Homework 5

Tejas Kamtam

305749402

Discussion 1B - Friday, 10 am

TA: Parshan

Problem 1

> Given an unsorted integer array, find all pairs with a given difference k in it without using any extra space (sorting is allowed).
> arr = (1, 5, 2, 2, 2, 5, 5, 4)
> k = 3
> Output: (2, 5) and (1, 4)

Algorithm

- get an array `A` of the unique values of the input array
- Sort the array `A` in increasing order
- Get `l` and `r` pointers to the first value
- check `diff = arr[r] - arr[l]`
  - if `diff == k`: we found a pair so output the values at the indices, and increment the left and right pointers s.t. `l += 1` and `r += 1`
  - else if `diff > k`: shift the left pointer to the right such that `l += 1`
  - else: shift the right pointer to the right such that `r += 1`

- repeat while the `r` pointer is less than the length of the array `A`

Time/Space Complexity

- sorting takes $O(n \log n)$
- with something like heapsort, we can sort in place
- the while loop will loop for all elements in the `arr` so at most $O(n)$
- pointers are constant space
- so auxiliary space is $O(1)$

Proof by Cases

- Sorting with heapsort can perform the sort in place given the array is passed correctly during precomputation
- We initialize both pointers to the starting value because the difference could come from a pair of the same value
- At each iteration, there are 3 cases:
  - `diff == k`: we have found a valid pair, so we want to shift to the next set of values on both pointers since we only find 1 distinct solution per unique pair
  - `diff > k`: this means that the value at our right pointer minus the value at our left pointer is greater than `k`. Because our right pointer may have been shifted by a previous iteration, there is a possibility that we can achieve a difference of `k` by only shifting the left pointer one down. So, we do so
  - `diff < k`: this means that our difference is not large enough, so we should move our right pointer to the right because we have already explored all

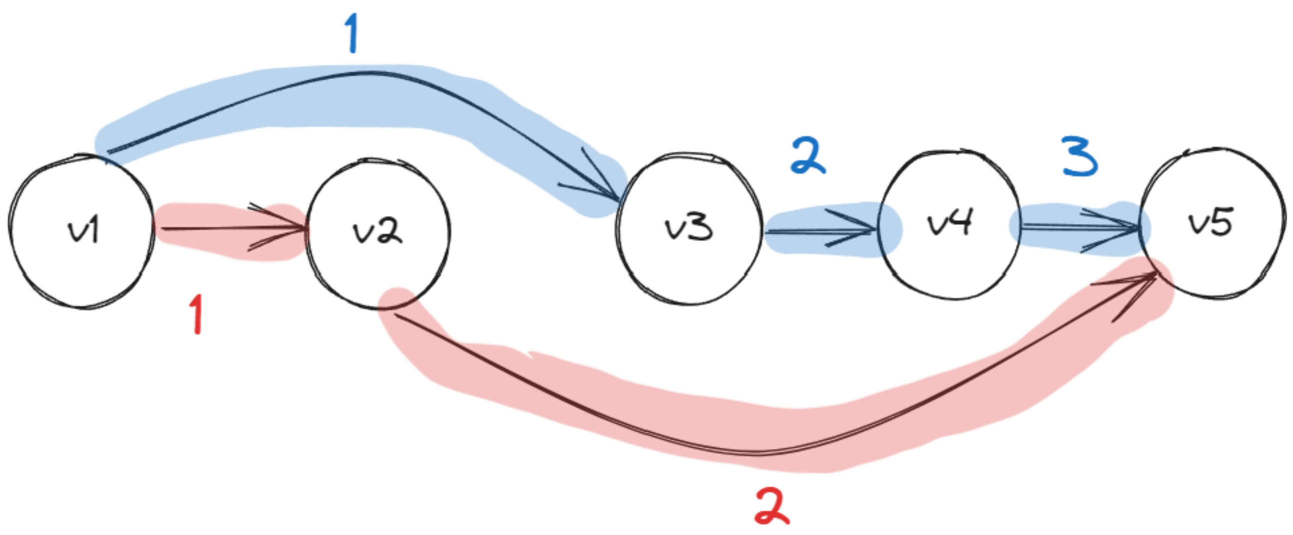
    options with moving the left pointer further left as the array is sorted

