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?

Answer: when k = 45, Fibonacci = 1134903170, when k = 50, Fibonacci = 12586269025, when k = 55, Fibonacci = 139583862445, the running time from k = 45 is kind of fast, but when the k = 50, it is getting slower, then k = 55, it is getting much slower than k = 45, when k is getting bigger, the much slower program I get. The running time is about $O(2^n)$

(b) [**2**] Explain the behavior above, and say how you can overcome the issue.

Answer: T(n) = T(n-1) = T(n-2) + O(1), let's assuming T(n-2) is close to T(n-1), then we get T(n) = 2*T(n-1) + O(1), then next iteration will be T(n) = 2*(2*T(n-2) + O(1)) + O(1), next will become T(n) = 2*(2*(2*T(n-3) + O(1)) +O(1))+O(1)), we can conclude, T(n) = $(2^k) * T(n-k) + (2^k - 1)$, running time is $O(2^k)$, but this is not the tight upper bound. Then we can use Dynamic programming to make it faster, It should be clear that if we already computed $Fibonacci(n-2)$ and $Fibonacci(n-1)$, then we can add them to get $Fibonacci(n)$. Next, we add $Fibonacci(n-1)$ and $Fibonacci(n)$ to get $Fibonacci(n+1)$, We repeat until we reach n = k. This is algorithm when we computing Fibonacci by hand.

(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.

Answer: Because for ShortestPath, it involves looking up the value of `ShortestPath(u', v, L-1))` for all out-neighbors u' of u, which means if u have n neighbors, running time become $O(1) * n = O(n)$ (looking up one edge value take O(1)), neighbors have neighbors, so by using this algorithm, we will eventual go every neighbors reach to last L which means we will go through every edges in this graph, then compare with each total values of edge to see which one is the lowest value that reach to v, this is the shortest path, but still, we need to go through every edge, if we have m edges so that the running time of lookup is O(1), we get O(1) * $m = O(m)$

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?]

Answer: We divided the array into groups of 3, according to the video from professor, the size of sub-problem will be T(n/3), we get $T(n) <= T(3n/4) + T(n/3) + cn$ for some constant c

claim: $T(n) <= \alpha cn^r$ for some constant α and r, because $T(n) <= \alpha cn$ does not work, because this will cause α become negative, so we set $T(n) <= \alpha cn^r$

base case: base case is immediate setting $\alpha = 1$

proof: for the inductive step $T(n) <= \alpha cn^r$ for all $1 <= n' < n$
$T(n) = T(3n/4) + T(n/3) + cn$
$<= (\alpha c(3n/4)^r) + (\alpha c(n/3)^r)$ + cn
$= \alpha cn^r\ ((3/4)^r + (1/3)^r)$ + cn
then we set $(3/4)^r + (1/3)^r = 1$, we get r is about 1.15, so that T(n) $\approx$ O($n^{1.15}$), then compare with professor's answer from video is $T(n) <= 20cn$ which is O(n), then conclude, this is not an improvement over the original algorithm, divide into 3 groups will make slower than groups of 5.

(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.]
Answer: For this question, let us divide this array into 5 pieces, because x that appears at least n/5 times in the array, so I assume divide into 5 piece small is reasonable, if the size of sub array can be divide into equal, then make 5 array size the same, but if sub array cannot be divided into equal size, adjust the last array size for example, I have n = 88, I have 88 integers in my array, then 88/5 = 17.6, the first four array size will be 18, the last array size will be 16, so the size of 5 array will be 18, 18, 18, 18, 16, then start from first element of every array, check if this element = x, if not, check next element, return until that element = x, the worst case of this algorithm, is that x is at last index of the array, which I need to compare with every element in array to see if that element = x, which take n/5 time to find x, so T(n) = O(n/5) + O(1), O(1) is compare running time. so the running time is O(n) with selection algorithm.

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 $O(\max_i(d_i))$.
Answer: So in order to improve algorithm to $O(\max_i(d_i))$, we will need to set up bottom-up dynamic programming instead of top-down.
n is adding up value of coin, initial as 0
MinCoins(N): min number of coins needed for representing N
Base case: if n = N, return "number of coins"
if $i < k$ return "impossible"
for d_j in $d_1, ...d_k$:
if $n < N$
compute 1+ MinCoins(n + d_j)
the for loop is going to take k steps, and among all the answers, take the smallest value.
Our base case is correct because when adding up number n = N, the loop is done, we get the change for N. After k iterations, the N will be adding up. For the space, the original algorithm, because the iteration calls N every time, so the space will be N, but new algorithm, instead of calling N, I call d_j every time which the worst case is max(d_i), so that the space become $O(\max_i(d_i))$

