Haohan Chen Discussion 1D Fri 4-5:50
TA: Vincent

CS180 HW#5

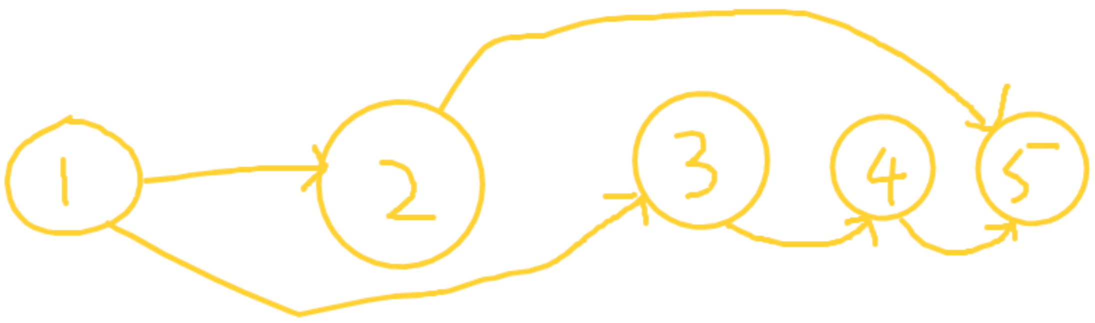
1. Algorithm:
   - Sort the array in ascending order
   - Initiate two pointers with one at the start of the array and one to the right of it
   - Find the difference between the elements pointed to by the two pointers
   - If the difference is equal to the difference, output the two elements and increment both pointers by 1
   - If the difference is less than the target, increment the right pointer by 1
   - If the difference is more than the target, increment the left pointer by 1
   - Repeat this process, until we reach a situation where the desired pointer cannot be incremented(meaning it has reached past the end of the array)

Time Complexity Analysis:
- For sorting the arrays with heap sort takes n log n time.
- Then with our two pointer method, in the worst case scenario, we will increment our pointers O(n) times before our algorithm terminates. So in the worst case scenario we will have to do 2 * O(n) comparisons. Thus, this will be O(n) time.
- Thus our total time complexity is O(n log n)

Proof:
- How our algorithm works is by exploiting the ascending order provided by the sorted array
- We test the difference pointed to by our two pointers and if the difference is smaller than our current target, we know that there is a possibility that we can get our target difference with our current left pointer, so we move our right pointer by one (and thus making our difference larger).
- If the difference is larger, then we that there is no possibility of our left pointer being able to reach the target sum, and thus we increment our left pointer
- In this way, we can “exhaustively” search through our array
- Proof by Contradiction:
  - Say our algorithm missed a pair of numbers that has our target number
  - For that to happen, there can only be two possibilities. 1. Our left pointer never was pointed at the larger number of the pair. Or 2. Our right pointer never was pointed at the smaller number of the pair.
  - For possibility one to happen, our left pointer must have been shifted before our right pointer got to the smaller of the pair of numbers. For that to happen, an earlier pair of numbers with our left pointer had a difference larger than our target. However, if that was the case, then as stated earlier, it is impossible for our left pointer to be one of the pair of numbers, which contradicts our premise.
  - For possibility two to happen, our right pointer must have been shifted before our left pointer got to the greater of the pairs of numbers. For that to happen, an earlier pair of numbers with our target number right pointer had a difference smaller than our target. However, if that was the case, then as stated earlier in the “Proof” section, it is impossible for our right pointer to be one of the pair of numbers, which contradicts our premise.

- By contradiction, our algorithm always produces the correct solution

2.a.



Counterexample shown above. As shown above, according to the algorithm provided, we would go to edge (1,2) then (2,5). However, that is not the longest path possible instead being route (1,3), (3,4), and (4,5).

b.Algorithm:

- Create an array of length n with n being the number of nodes in our graph
- Each node is assigned a specific index on the array with earlier nodes in our directed graph occupying an earlier position in the array
- Set each element of the array to 0.
- Iterating through each index of our array:
    - Go to the node assigned to that index and for each node N that our current node is pointing to(directed edges going out of our current node):
        - If N's element value is less than our current node's element value + 1, then set N's element equal to that value.

Time Complexity Analysis:

- Creating the array and setting the initial values to 0 be take O(n) time(if somehow the programming language we are using don't have the initial value of an integer to be 0)
- Then we iterate through our array(which is equal to the number of nodes in our graph), and for each step of the iteration we must go to every node adjacent to that specific node(which is equal to edges coming out of our node). So that would be n*(Eaj), which is equal to O(V) in the worst case scenario.
- Thus our total time complexity is O(n+V), which in more densely connected graphs, would be O(n^2)

Proof:

- Our algorithm utilizes dynamic programming
- How we formulate our solution is by basically going to a node(we must make sure that it is actually reachable from the first node in our graph), then checking on all nodes it is pointing to with whether traveling in the path set by our current node would lead to a longer path length than what is currently the max recorded for the node we are checking.
- Proof by Induction:

- Basis: In the case of 2 nodes, we would have a longest path of 1. (Technically super base case would be if we only have 1 node, but that is quite trivally a length of 0)
- Induction Step: We assume for the R-th node, we have the longest path it takes to travel to the R-th node
- For the R+1-th node, we can always find the longest path available as in all previous iterations, we would've already accounted for every possible path that could travel to the R+1-th node and we check the length of each one and keep track of the maximum. Thus we can always find the longest path
- By induction, our algorithm always returns the longest path possible.

