Arthur Zhou

Dis 1A, 10:00am - 11:50am
Parshan

1) • Create a pointer called l (left) pointer at the start of the array and one called r (right) at the end of the array

• While $l < r$: ..

  ◦ If the sum of array[l] + array[r] == k
    ▪ Remove the pair (array[l], array[r]) from the array
    ▪ Move left pointer to the right by 1 and right pointer to the left by 1 to find the next pa

  ◦ If array[l] + array[r] > k:
    ▪ Move right pointer to left by 1

  ◦ If array[l] + array[r] < k:
    ▪ Move left pointer to the right by 1

• Return all pairs

Time / Space Complexity: Since our array is already sorted scanning the array via 2-pointer takes only $\boxed{O(n)}$ time since we only scan the array once. Additionally space complexity is $\boxed{O(1)}$ because we don't need any other significant bits of space, we just need the 2 pointers. This is unlike if we sorted it which would need $O(\log n)$ arrays for space.

Proof: Let's look at the 3 cases for array[l] + array[r]:

1. If sum == k, then we remove the pair and move onto the next values by shifting l to the right by 1 and r to the left by 1. This is self explanatory as we have found a valid pair and we can move both pointers to find more possible pairs
2. If sum > k, we have to move the right ptr to the left by 1, since the array is in a sorted order, this ensures that we decrease the sum since moving the right pointer leftwards shrinks array[r] and therefore sum too. This still ensures that we can find the remaining valid pairs that sum to k
3. If sum < k, we move the left ptr to the right by 1. Since the array is sorted in increasing order, this ensures that we increase the sum since array[l+1] increases from array[l] and therefore increasing our new sum. Still ensures we can still find all valid remaining pairs that sum to k

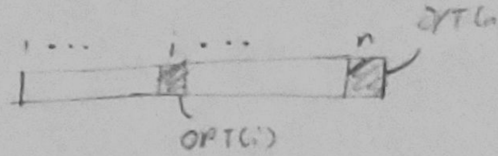
Our algorithm ends after $l >= r$ since at that point we have checked any pair and we know that none of the ~~(?)~~ remaining can hold a valid pair that equals k.

2)

a)
$$\begin{tikzcd} v_1 \arrow[r] \arrow[rr, bend right] & v_2 \arrow[rrrr, bend left] & v_3 \arrow[r] & v_4 \arrow[r] & v_5 \arrow[r] & v_6 \end{tikzcd}$$

This wouldn't work as the algorithm would keep choosing the shortest paths.
Our algorithm would choose (v_1, v_2), and since (v_2, v_6) is the only path from v_2 that path is only 2 edges long while the initially longer path (v_1, v_3) actually nets us a longer path containing $(v_1, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6)$

b) Find OPT(i), or the max. # of paths we can get from v_1 to v_i. Create a 1D array for this DP problem w/ all values set to 0 initially.



- For $1 \le i \le n$
  - For each edge coming into node i from node j,
    - $OPT(i) = \max(OPT(i), OPT(j)+1)$
- Return OPT(n)

Time Complexity: Initializing & traversing the array takes $O(n)$ time bc we have one element per node in our graph. Our For loop traverses all nodes once and all edges once. No calculation of $> O(1)$ were done during these checks. this gives us $O(n+e) + O(n) = \boxed{O(n+e)}$

Proof: Induction:

BASE CASE: the longest path from v_1 to v_1 if $n=1$ is obviously 0

Assume we correctly calculated $OPT(i-1), OPT(i-2), \dots, OPT(1)$

Proof: the max. # of paths at OPT(i) must come from nodes before v_i, and if our solution searches all possible edges, then the max. # of paths must be OPT(i). We do this until OPT(n) where we then return our result

1) Done on paper
2) Done on paper
3) Our goal is to maximize OPT(n), let's call OPT(i) the optimization of the first i letters.
    - Create a 1D array of size n, initialize all values to 0
    - Set OPT(1) equal to quality(y_1)
    - For 2 <= i <= n,
        - OPT(i) is the maximum of
            - OPT(i - 1) + quality(y_i)
            - OPT(i - 2) + quality($y_{i-1}y_i$)
            - …
            - OPT(1) + quality($y_2y_3\ldots y_{i-1}y_i$)
            - quality($y_0y_1\ldots y_{i-1}y_i$) //CAN'T FORGET THIS CASE
        - When the maximum is found, note down what word separations OPT(i) had
    - Return OPT(n) and its separation of words

Time complexity: As we scan through the entire 1D array, the worst possible case for us is if we have to traverse through the entirety of the OPT's while searching for the maximization of OPT(i), leading us to have a time complexity of $O(n^2)$.

Proof: prove by induction
BASE CASE: OPT(1) is easy: you just set the quality(y_1) as that is our only value we have
Assume: That our algorithm calculated OPT(i - 1) correctly and all values beforehand.
Proof: There's only i combinations when a new letter is added to the end of our string-it can be itself, the previous two characters, the previous three, all the way down to a word that somehow could contain all i characters we currently have. As a result we can use this assumption to find OPT(n) and return the separation of words that we need.

