1 Q1: Basics: Recursion and dynamic programming

1.1 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?

Please see attached Homework2.c for implementation (python was too slow, lol)

Running times:

Fibonacci 45: 10.952583

Fibonacci 50: 82.760059

Fibonacci 55: 877.220084

The running time is clearly growing very quickly. I would guess that it's some very high degree polynomial or maybe even exponential from the numbers alone, but this is a classic example of the $(goldenRatio)^n$.

1.2 Explain the behavior above, and say how you can overcome the issue.

The running time is so poor because we're recomputing values over and over. We could overcome the issue by computing each fibonacci number for indices less than n one time and storing the result. This would let us reference the smaller fibonacci numbers when computing other numbers that rely on the answer, instead of recomputing. Essentially, it's better to use dynamic programming with a bottom up approach.

1.3 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.

Given the problem assumptions, we know that all subproblems have been solved and that each node has $deg(u)$ out-degree. The relation between the out degree of all nodes and the number of edges in the network is $count(edges) =$

$sum(outdegrees)/2 where sum(outdegrees)$ is the sum of all $deg(u)$. This holds because each edge has a node on each side so each edge would add 1 to the out degree of 2 nodes. Therefore, if the time it takes to compute this problem for each node is $deg(u)$ for all u, we know that this algorithm will run in $O(sum(out_degrees)) = O(count(edges) * 2) = O(2 * m) \in O(m)$

2 Q2 : Linear time selection

2.1 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?]

The correctness of this algorithm follows the same proof as for the 5 sub array case, but instead of $3n/10$ elements less than or equal to and greater than x (the median of the middle subarray), you end up with $2n/6$ elements less than or equal to and greater than x.

Since x is an approximate median, we still know that it is greater than 25 per cent of all elements and less than 25 percent of all elements. The worst case is that it's exactly on that border and one of our sub arrays contains exactly $3n/4$ elements. We also know that we have to keep recurring on the middle of the 3 arrays if we haven't yet found an approximate median. Thus the recurrence relationship is $T(3n/4) + T(n/3) + cn$ for some constant c.

Allow for some base case where this is immediate, e.g. 1 element, the median is that element. Now lets solve our recurrence. Assume that for all n ¡ N that $T(n) = \alpha cn$ Thus, by inductive hypothesis we must show $\alpha cn \leq \alpha c(3n/4) + \alpha c(n/3) + cn$. After combining terms we are left with $\alpha cn \leq \alpha c(13/12)n + cn$, which does not have a positive integer solution. Thus this method is not faster than the original with 5 parts – it does not run in linear time since it violates our assumption that there is some α that works if it's linear.

2.2 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.]

Using the selection algorithm mentioned above, you can query several spots through the array, A, and see if you find a duplicated value. For example, in the case that the element is repeated $n/5$ times, query spots $0, n/6, 2n/6$, etc. until you find a duplicated value. You're guaranteed to have a duplicate, because we're querying the nth largest every n/6 times and one of the elements is repeated $n/5$ times. Since $n/5$ is bigger than $n/6$ that value will be repeated.

For time complexity, since the selection happens in linear time and we just need to do it some constant number of times, e.g. 7 in the above example, this runs in $O(7 * n) \in O(n)$ time. This algorithm is worse if the element is not repeated so much because we have to search more positions.

3 Q3: Coin change revisited

3.1 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(maxi(di)).

Consider the bottom up case for this algorithm. That is we start with an array of length n+1 and start filling in the lowest number of coins we can use to return the change for an amount equal to the index. E.g. it takes zero coins to return zero cents, one coin to return one cent, etc. The general idea for this algorithm is that for any index in the array, compute the exhaustive search for how you would get to that value, by first subtracting the highest value coin less than the current target, then trying for the next lowest value coin, etc. When you subtract the value of the coin of from the target, you get a value that is in the array already, thus you know the lowest number of coins to get that value. Now add one to that number and store it in the array at the new index. Do the same for the second largest coin and compare with the previous answer, preferring the lowest value. Continue until you're out of denominations This fills the n + 1 size array with all the answers to every possible subproblem and then returns the answer for the last value, which is our target.