3. Algorithm:

- For each character in the string, call the quality function from the first character of the string to our current character of the string. Keep track of the maximum number returned from these function calls
- Whatever character yielded the maximum quality in the last iteration, cut the string at that character and keep track of what character caused it.
- Iterate through our new string and repeat this procedure of calling the quality function and keeping track of the maximum quality found among the calls, before then creating our new string based on the character which created the highest quality. Repeat this until our string is empty
- All characters we kept track of during our iterations will be where we put our partitions.

Time Complexity Analysis:

- Each iteration calling the quality function is O(n) time.
- In the worst case scenario, the entire string is composed of only a single character(ex. "aaaaaa"). If that is the case we will need to iterate n times. Thus our time complexity would be n*O(n). Thus our total time complexity would be O(n^2).

Proof:

- Proof by Induction:
- Base Case: Base case is a string of one character which is trivial as there are no partitions
- We use dynamic programming with us, forming our solution with the observation that once a string of characters hit a maximum quality value along one of its characters, it will decrease in value from that point onwards, so we can safely cut the string at that point to find more words.
- Induction Step: We will assume for a string of length R, that we have formed the optimal location of the partitions for maximum quality
- For the R+1 character, we know from the observation stated two bullets earlier that it must be the start of a new string. Thus with our algorithm, we can continue the procedure of finding the next word and thus guarantee that our partitions stay maximum.
- By induction, our algorithm always produces the optimal partitions.

4. a.

Minutes	1	2	3

A	2	9	9
B	1	10	19

- Our algorithm will choose A for minute 1 (making our total 2). Then for minute 2, our computer would choose to move. Then for minute 3, we would be on computer 3 (thus making our total 21).
- However the optimal solution would be to choose B for minute 1 (thus making our total 1). Then we would stay on B for minute 2 (making our total 11). Then for minute 3 we would stay on B, making our total 30. Thus the algorithm provided did not find the maximum possible total, so by counterexample, the algorithm provided is not optimal

B. Algorithm:

- Create an array of size n. n being the number of minutes we have available.
- Each element of the array at a specific index J represents the largest sum possible at that number.
- For minute 1, pick the element with the largest computing power. For minute 2, put the larger of the sum of the first two minutes between A and B respectively. Keep track of which machine we are currently at for minute 3(so if we put the sum of B for minute 2, we are at B for minute 3).
- Iterating through each minute M, starting at minute 3:
  - First take the processing power P on the machine we are currently at for minute M. Then find the sum of P and the value of element M-1. We will denote this as D_1.
  - Then we take the processing power P_1 of the machine we are currently not at for minute M. Then find the sum of P_1 and the value of element M-2. We will denote this as D_2.
  - Whichever is bigger between D_1 and D_2, that will be our element M.
- Return the last element of the array which will be the maximum number.

Time Complexity Analysis:

- Creating the array would be O(1) time.
- Doing the first two minutes requires O(1) time.
- Then we must iterate through the rest of the array and for each minute the comparisons we do are only O(1) time. Thus the iteration would take O(n) time.
- Thus our total time complexity would be O(n) time.

Proof:

- We used dynamic programming to formulate our solution. At every minute we first check to see if we need to want to stay(add the optimal sum at the minute before with our current machine's computing power at our current minute) or it would've been better to move to the other machine in the last minute(add the optimal sum 2 minutes before to our opposing machine's computing power). We then put the larger of the two for the optimal sum at our current number
- Proof by Induction:
  - Base Case: When we only have one minute, we just choose the larger of the two

- Induction Step: We assume for the R-th minute, we have the maximum sum computing power at that minute (and the sum of every other minute before R with our algorithm).
- For the R+1-th minute, we can always find the maximum sum computing power at that point by using the methodology outlined in the first bullet in the "Proof" section. Thus our algorithm ensures we can always find the maximum sum computing power at any given minute.