4)
    a) If n = 3
    Computer A: $a_1 = 10$, $a_2 = 10$, $a_3 = 10$
    Computer B: $b_1 = 5$, $b_2 = 10$, $b_3 = 21$
    This solution would fail on us as the algorithm would choose a_1, then take the b_3 if statement because it's greater than a_2 and a_3 combined. This would lead the algorithm to getting an answer of 31 processes completed while the true solution would be to stick with computer B the entire time and get 36 processes done.
    b) OPT(i, x) is the maximum amount of processes we can have complete by time i ending at computer x
        - Create a nx2 array where n is the number of minutes and the two others are for computers A and B

- Set OPT(1, A) and OPT(1, B) to a_1 and b_1 respectively
- Set OPT(2, A) and OPT(2, B) to $a_1 + a_2$ and $b_1 + b_2$ respectively, we must do this since we can't look ahead to i + 1 if there are only two values
- For 3 <= i <= n,
    - OPT(i, A) = max(OPT(i - 1, a) + a_i, OPT(i - 2, b) + a_i)
    - OPT(i, B) = max(OPT(i - 1, b) + b_i, OPT(i - 2, a) + b_i)
- Return max(OPT(n, A), OPT(n, B))

  Time complexity: we scan through the entire array once, meaning that our time complexity is O(2n) = O(n). We do a constant number of operations too for each scan so we don't need to multiply anything else

  Proof: prove by induction
  BASE CASE: OPT(1, x) and OPT(2, x) are easily found since we set them manually
  Assume: that our algorithm calculates OPT(i - 1, x) correctly for both computers
  Prove: At time i, we have only two considerations: adding the next value for the same computer or switching computers at the cost of the previous value we had. Our algorithms account for both possibilities for both computers, thus proving that it's true.

5) This is like the knapsack problem but with fabric instead
    - Initialize an n*m 2D array. Let's say the n is the length of the fabric and m are the sizes you can cut the fabric up into and sell.
    - OPT(i, j) is the highest profit we can get by slicing up the fabric that's i inches long to pieces of length j at maximum
        - Set all values in the first row and column to 0 as we cannot sell a piece of fabric 0 inches long nor split the fabric up into no pieces
    - For 1 <= i <= n,
        - For 1 <= j <= m,
            - OPT(i, j) = max(OPT(i - j, j) + the price of fabric length j, OPT(i, j - 1))
    - Return OPT(n, m)

Time complexity: we have to traverse the entirety of the 2D array, meaning that we have to scan n*m elements total. We perform only a constant operation on each element, so we don't get any extra time there. However, our overall performance is O(nm) which gets worse if n and m are close in value.

Proof: prove by induction

BASE CASE: the maximum value when the fabric is length 0 or we have no options to cut it up into should always return 0, hence that's why the first row and column are both 0
Assume: the problem calculated the maximum profit for OPT(i - 1, j - 1), OPT(i - 1, j), and OPT(i, j - 1)
Prove: The maximum profit for OPT(i, j) is either cutting the fabric up to include piece j or not. OPT(i - j, j) + the price of fabric length j includes our new length by sacrificing the last j inches while OPT(i, j - 1) is if our previous optimization is better. Our solution calculates the maximum of both cases and sets OPT(i, j) to it.

6)

- Create a 2D array where the first dimension is the combinations of coins that you can remove and the second is the combinations of coins that your opponent can remove
- OPT(i, j) is the highest sum of values we can get from our combination of coins that includes v_i but not v_j
- For i and j = 1, set OPT(i, i + 1) and OPT(j, j + 1) to the value of the only coin in that combination
- For i = 2, set OPT(i, i + 2) to the value of the largest coin while for j = 2, set OPT(j, j + 2) to the value of the smaller of the 2 coins
- For all valid coin combination lengths 3 <= j <= n,
    - For all valid 1 <= i <= n - j,
        - OPT(i, i + j) = max(
            - min(OPT(i + 1, i + j - 1), OPT(i + 2, i + j)) + v_i
            - min(OPT(i, i + j - 2), OPT(i + 1, i + j - 1)) + v_{i+j-1})
- Return OPT(1, n + 1)

Time complexity: if we scour the entire array we have $O(n^2)$ for the total number of elements we have assuming we have a coefficient of n for the rows and columns. For each element we perform an O(1) operation so we don't have to multiply our time complexity by anything else.

Proof: prove by induction
BASE CASE: relatively straightforward: if we have only one coin in each combination the maximum we can choose is just that one coin. If we have two we have us as the first movers take the higher value while the opponent takes the smaller of two values.
Assume: for all previous amounts of coins that what we have right now (let's call that c) our algorithm correctly returns the maximum value we can get from it.
Prove: we can only choose the coin at the start or end. Since our opponent is using the strategy listed in the description of the problem, they're maximizing their own profit–in turn which is minimizing our own profit. We can turn this around at our opponent by looking at the minimum

of the two options our opponent has when it’s the next turn. So we simply look at the optimization of the minimum that the opponent could get each step of the way.