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 was calculated for the following Fibonacci numbers using C++ and the Chronos timing library:

$\rightarrow F(45) = 5$ seconds

$\rightarrow F(50) = 62$ seconds

$\rightarrow F(55) = 697$ seconds

Based on the behavior of the runtimes for these values, we can see that the recursive implementation of fibonacci using the above algorithm produces an exponential running time.

(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. Memoization helps avoid traveling down the entire subtree for a value that is needed to computed the N^{th} fibonacci number. The result is that for each previous fibonacci number we get $2n$ pairs. Assuming a find operation for the memo table is constant time, then the recurrence relation for the memoized fibonacci is $T(n) = O(n+1)$ which simplifies to a runtime of $O(n)$ where n refers to the total number of values needed to produced the n^{th} fibonacci number.

(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.

Based on the problem statement, we can say the following things about the graph:

1. The sum of all the out-degrees for the graph is equal to m where m represents the number of edges in the graph
2. The number of out degrees for a node n can be represented as $1 \leq p \leq m$ where p is the number of outdegrees that is at least 1 and at most m where m represents the total number of edges in the graph.

For every node u in the graph, we look up the shortest path in the cache for all out neighbors u' which can take up $O(m)$ if u has an edge to every other node in the graph. Thus if we assume every node in the graph is connected and the number of nodes $u \leq m$ where m represents the total number of edges in the graph, then we can say that the overall running time is $O(m)$.

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?]

In this problem we partition the array A into $n/3$ subarrays of size 3 ie. the first 3 elements, next 3 elements etc. Next, the suba-arrays are sorted according to the Blum et. al procedure and since they are of constant size, the standard procedure is constant time per sub-array. Then, let $B[0], B[1], B[n/3-1]$ be the array in which B[i] is the middle element, which int this problem is the second element of the ith sub-array. Then we we return the median which would be the n/6 th smallest element of B[]. Modifying the Blum Proof slightly, we can that there are n/6 elements of B[] that are $\leq x$, where x is the approximate median we are looking for. The elements of B[i] are the 2nd smallest elements of some sub-array in A[]. Thus we have at least $\frac{2n}{6}$ elements of A[] are $\leq x$ and we can also argue that there are at least $\frac{2n}{6}$ that are $\geq x$. Then since $\frac{2n}{6} < \frac{n}{4}$ if x is the rth smallest element of A[], then $\frac{n}{4} \leq r \leq \frac{3n}{4}$. The recurrence would be modified to be $T(n) \leq T(\frac{3n}{4}) + T(n/3) + cn$ This would be an improvement of the algorithm because we are subdividing the array into chunks of size $n/3$.

(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 $N, d_1, \ldots, d_k$ 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 $O(\max_i(d_i))$.

Algorithm Correctness

For this problem, we define a fixed size for the tabulation array such that the storage for the table is $max(Coins[d_1, d_2, ...d_n])$. Then since we we only have a fixed size which cannot store the results, once we reach the limits of the array capacity we can start to override the value in the current index because each subproblem who value is stored is based upon the previous value that was computed. To compute this new index we use the modulo which handles the wrap-around. **Runtime**

The space time complexity as proven above in the correctness section is $O(max_i(d_i))$

(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

Algorithm 1 Min Change

```
1: function MIN CHANGE(Coins[], Amount):
Require: Coins[d_1, d_2, d_3, ...d_i] where each d_i corresponds to a coin denomination and the list of coins is in
    ascending order. Amount is a non-negative value
2:     maxDenom ← max(Coins)
3:     table[] ← new array of size max coin denomination
4:     table[0] = 0
5:     for i = 1 to Amount(inclusive) do
6:         minDenom = Amount + 1
7:         for coin in Coins do
8:             Change = i - coin                                  ▷ coin represents coin value
9:             if Change < 0 then
10:                continue
11:            else
12:                minDenom = min(minDenom, table[Change  mod maxDenom] + 1)
13:            end if
14:        end for
15:        table[i  mod maxDenom] = minDenom
16:    end for
17: return table[maxDenom − 1]
18: end function
```

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.
Counter Example:
If we have a group of n people where everyone except for 1 person get along with one another. The person who does not get along with everyone has the highest value forms their own team call this $Team\ A$, and the remaining people form a team together and call this $Team\ B$. If the sum of the $Team\ B$'s values is greater than the total value of $Team\ A$, then we have found a counter-example, for the algorithm always finding the optimal solution.

(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!]