CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2024**

Homework #3

Name(s): Raashi Maheshwari, Tanvi Yamarthy

Homework Policy

- If you leave a question completely blank, you will receive 20% of the grade for that question. This however does not apply to the extra credit questions.
- You may also consult all the materials used in this course (lecture notes, textbook, slides, etc.) while writing your solution, but no other resources are allowed.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question). See "Practice Homework" for an example.
- The extra credit problems are generally more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.
- **Groups:** You are allowed to form groups of size *two* or *three* students for solving each homework (you can also opt to do it alone if you prefer). The policy regarding groups is as follows:
  - You can pick different partners for different assignments (e.g., from HW1 to HW2) but for any single assignment (e.g., HW1), you have to use the same partners for all questions.
  - The members of each group only need to write down and submit a single assignment between them, and all of them will receive the same grade.
  - For submissions, only one member of the group submits the full solutions on Canvas and lists the name of their partners in the group. The other members of the group also need to submit a PDF on Canvas that contains only a single line, stating the name of their partner who has submitted the full solution on Canvas (Example: Say A, B, and C are in one group; A submits the whole assignment and writes down the names A, B, and C. B and C only submit a one-page PDF with a single line that says "See the solution of A").
  - You are allowed to discuss the questions with any of your classmates even if they are not in your group. **But each group must write their solutions independently.**

Problem 1. You are given a stream of integers. Given a window size W, output the median of the last W numbers. The algorithm should run in $n \log W$ for a stream of n integers.

Solution. Pseudocode:

```
function findSlidingWindowMedian(stream, W):

maxHeap = MaxHeap()
minHeap = MinHeap()
medians = []

for i = 0 to length(stream) - 1:
    num = stream[i]

    if maxHeap is empty OR num <= maxHeap.peek():
        maxHeap.insert(num)
    else:
        minHeap.insert(num)

    if maxHeap.size() > minHeap.size() + 1:
        minHeap.insert(maxHeap.extractMax())
    elif minHeap.size() > maxHeap.size():
        maxHeap.insert(minHeap.extractMin())

    if i >= W:
        remove_num = stream[i - W]
        if remove_num <= maxHeap.peek():
            maxHeap.remove(remove_num)
        else:
            minHeap.remove(remove_num)

        if maxHeap.size() > minHeap.size() + 1:
            minHeap.insert(maxHeap.extractMax())
        elif minHeap.size() < minHeap.size():
            maxHeap.insert(minHeap.extractMin())

    if i >= W - 1:
        if maxHeap.size() > minHeap.size():
            medians.append(maxHeap.peek())
        else:
            median = (maxHeap.peek() + minHeap.peek()) / 2
            medians.append(median)

return medians
```

For each number, you first have to insert it onto the right heap. If it is less than or equal to the root of the max-heap (larger of smaller half), we have to add it to the max-heap. If the number is greater than the root of the max-heap, it goes into the min-heap (stores larger numbers). For each ins, we need to make sure the size difference between the maxHeap and minHeap is no more than 1. If the maxHeap has one extra element, you move its root to the minHeap and vice versa.

Let us think about the proof of correctness for this algorithm: The main invariant for this algorithm follows that each and every element in the max heap should be less than or equal to all the elements in the min heap. And at most, the difference between them should be limited to 1. This algorithm

will work accordingly as long as these invariants are found to be followed. The way that the algorithm works is that if the max heaps' size is greater than the min heaps' size by more than 1, then we will move our max heaps' root to the min heap. What does this to? Firstly, it makes sure that the difference in the size between heaps is at most 1 and then it also ensures that every element that is in the minimum heap will be greater or at most equal to every element that is in the max heap. What happens if we add a new element which has a root that is larger than the max heap? Then we add it to the minimum heap. The same process occurs where if the min heap ¿ max heap, then we will move its' root to the max heap. Doing these above steps, allows for us to maintain the invariant constraints of our algorithm. When we are trying to find the median, then we can logically think about the two heaps. If the max heap has 1 element more than the min heap, then the root of the max heap will be the "middle point" , and the same vice versa for the min heap. In the case that both heaps have the same size, then we get the roots of both the heaps, add them and divide by 2. In other words, we find the average of the roots. Another thing that we must consider for this algorithm is the functionality of the sliding window technique. If our sliding window will exceed more than our N elements, then we do the same manipulations where we move the root either to the min or max heap. This will once again ensure that the difference between the heaps is at most 1, and that they maintain heap order.

Time Complexity: $O(nlogW)$ The heap operations (such as insert, remove) take $O(logW)$ time to execute.

After each insertion or removal (n times), the heaps are balanced to make sure that the median can be easily calculated at each step.