5. Algorithm:

- Create a (n+1)x(n+1) array
- Each row will represent the rod length value available to us (so the i-th row would mean we have access to cutting a length of i out to sell). Note, we also have access to all i-1 lengths too.
- Each column would represent the size of the rod available to us(so the j-th column would mean a rod of length j is available to us)
- Element ij would mean the maximum value we can get from selling a rod of length j with values 0-i available to us(trivially, a rod of length 0 is worth 0, and value of length 0 is 0.
- For each row:
    - For each column:
        - Take the (i, j - i) element and to it add the value attached to i. i being the current row we are at. We will call this sum S.
        - The (i, j) element will be the greater number between S or (i-1, j).
        - Note: if (i, j-i) element doesn't exist (meaning it goes off the parameters of our array), we just put the (i-1,j) in for our current element.
- Return the value of the (n+1, n+1) element.

Time Complexity Analysis:

- The setup before our iteration(creating the array) takes O(1) time.
- For each element in our 2d array (n+1xn+1 dimension), we will do O(1) comparisons. Thus the iteration will take O(n^2) time total as we iterate through each element of the array
- Our total time complexity will be O(n^2).

Proof:

- How our algorithm works is by us iterating through the values of the rod while also iterating through the length of the rod 1 by 1. In other words/terms, we form the optimal solution by asking if whether the (max value of the rod minus the current rod length corresponding to the value we are currently at) + (current rod length value) is greater than what we had as the max value for the rod if we didn't use our current rod length value(which in our 2d array would be the (i-1,j)th element).
- Proof by Induction:
    - Basis: Base case is when we are iterating over the value we get for a rod of length 0 which means no matter the size of the rod, we only be able to get a value of 0 attached to it.
    - Induction Step: We assume for the R-th value attached to a rod of length R, we have the maximum value possible for each length of the rod should we sell it all.

- ○ Thus for the R+1-th value attached to a rod of length R+1, we can always find the maximum value we can get for that rod via the methodology stated by the first bullet in the “Proof” section.
- ○ So by induction, our algorithm can always find the maximum possible value we can get for any given rod.

6.
Algorithm:

- Create a 2d array with dimensions of nxn. With n being how many coins there are
- Each index along the row and column represents a coin in the sequence. These are assigned ascendingly, so index 1 would be coin 1, and index 3 would be coin 3. Similar reasoning follows the indexing for the columns.
- Each element in the 2d array will represent the maximum value player 1 could get with coin subsequence composed of the coins from V_i to V_j with i being the current row and j being the current column.
- Along the main diagonal meaning elements (i = j), mark those equal to the i-th coin
- For elements (i, i+1), it is simply the maximum of the the two coins in the subsequence
- Iterating through the 2d array with starting with the i+2-th column in each row:
    - The element at that specific location will be the maximum of the following two values:
    - 1. Value of coin i + the minimum value between element (i+2, j) or element (i+1, j-1).
    - 2. Value of coin j + minimum value between element (i+1, j-1) or element (i, j-2)
    - Whichever is the maximum will be the element at that specific location
- Return the value at element (0, n-1).

Time Complexity Analysis:

- First creating our 2d array will be O(1) time
- Going through the long diagonal to set all the values equal to the row’s coins would take O(n) time.
- Then going to the (i, i+1) elements would also take O(n) time.
- Then going through the main iteration, we go through each row and for each column starting at the i+2 column. This would be bounded by $O(n^2)$
- Thus our total time complexity would be $O(n^2)$.

Proof:

- We have two options at each turn of the game. First, we could either take the coin at the beginning of the current subsequence, or the end
- In either case our opponent will always try to leave us with the minimum score we could possibly get after their turn.
- This in turn will mean that if we take the first coin in our current subsequence:
    - Then our max value will be the sum of the value of first coin V_i, and the minimum score we could get after our opponent takes our turn V_{min}.

- V_{min} could be composed of a subsequence in which our opponent takes the first coin in their subsequence(coins V_{i+2} to V_j) or the last coin in their subsequence(coins V_{i+1} to V_{j-1}).

- If we take the last coin in our current subsequence:
  - Then our max value will be the the sum of the value of the last coin V_j and the minimum score we could get after our opponent takes our turn V_{min}
  - V_{min} could be composed of a subsequence in which our opponent takes the first coin in their subsequence(coins V_{i+1} to V_{j-1}) or the last coin in their subsequence(coins V_i to V_{j-2}).
- We would then take the maximum of the two options available
- This forms our dynamic programming approach to the problem
- Proof by Induction:
  - Basis: We have two base cases. First when we have a subsequence of just one coin, the max score will just be the value of that one coin. Second is when we have a subsequence of just two coins, the max score will be the higher value of the two coins
  - Induction Step: For any given subsequence composed of the 1st to R-th coin, we have the max score possible for that subsequence and every subsequence inside that subsequence too.
  - For any given subsequence composed of the 1st to R+1-th coin, our algorithm will always be able to find the maximum score possible by player 1. This is because, stated at the beginning of the “Proof” section, we know all possible max scores for any possible subsequence no matter which coin we take. And thus since our opponent is also playing optimally, they will leave us with the minimum score out of the possibilities depending on what coin we take. With those possibilities we test the max score possible between taking one coin over the other and thus forming our max score for the 1st to R+1-th coin.