The key to keeping the size at $maxi(di)$ is noticing that we never back track

more that the largest denomination in the array, so we can just keep a number of answers in the array equal to that size. E.g. if we have 25c as our largest value, we would only need 25 (maybe + 1) elements in the array to be able to do the subtraction step from above. A visual analogue is to say we're basically sliding the array to the right as we compute the values farther to the right. The correctness of this approach and the run time come from the exhaustive computations in the algorithm above, but the space complexity is now just at $maxi(di)$ since that's the new size of the array.

3.2 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.

Let's work bottom up to create the possible number of ways to generate a specific change value. Our base case is returning N=0 change. There is one way to do that, give no coins.

Let's make an array of length N + 1. We use + 1, because we need to track the base case above. The index of this array represents the value of change returned and the value at that index represents the number of possible permutations we could use to return the value of change. Set all values in the array to zero. Set the zero index value to 1. Now start filling in values, left to right starting with index = 1. If there are some denominations less than or equal to the index number we are at, consider those coins as possible candidates, else set the value to None. For all possible candidates, subtract their value from the current index and lookup the possible ways to make that number of coins. Add this number of coins to the possible ways to make the current value we care about. Do this for all denominations less than the current value.

This algorithm runs in time $O(k * N)$ because for each value less than n we have to subtract at most k coins and add up the values that we already computed. The correctness comes from the fact that this is an exhaustive search with memory of past problems.

4 Q4: Let them eat cake

4.1 Cake problem

Assume a base case of k = 1 slices (a sad day for all involved, only 1 slice...), it is best for satisfaction to eat the slice today, because it will provide more satisfaction than it does tomorrow (since it's guaranteed to be only 80 per cent as good tomorrow). Now continue from the bottom up, building an array that keeps the maximal satisfaction for that number of slices. For example, in the case of

k = 2, there are 2 possible options, eat 2 slices or eat one today and one tomorrow. The two options are $highestSatisfaction(2) = max(satisfaction[1] + 0.8 * satisfaction[1], satisfaction[2])$. In this case satisfaction[1] is known and does not need to be re-computed, whereas satisfaction[2] needs to be computed (simply $\sqrt{2}$). The max of these two options is the first, one slice per day at 1.8 vs. 1.414 in the case of 2 slices today. We keep the 1.8 as the maximum satisfaction for 2 slices and move on. Thus each time we iterate through the array, towards our target of k slices, we are making one additional computation, simply satisfaction[current$_index$], $whichwillbe\sqrt{current_index}$. We'll also need a parallel array that track what the number to eat per day is for the maximal case. Since the total number of new subproblems at each index is only one (plus some comparisons with the other possibilities), this algorithm will run in $O(n)$ time. Correctness follows from the fact that this is another encoding of exhaustive search with memory of solved subproblems.

5 Q5: Conflict-free subsets

5.1 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.

No it does not. Consider the counter example of 4 players with their values: $A = 10, B = 8, C = 8, D = 1$. I also assume teams must be of equal size, 2. Let A conflict with B and C, but there are no other conflicts. In this case the naive algorithm would choose player A (the highest value) and then be stuck choosing D as their other player. This gives a total value of 11. The most optimal choice of team would instead be to choose B and C, since their combined value is 16.

5.2 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!]

Given the tree structure, the recursive formulation for this problem would be to take a node at random, n, (try for all of possible n) and remove all neighbors of n from the candidate nodes for a team. At this point recur and now select

another random node (try them all) and remove it's neighbors. Continue until you have all possible teams and their values. Now choose the one with the highest value.

To transition this solution to a dynamic programming approach, we would take much the same steps as above, in a depth first, top down approach. Given this approach we will already know some of the solutions to the subproblems, so we can reduce the total number of calculations.

The correctness of this algorithm follows from the recursive approach's exhaustive search. We're doing the same search, but not recomputing values.

As for the running time at each step in the problem, we need to consider a number of subproblems equal to the number of remaining candidates minus the degree of the node we're adding. This is because we remove the nodes that are attached to the node we're adding. The worst case running time would be if the conflict graph was a null graph (no edges), although the problem statement enforces it must be a tree – any graph close in number of edges to the null graph would have a poor running time. In these worst cases the subproblems are not much smaller than the original problem since the number of excluded nodes in the next step is equal to or close to 0. In this case you'd need to recur n times. Assuming that each subproblem is solved (or will be eventually solved by recursion down to a base case), this means that for all n nodes in our graph, we'll have to compute n-1 subproblems with an input node array of size n-1. The next case would be n-2, etc. This is $n^2/2 \in O(n^2)$ time.