Thus, this algorithm takes $O(nlogW)$ time to execute.

Problem 2. You are given an array of characters $S[1:n]$ such that each $S[i]$ is a small case letter from the English alphabet. You are also provided with a black-box algorithm *dict* that given two indices $i, j \in [n]$, $dict(i,j)$ returns whether the sub-array $S[i:j]$ in array S forms a valid word in English or not in $O(1)$ time. (Note that this algorithm is provided to you and you do not need to implement it in any way).

Design and analyze a dynamic programming algorithm to find whether the given array S can be partitioned into a sequence of valid words in English. The runtime of your algorithm should be $O(n^2)$.

Example: Input Array: $S = [maythef orcebewithyou]$.

Assuming the algorithm *dict* returns that $may, the, f orce, be, with$ and *you* are valid words (this means that for instance, for *may* we have $dict(1,3) =$ True), this array can be partitioned into a sequence of valid words.

Solution. To solve this problem, we can use a nested loop with dynamic programming approach. Since we want to check whether the string can be broken into a sequence of valid English words, we would first have to split the word into substrings. We'd start off by checking if a valid English word is created with the last letter of the word being at the index we're at. If it is, we would continue moving forward and checking the rest of the string to see if it can be broken into valid English words.

The pseudocode is as follows:

```
def partition(Array S, index n)

    #this is initializing the dynamic programming array
    #dp_array[i] will be true if the subarray can be partitioned into a sequence of valid words
    dp_array = [False] * (n+1)

    #this is base case for an empty string
    dp[0] = TRUE
```

```
#iterating over every "end" index i to see if the substring of S[1, i] is a valid English word
for i from 1 to n:
    #for each index "i", we check to see if S[j, i] is a valid word and if the prefix upto
    #"j -1" is valid
    for j from 1 to i:
        #two conditions:
        # 1. dict(j, i) must be true which shows that the substring S[j: i]
        #would form a valid word
        # 2. dp[j -1] must be true, which means that S[1, j-1] can be
        #partitioned into valid words
        if dp_array[j - 1] and dict(j, i):
            #if conditions are met, the array is marked as true
            dp_array[i] = TRUE
            break

#return the result, which is the answer to whether the input string array
#can be partitioned into valid English words
return dp_array[n]
```

Proof of Correctness

We can use induction to prove that if the algorithm labels `dp_array[i]` as `True`, then the substring up to index i, $S[1:i]$, can be split or partitioned into a sequence of valid English words.

Base Case: When $i = 0$, this would mean that our input is an empty string with no characters. An empty string forms a valid partition since there is nothing to check. Thus, `dp_array[0]` is `True`.

Inductive Step: Assume that the algorithm has correctly computed the values of `dp_array[1]`, `dp_array[2]`, ..., `dp_array[k - 1]` up to some value $k-1$. This shows that if `dp_array[j]` is `True` for any value $j \leq k-1$, then the substring $S[1:j]$ can be partitioned into valid words.

To compute `dp_array[k]`, the algorithm considers all possible starting indices j from 1 up to k, where the substring ends at k. In other words, we consider substrings of the form $S[j:k]$.

For each value of j, we check two conditions:

1. `dp_array[j - 1]` is `True`, which means that the substring $S[1:j-1]$ can be split into valid words.
2. `dict(j, k)` is `True`, indicating that the substring $S[j:k]$ is a valid English word.

When both of these conditions are met for any j, then we can conclude that `dp_array[k]` is `True`. These conditions are sufficient to show that $S[1:k]$ can be partitioned into valid words, meaning `dp_array[k]` is `True`.

These conditions are also necessary because for $S[1:k]$ to be partitioned into valid words, there must exist an index j such that:

- $S[1:j-1]$ provides a sequence of valid words (i.e., `dp_array[j - 1]` is `True`), and
- $S[j:k]$ is a valid word (i.e., `dict(j, k)` is `True`).

Our algorithm checks these conditions by iterating through all possible values of j for each k. Thus, if a valid split exists, the algorithm will find it, and `dp_array[k]` will be correctly marked as `True`, indicating that $S[1:k]$ can be partitioned into valid words.

Conclusion: By induction, we can conclude that if $S[1 : i]$ can be partitioned into valid words, then `dp_array[i]` will be `True` for any i. This logic extends to $i = n$: if $S[1 : n]$ can be partitioned into a sequence of valid words, then `dp_array[n]` will also be `True`.

