees of part (a)—namely, that water from any farmer can reach any other farm via the irrigation pipes—will continue to hold even if any single pipe gets clogged. Describe an algorithm for this part that uses your algorithm from part (a), and justify its correctness and runtime.

[We are expecting: An English description of the algorithm, a and a brief justification of its correctness and runtime.]

(c) What is the minimum number of irrigation pipes, as a function of the number of farms, n, that are necessary in any network that satisfies the desired property described in part (b)? Prove your answer—that is, both (1) the claimed number can be achieved and (2) no smaller number is possible.

[We are expecting: A rigorous proof.]

2. **(Prim's vs. Kruskal's) (Graph Algorithms; Greedy Algorithms) (Difficulty: Easy)**
Consider the graph G below.



[We are expecting: For both, just a list of edges. No justification is required.]

(a) What MST does Prim's algorithm return? In what order does Prim's algorithm add edges to the MST when started from vertex C?

(b) What MST does Kruskal's algorithm return? In what order does Kruskal's algorithm add edges to the MST?

3. **(Longest Increasing Subsequence) (Dynamic Programming) (Difficulty: Easy)**

   Let A be an array of length n containing real numbers. A *longest increasing subsequence* (LIS) of A is a sequence $0 \leq i_1 < i_2 < \ldots < i_l < n$ so that $A[i_1] < A[i_2] < \ldots < A[i_l]$, so that l is as large as possible. For example, if $A = [6, 3, 2, 5, 6, 4, 8]$, then a LIS is $i_0 = 1$, $i_1 = 3$, $i_2 = 4$, and $i_3 = 6$ corresponding to the subsequence $3, 5, 6, 8$. (Notice that a longest increasing subsequence doesn't need to be unique).

   In the following parts, we'll walk through the recipe that we saw in class for coming up with DP algorithms to develop an $O(n^2)$-time algorithm for finding an LIS.

   (a) **(Identify optimal sub-structure and a recursive relationship)** We'll come up with the sub-problems and recursive relationship for you, algorithm you will have to justify it. Let $D[i]$ be the length of the longest increasing subsequence of $[A[0], \ldots, A[i]]$ that ends on $A[i]$. Explain why

   $$D[i] = \max(\{D[k] + 1 : 0 \leq k < i, A[k] < A[i]\} \cup \{1\})$$

   [We are expecting: A short informal explanation. It's good practice to write a formal proof, but not required.]

(b) **(Develop a DP algorithm to find the value of the optimal solution)** Use the relationship from part (a) to design a dynamic programming algorithm returns the *length* of the longest increasing subsequence. Your algorithm should run in $O(n^2)$-time and should fill in the array D defined above.

[We are expecting: Pseudocode. No justification required.]

(c) **(Adapt your DP algorithm to return the optimal solution)** Adapt your algorithm above to return an actual LIS, not just its length. Your algorithm should run in $O(n^2)$-time. **Note:** Actually, there's an $O(n \log(n))$-time algorithm to find an LIS, which is faster than the DP solution in this exercise!

[We are expecting: Pseudocode or an English explanation of the algorithm.]

4. **(Summing Elements) (Divide and Conquer; Dynamic Programming) (Difficulty: Medium)**

   Consider the following problem:

   > Let S be a set of positive integers, and let n be a non-negative integer. Find the minimum number of elements S needed to write n as a sum of elements of S (possibly with repetitions).
   >
   > For example, if $S = \{1, 4, 7\}$ and $n = 10$, then we can write $n = 1 + 1 + 1 + 7$ and that uses four elements of S. The solution to the problem would be "4."

   Your friend has devised a divide-and-conquer algorithm to find the minimum size. Their pseudocode is below.

```
def minimumElements(n, S):
  if n == 0:
    return 0
  if n < min(S):
    return None
  candidates = []
  for s in S:
    candidate = minimumElements(n-s, S)
    if candidate is not None:
      candidates.append(candidate + 1)
  return min(candidates)
```

   (a) Prove that your friend's algorithm is correct.

   [We are expecting: A proof by induction. Make sure to state your inductive hypothesis, base case, inductive step, and conclusion.]

(b) Argue that for $S = \{1, 2\}$, your friend's algorithm has exponential runtime—that is, of the form $2^{\Omega(n)}$.

[We are expecting: A short but convincing justification, which involves the recurrence relation that the runtime of your friend's algorithm satisfies when $S = \{1, 2\}$.]

(c) Turn your friend's algorithm into a top-down dynamic programming algorithm. Your algorithm should take $O(n|S|)$-time.

[We are expecting: Pseudocode, and an English description of the idea of your algorithm. You should also informally justify the runtime.]

(d) Turn your friend's algorithm into a bottom-up dynamic programming algorithm. Your algorithm should take $O(n|S|)$-time.

[We are expecting: Pseudocode, and an English description of the idea of your algorithm. You should also informally justify the runtime.]

5. **(Greedy Change) (Greedy Algorithms) (Difficulty: Medium)**

Consider the problem of **making change**. Suppose that coins come in denominations $P = \{p_0, \ldots, p_m\}$ cents (e.g., in th US, this would be $P = \{1, 5, 10, 25\}$, corresponding to pennies, nickels, dimes, and quarters). Given n cents (where n is a non-negative integer), you would like to find the way to represent n using the fewest coins possible. For example, in the US system, 55 cents is minimally represented using three coins: two quarters and a nickel.

(a) Suppose that the denominations are $P = \{1, 10, 25\}$ (aka, the US ran out of nickels). Your friend uses the following greedy strategy for making change:

```
def makeChange(n, P):
  coins = []
  while n > 0:
    # p_opt is the largest coin <= n
    p_opt = max([p if p <= n else 0 for p in P])
    n = n - p_opt
    coins.append(p_opt)
  return coins
```

Your friend acknowledges that this won't work for general P (for example if $P = \{2\}$, then we simply can't make any odd n), but claims that for this particular P it does work i.e. your friend claims that this algorithm will always return a way to make n out of the denominations in P with the fewest coins possible.

Is your friend correct for $P = \{1, 10, 25\}$?

[We are expecting: Your answer, and either a proof or a counterexample. If you do a proof by induction, make sure to explicitly state your inductive hypothesis, base case, inductive step, and conclusion.]

(b) Your friend says that additionally their algorithm should work for any P of the form $P = \{1, 2, 4, 8, \ldots, 2^s\}$.

Is your friend correct for $P = \{1, 2, 4, 8, \ldots, 2^s\}$?

[We are expecting: Your answer, and either a proof or a counterexample. If you do a proof by induction, make sure to explicitly state your inductive hypothesis, base case, inductive step, and conclusion.]