- These are the only 3 possible cases and because they cover the scope of possible differences for each pair, we have exhaustively covered each situation
- Additionally, if we consider our algorithm to not be optimal, then it means we missed a pair. However, because our algorithm runs for every value in the list (through both pointers, so long as the difference is still valid and the pointers are valid), it is not possible for us to have missed such a pair due to the exhaustive search - a contradiction.

Problem 2

> Exercise 3 on page 314

Part a



The directed, ordered graph above is a counterexample to the given algorithm. The optimal solution that maximizes the path length is shown in blue and has a length of 3. However, because the given algorithm takes the earliest possible next node reachable, it chooses to go to node v_2 which results in a suboptimal path length of 2.

Part b

Algorithm

- We initialize an array of zeroes, `OPT`, of size `V`, where `V` is the number of nodes in the graph, for which the value at position `i` represents the length of the path from node v_1 to v_i
- For each node `i`, we update the following:
  - `OPT[i] = max{OPT[j]+1}` over all `j` that can reach `i`
- For the base case, the path from node $v_1 \to v_1$ is length 0, so we enter `0` at position `1` (because the problem gives nodes as 1-indexed)
- Then, simply return the value at `OPT[V]` as the longest path length

Time Complexity

- Our outer loop performs computation for every vertex so the algo is at least $O(V)$
- For each node, we check all nodes that can reach it which is at worst $O(V)$
- So, overall time complexity is $O(V^2)$
- Space complexity is $O(V)$ for the memoized array

Proof by Contradiction

- Consider our algo is not optimal, these means we missed a possible longer path
- For the base case, we observe the path length from the first node to the first node is 0

- The recurrence relation our algorithm operates on consists of some $n \leq V$ number of cases:
  - all nodes reachable to node `i`
- This exhaustively considers all accessible, valid nodes correctly
- Finally, our algorithm memoizes the recurrence relation so the time complexity stands valid
- And it returns the last value in the cached array
- Thus, we exhaustively consider all cases, so it is not possible for our algorithm to have missed a valid path - a contradiction

Problem 3

> Exercise 5 on page 316

Algorithm

- Given the string 1-indexed string `y`
- Initialize an array, `OPT` of size `len(y) + 1` (because we want to be able to access the character `""` at the beginning)
- Set `OPT[0] = 0` (because we are given a string that is 1-indexed)
- Initialize a variable `max_quality = 0`
- Initialize an array `SEG` to represent the segmented string
- For each character in the string, starting at `i=1`, perform the following:
  - Set `OPT[i] = max{OPT[j] + quality(y[j+1], … ,y[i])}` where `max` operates over all `j ⩽ i`
  - If `OPT[i] > max_quality`, we update it to represent the highest overall quality we can represent

    - Additionally, save the index `j` that contributed to the max quality in the `SEG` array
    - Every time our `max_quality` is updated with a value that relied on a previous word (found when `OPT[j]` is non zero in the max function), save
- Now, `OPT[-1]` (last value) represents the maximal quality and the `SEG` array shows the locations of the string dividers
- Output this segmentation

Time Complexity

- We run through each element in the string of length `n` so it is lower bounded by $O(n)$
- For each iteration, we look at all substrings from index `j` to `i` which is $O(n)$
- So overall, the algo is $O(n^2)$