(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.
Answer: This problem can be seen as expansion of shortest path which is find all the solution path to the end, the things different is every vertices, that has k edges, that means, if we have coins of denominations $d_1, d_2, ...d_k$, for example, k = 8, then every vertices that have 8 edges and 8

neighbors. And each neighbors have 8 edges and also 8 neighbors. Then the length of every edge is fixed which is d_1 to d_k
Recursive : FindAllPath(u, N):

variable: count, count number of ways, initial count = 0. u, coins value add up, initial u = 0

Base case : if(u = N, return count + 1)
if ($u > N$, return "impossible")

Iterations:
FindAllPath(u + d_1, N);
FindAllPath(u + d_2, N);
FindAllPath(u + d_3, N);
..... (k times of recursive call)
FindAllPath(u + d_k, N);

Correctness: The base case is when coins is added to N, I set count + 1, count is the number of ways, for each iteration, I consider every possible way of adding coins, which give a k time of iteration, then go to next step, and every neighbor of vertices until adding up to N.
Running time: so in my algorithm, for every iteration, it takes k times recursive call, the worst case is how many times the smallest coins value can add up to N, let us assume d_1 is the smallest coins value, then running time is $O(k^{N/d_1})$ which is polynomial.
Space: k times iteration, space is O(the number of ways$* \max_i(d_i)$)

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!]

Answer:
Recursive: findLargestSatisfaction(k, t)

Variable: t is initial = 1, kArray is initial array from [k,k-1,k-2,...,1], satisfaction is initial = 0, and satisfactionArray, initial = empty

Base case: if(length(kArray) = 0), return max(satisfactionArray).

iteration:
for (int i = 0; $i <$ length(kArray), i++)

if(i = 0)
satisfaction = satisfaction + $0.8^{t-1} * \sqrt{k - kArray[i]}$
satisfactionArray.add(satisfaction)
findLargestSatisfaction(k-i, t+1)

else if (i = 1)
satisfaction = satisfaction + $0.8^{t-1} * \sqrt{k - kArray[i]}$

delete first element in array
satisfactionArray.add(satisfaction)
findLargestSatisfaction(k-i, t+1)

else if (i = 2)
satisfaction = satisfaction + $0.8^{t-1} * \sqrt{k - kArray[i]}$
delete first and second elements in array
satisfactionArray.add(satisfaction)
findLargestSatisfaction(k-i, t+1)

else if (i = 3)
satisfaction = satisfaction + $0.8^{t-1} * \sqrt{k - kArray[i]}$
delete first, second and third elements in array
satisfactionArray.add(satisfaction)
findLargestSatisfaction(k-i, t+1)

else if (i = 4)
satisfaction = satisfaction + $0.8^{t-1} * \sqrt{k - kArray[i]}$
delete first, second, third, and forth elements in array
satisfactionArray.add(satisfaction)
findLargestSatisfaction(k-i, t+1)

.....

else if(i = length(kArray) -1)
satisfaction = satisfaction + $0.8^{t-1} * \sqrt{k - kArray[i]}$
delete all elements in array, which means he eats every cake
satisfactionArray.add(satisfaction)
findLargestSatisfaction(k-i, t+1)

Correctness: The base case is return the maximum value of the satisfaction, I also covered every possible value and add into satisfaction array, so that we can possible compare every possible value.

Running time: the for loop iteration takes k times to iterate, the worst case need to find the value of eat one cake every day, because after first day, if he/she skips couple days without eating cakes, it is impossible to get the best satisfaction because of the decay parameter, so this situation was not even considered, so that my algorithm gives a time of k^2, the running time is $O(k^2)$

