CS 6150: HW 2 – Divide & Conquer, Dynamic Programming

Submission date: Monday, Sep 27, 2021, 11:59 PM

This assignment has 5 questions, for a total of 50 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Basics – recursion and dynamic programming	10	
Linear time selection	10	
Coin change revisited	10	
Let them eat cake	10	
Conflict-free subsets	10	
Total:	50	

Question 1: Basics – recursion and dynamic programming .... **[10]**

(a) [**4**] Consider the following recursive subroutine for computing the kth Fibonacci number:

```
function FIBONACCI($k$):
    if $k = 0$ or $k = 1$ then
        return 1
    else
        return Fibonacci($k - 1$) + Fibonacci($k - 2$)
    end if
end function
```

Implement the subroutine above, and find the Fibonacci numbers for $k = 45, 50, 55$. (You may need to explicitly use 64 bit integers depending on your programming language.) What do you observe as the running time?
The running time for the recursive fibonacci function is exponential.

(b) [**2**] Explain the behavior above, and say how you can overcome the issue.
The recursive implementation of the fibonacci sequence as given in the pseudocode above does repeated work. For example to calcualte the 6th Fibonacci number, F(6) the following information is needed:
$\rightarrow F(6) = F(5) + F(4)$
$\rightarrow F(5) = F(4) + F(3)$, $F(4) = F(3) + F(2)$
$\rightarrow F(3) = F(2) + F(1)$, $F(2) = F(1) + F(0)$
The recurrence relation for the fibonacci sequence can be represented as $T(n) = 3 + T(n-1) + T(n-2)$. For very large fibonacci numbers such as $k = 45$, we end up doing double the work for a calculation. To resolve this issue, we can use dynamic program and memoization where we store the result of previously computed fibonacci numbers which are needed to compute the next fibonacci number. This resolves the time taken to compute a fibonacci number needed to compute the next fibonacci number with a constant time look up operation which improves the overall runtime of computing fibonacci numbers.

(c) [**4**] Recall the L-hop shortest path problem we saw in class. Here, the procedure `ShortestPath(u, v, L)` involves looking up the values of `ShortestPath(u', v, L-1)` for all out-neighbors u' of u. This takes time equal to $deg(u)$, where deg defers to the out-degree.
Consider the total time needed to compute `ShortestPath(u, v, L)`, for all vertices u in the graph (with v, L remaining fixed, and assuming that the values of `ShortestPath(u', v, L-1)` have all been computed). Show that this total time is $O(m)$, where m is the number of edges in the graph.

Question 2: Linear time selection .... **[10]**

Recall the linear time selection algorithm we saw in class (median-of-medians, lectures 4 and 5) and answer the following questions. Please provide detailed justification.

(a) [**5**] Instead of dividing the array into groups of 5, suppose we divided the array into groups of 3. Now sorting each group is faster (although both are constant time). But what would be the recurrence we obtain? Is this an improvement over the original algorithm? [*Hint:* What would the size of the sub-problems now be?]

(b) [**5**] The linear time selection algorithm has some non-obvious applications. Consider the following problem. Suppose we are given an array of n integers A, and you are *told* that there exists some element x that appears at least $n/5$ times in the array. Describe and analyze an $O(n)$ time algorithm to find such an x. (If there are multiple such x, returning any one is OK.) [*Hint:* Think of a way of using the selection algorithm! I.e., try finding the kth smallest element of the array for a few different values of k.]

Question 3: Coin change revisited .... **[10]**

Recall the "coin change" problem, where we have coins of denominations $d_1, d_2, \ldots, d_k$ (an unlimited supply of each), and the goal is to make change for N cents using the minimum *number* of coins (here $N, d_1, \dots, d_k$ are given positive integers).

We saw in class that a greedy strategy does not work, and we needed to use dynamic programming.

(a) [**4**] We discussed a dynamic programming algorithm that uses space $O(N)$ and computes the minimum number of coins needed. Give an algorithm that improves the space needed to $O(\max_i(d_i))$.

(b) [**6**] Design an algorithm that outputs the number of different ways in which change can be obtained for N cents using the given coins. (Two ways are considered different if they differ in the number of coins used of at least one type.) Your algorithm needs to have time and space complexity polynomial in N, k.

Question 4: Let them eat cake .... [**10**]

Alice buys a cake with k slices on day 1. Each day, she can eat some of the slices, and save the rest for later. If she eats j slices on some day, she receives a "satisfaction" value of $\sqrt{j}$. However, a cake loses freshness over time, and so suppose that each passing day results in a loss of value by a factor $\beta = 0.8$. Thus if she eats j slices on day t, she receives a satisfaction of $\beta^{t-1}\sqrt{j}$ on that day.

Given that Alice has k slices at the start of day 1, given the decay parameter $\beta = 0.8$, and assuming that she only eats an integer number of slices each day, give an algorithm that finds the optimal "schedule" (i.e., how many slices to eat on days $1, 2, 3, \dots$). The algorithm must have running time polynomial in k. [*Hint:* dynamic programming!]

Question 5: Conflict-free subsets .... [**10**]

Suppose we have n people, of which we want to pick a subset to form a team. Every person has a (non-negative) "value" they can bring to the team, and the value of a subset is simply the sum of values of those in the subset. To complicate matters however, some people do not get along with some others, and two people who don't get along cannot *both* be part of the chosen team.

Suppose that conflicts are represented as an undirected graph G whose vertex set is the n people, and an edge ij represents that i and j do not get along. The goal is now to choose a subset of the people that maximizes the total value, subject to avoiding conflicts (as described above).

(a) [**3**] Consider the following natural algorithm: choose the person who brings the highest value, remove everyone who is conflicted, choose the one remaining with the highest value, remove those conflicted, and so on. Does this algorithm always find the optimal subset (one with the highest total value)? If so, provide formal reasoning, and if not, provide a counter-example.

(b) [**7**] Suppose the graph G of conflicts is a (rooted) tree (i.e., a connected graph with no cycles). Give an algorithm that finds the optimal subset. (If there are many such sets, finding one suffices.) The algorithm should run in time polynomial in n. [*Hint:* Once again, think of a recursive formulation given the rooted tree structure and use dynamic programming!]