Proof by Contradiction with Exhaustive Case Analysis

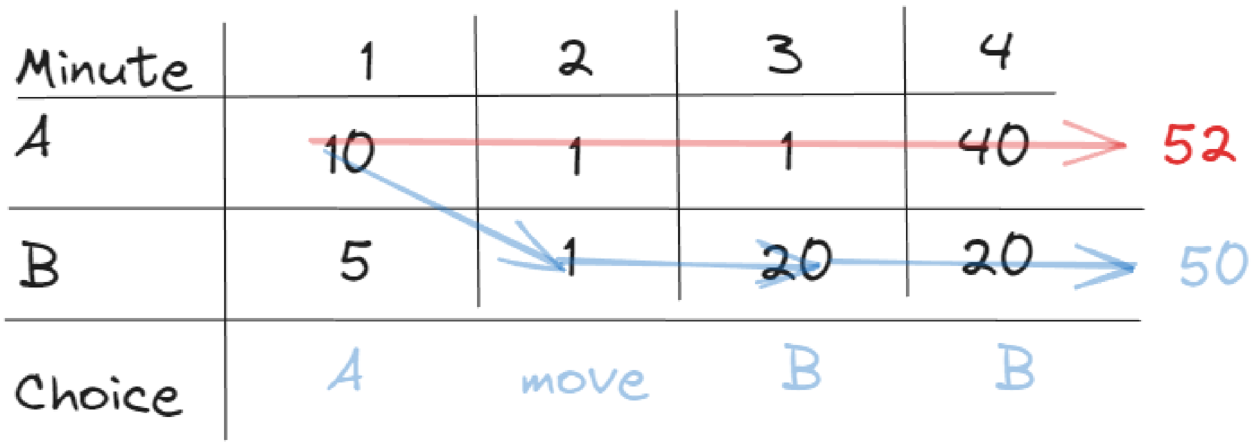
- Consider our algorithm to be sub-optimal, then we must have produced at least 1 incorrect segmentation
- Our algorithm ensures the base case for no elements in the array, the maximal quality is 0 and thus the segmentation/segments are the entire array
- For a sequence length of 1, similarly it is 0 or the quality of the character as given by our selection for `OPT` as the max of `0` and the base case + quality of the character
- For any length of string, our algo looks at every possible continuous segment of the string up to that point in the iteration thus ensuring we take the maximum over all valid substrings

- This is an exhaustive consideration of all possible cases that is correctly identified as the recurrence relation: 0 or the cumulative possible qualities
- Thus, it is not possible for our algo to have missed such a segmentation - a contradiction

Problem 4

> Exercise 10 on Page 321

Part a



Minute	1	2	3	4
A	10	1	1	40
B	5	1	20	20
Choice	A	move	B	B

The example above shows an optimal solution of staying on A and achieving a plan value of 52.
The proposed algorithm however, would first choose to begin on A for minute 1. Then, because `20 > 1 + 1`, it would `move` on minute 2 and select `B` for minute 3. Finally, the algo would be on minute 4 but because there are no longer any minutes, we assume the algo stays on `B` as to not waste steps resulting in a total plan value of 50 - which is not optimal.

Part b

Algorithm

- Initialize two arrays, `OPTA` and `OPTB` where the value in either represents the maximum value of a plan that ends on computer `A` or `B`, respectively, for minutes 1 to `i`
- On minute 1, initialize `OPTA[1] = a1` and `OPTB[1] = b1` (because the minutes are 1-indexed)
- For each following minute `i`, perform the following:
  - `OPTA[i] = ai + max{OPTA[i-1], OPTB[i-2]}`
  - `OPTB[i] = bi + max{OPTB[i-1], OPTA[i-2]}`
- Then return `max{OPTA[-1], OPTB[-1]}`

Time Complexity

- We run through each minute up till `n` minutes
- For each iteration we calculate a maximum for at most 2 elements - a constant time operation
- Thus, the overall algo is $O(n)$

Proof by Exhaustive Case Analysis

- Consider our algorithm is sub-optimal, thus outputting a plan value less than the true optimal - this means we must have made an incorrect choice to `move` on either `OPTA` or `OPTB` depending on which computer the optimal solution ends on
- We simultaneously consider both options of choosing to start on either computer `A` or `B`
- Then for each minute we choose the globally optimal choice by considering all cases:
  - Consider the following cases for computer `A` - a symmetric case consideration is true for computer `B`
  - At time `i`, we run on computer `A` and thus collect the steps of the computation available on minute `i` for computer `A` and add it to our previous choice