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

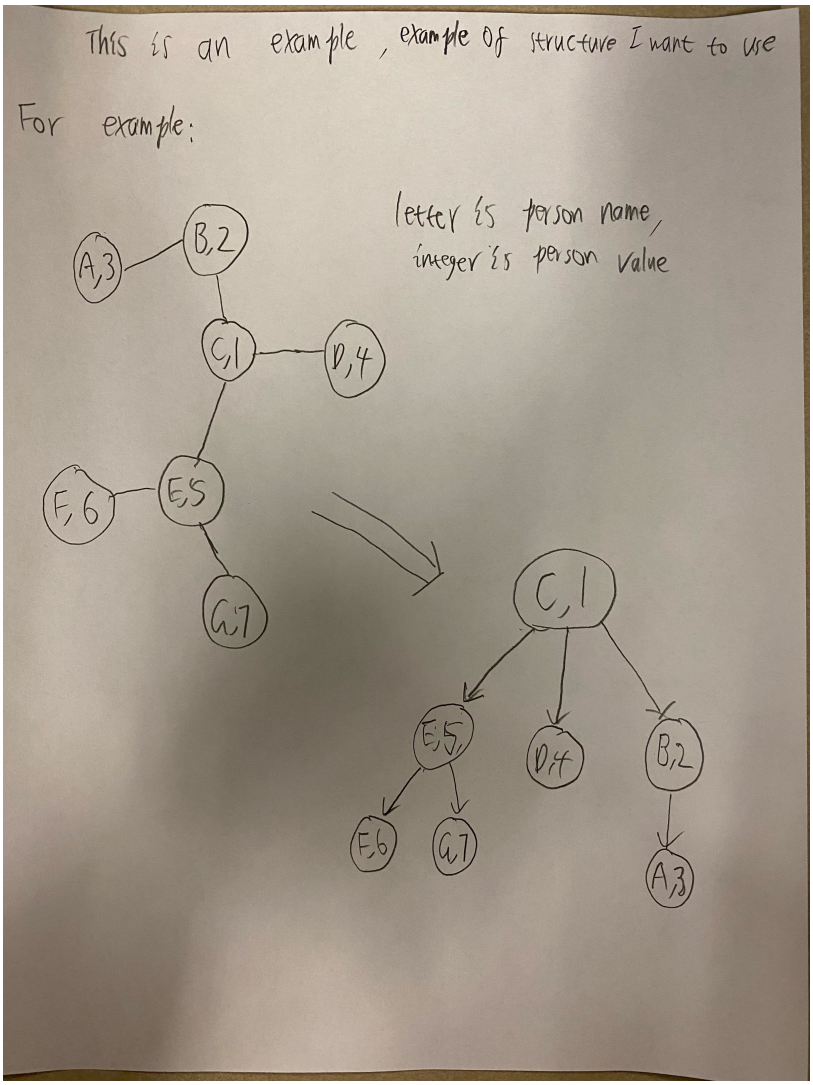
Suppose we have n people, of which we want to pick a subset to form a team. Every person has a (non-negative) "value" they can bring to the team, and the value of a subset is simply the sum of values of those in the subset. To complicate matters however, some people do not get along with some others, 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.
Answer: This algorithm will not always find the optimal subset. I have a counter-example here. For example, we have 3 people, n = 3, A,B,C. People A has value of 5, people A has conflict with people B and C. People B has value of 4, conflict with A, but not conflict with C. People C has value of 3, conflict with A, but not conflict with B. Then we apply this algorithm to this situation, the person with highest value is A, remove B and C, so subset team only have A in it, with value of 5. But the subset team two have B and C that can form a team, B has value of 4, C has value of 3, B + C = 7, so clearly subset with B and C (7) has higher value of subset only A (5).

(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!]
Answer: I attach a hand-write example of the structure I am going to use for this question, in order to give me a better sense of my algorithm. Letter represents the name of person, integer represents the value of the person. Suppose there is a method called isConflict() already exit, because from the question, we already know who is conflict with who. Only people without conflict will be connected together.



class TreeNode:
string id;
int value;
TreeNode parent;
TreeNode[] children;

Since from the question, we can know, the graph G is already changed to a rooted tree, so we dont have to provide the pseudocode code for buildTree, previous is the pseudocode for every node I assume. Starting from graph G is already a rooted tree, which means every node should already have parent and childrens. I can directly use them. The depth from top node to bottom node is k

Recursive: findMaxValue(node n)

Variable: **valueArray**, initial = empty, **totalValue**, initial totalValue = the most top node value

Base case: if(thisNode.children.length = 0) return valueArray.add(totalValue);

Iteration:
for(i = 0, i < thisNode.children.length, i++)
totalValue = totalValue + thisNode.children[i].value ;
findMaxValue(thisNode.children[i]);

Final result:
return maximum value from valueArray;

Correctness: This algorithm, will go through every person, try every possible form with other person, and add their value together, then find the maximum value of those forms. You can pick any random person as the most top node, try to find person along with him/her. And my algorithm will not have any duplicate person in one team. Because my algorithm, only person get along can form into a team.

Running time: Since I go through every person to find the highest value, if there are n people. My running time would be O(n), if there is k people, my running time would be O(k), so basically, running time is O(how many people).