    - At the previous choice we could have either stayed on `A`, thus adding to `OPTA[i-1]`
    - Or we could have moved, thus adding to the value at `OPTB[i-2]` (because `i-1` was a move)
    - So, we take the maximum of both cases and add it to the value we accumulate on computer `A` at time `i`: value `ai` - and simultaneously the converse for `OPTB`
- Because these are all the choices: stay on the computer or move, we have exhaustively covered each possible case
- Because we take the maximum of the possible case at each step and we consider both using computer `A` or `B`, it is not possible for us to have mistakenly made a `move` operation as it would not have been maximal in our case consideration - a contradiction to our consideration above

Problem 5

> Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces.
> For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)
> length: 1 2 3 4 5 6 7 8
> price: 1 5 8 9 10 17 17 20

length:	1	2	3	4	5	6	7	8
price:	1	5	8	9	10	17	17	20

Algorithm

- We can model this problem as the Knapsack problem we covered in class (the version allowing taking duplicates)

- We can analogize the array of the value of the items to the price array here
- We can analogize the array of the weight of the items to the length array here
- And, we can analogize the maximum knapsack size to be the total length of the rod
- Then, run the knapsack algorithm discussed in class
- The algo develops a matrix S to memoize the values, so return the last value at `S[-1][-1]`

Time/Space complexity

- The time complexity is the same as discussed in class
- So, it is $O(nS)$ where S is the size of matrix used to memoize results
- In the example, it so happens that the rod length array and price array are equal in length so it is $O(n \cdot (n \cdot n)) = O(n^3)$, however it is not guaranteed that we will be given prices for each integer subdivision of the rod (We could get prices for every even inch length, etc.)

Proof

- The proof is the same as discussed in class, so long as we can prove the analogies discussed in the algorithm hold
- The problem statement says to find the optimal subdivision of the rod which maximizes the value which we observe, through trial and error, can be solved using DP for an efficient solution
- Because the goal is the maximal value conditioned on the cap of the length of the rod, the problem is analogous to the knapsack problem with a cap on the weight of the knapsack

- Each rod fragment has a price just as each item in the knapsack problem has a value
- Similarly, each fragment has a length just as each item has a weight where each sum to the condition on the problem
- The only point of doubt is that we can choose 2 halves of the rod akin to selecting the same item twice in the knapsack problem. However, this is fine because this "unbounded" knapsack problem is precisely the one discussed in class

Problem 6

> Consider a row of n coins of values v1 … vn, where n is even. We play a game against an opponent by alternating turns (you can both see all coins at all times). In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can win if we move first, assuming the opponent is using an optimal strategy.
> Example 1: (5, 3, 7, 10): The user collects maximum value as 15 (10 + 5) - Sometimes the greedy strategy works
> Example 2. (8, 15, 3, 7): The user collects maximum value as 22 (7 + 15) -- In general the greedy strategy does not work

Algorithm

- Initialize a 2D matrix `OPT` such that the value at `OPT[i][j]` represents the maximum value we can collect from coin `i` to coin `j` in the row of coins where `vi` and `vj` represent the coins' values respectively
- For each

- Then, for each coin `i` we loop through each coin `j` and perform the following:
    - $\text{min1} = \min\left\{\text{OPT}[i+2][j],\ \text{OPT}[i+1][j-1]\right\}$
    - $\text{min2} = \min\left\{\text{OPT}[i+1][j-1],\ \text{OPT}[i][j-2]\right\}$
    - $\text{OPT}[i][j] = \max\left\{v_i + \text{min1},\ \text{OPT}[i+1][j-1]\right\},\ v_j + \text{min2}\right\}$
    -