Problem 3. Suppose you are given an array $A[1 : n]$ of integers that can be positive or negative. A *contiguous* subarray $A[i : j]$ of A is called a **positive** subarray if sum of its entries is strictly positive, i.e.,

$$\sum_{k=i}^{j} A[k] > 0.$$

Design and analyze a dynamic programming algorithm that in $O(n^2)$ time finds the smallest number of positive contiguous subarrays of A such that every positive entry of A belongs to exactly one of these subarrays. You can assume that A has at least one positive entry and thus the answer is *at least* one.

(25 points)

Example: The correct answer for the following array is 3 subarrays.

positive-cover.png

Solution. We can use a dynamic programming algorithm to handle this question. We essentially want to have an array that holds the minimum positive sub arrays that can keep track of the positive values starting from index 0 to array[i]. We start by having a base case, which is when our sub array has 0 elements. This can be seen in the line DP[0]=0. We ensure that we are setting the rest our DP array to contain infinity values as placeholders before we fill the rest of it gradually. In this algorithm, you can see that we have two nested for loops, hence our quadratic run time. The outer loop traverse from i=1 all the way till i=the

length of the array. For every i'th index, we want to calculate the minimum sub array number needed to hit on all the positive elements in the array from index 1 to index 1. Thus, for every i, we want to see: which sub arrays end at this index? Then, we see a nested inner for loop. This inner for loop allows us to examine the sub arrays which do end at index i. Then we traverse backwards through the sub arrays and add to a sum variable. Let us take an example. We can assume that our input array is [1,-2,3,4,-1,2]. Once we do the basic initialization, this gets our DP array looking like: [0, inf, inf, inf, inf, inf]. In the first iteration (our first for loop), we are getting the sub arrays that end at index 1, and adding to our sum variable. Then we update our DP[i], with the minimum value between the current DP value and the old ones. We are constantly updating our minimum sub array sum value, and then at the end, we will return dp[n], because that will have the minimum sub array value.

```
public int minPlusArrays(array):
    int [] DP=new int [array.length+1]
    dp=[infinity// Math.MAX] //intialize
    dp with the largest number to compare to for minimum

    DP[0]= 0; //starting point, we have covered
    0 subarrays so far.
    There won't be any positive sum sub arrays/prefix sums to this point.

    for (int i=1; i< array.length; i++){ //traverse through the indices of the array
         currentSum=0
         for (int j=i; j>=1; j--): //find the subarrays that end at i
            currentSum+array[j-1]
            if currentSum>0:
                DP[i]=Math.MIN(DP[i],(DP[j-1]+1),
            else:
                break

    }
    return DP[array.length]

    }
```

Let us now look at the proof of correctness for this algorithm: Like I mentioned before, the base case is when DP[0]=0, indicating that at this index there is no existing prefix sum. Our inductive step lies in the assumption that DP[1] to DP[i-1] are already defined, and thus, these sums can be used to calculate the minimum sum for DP[i]. Since we know that the previous indices in DP[i], represent the previous minimum positive sub array sums, we can find the new DP[i], by finding the minimum between the computed DP[i] and DP[j-1]+1. Thus, our algorithm is bound to work as it accounts for the base cases and the inductive steps as well for the other other iterations of the array.

Problem 4. We define a number to be **special** if it can be written as:

$$n = a \cdot 197 + b \cdot 232$$

where a and b are non-negative integers. For example:

- 696 is special because it can be written as $696 = 0 \cdot 197 + 3 \cdot 232$.
- 2412 is special because it can be written as $2412 = 4 \cdot 197 + 7 \cdot 232$.

- 267 is not special (since it cannot be written as $a \cdot 197 + b \cdot 232$ for non-negative a and b).

The goal of this problem is to write a dynamic programming algorithm **Special(n)** that, given a positive integer n, either finds non-negative integers a and b such that $n = a \cdot 197 + b \cdot 232$, or returns "no solution" if no such a and b exist.

Remember to separately write your specification of recursive formula in plain English , your recursive solution for the formula and its proof of correctness , the algorithm using either memoization or bottom-up dynamic programming , and runtime analysis.

Solution.

Recursive Formula Specification in English

```
A number n is special if it can be written as a*197 + b*232 where a and b
are greater than or equal to 0

Base Case: If n = 0, it is special since a = 0 and b = 0

Recursive Case:
    If n < 0, it is not special
    If n > 0, n is special if:
        n - 197  is special
        n - 232 is special
    **because you can add 197 or 232 more**
```

Recursive Solution for the Formula

```
def checkIsSpecial(n) {
    true is n = 0
    false if n < 0
    checkIsSpecial(n - 197) OR checkIsSpecial(n - 232) if n > 0
}

for values of n less than or equal to 0:
def Special(n) = {
    (0,0) if n = 0
    "no solution" if n  < 0
    if checkIsSpecial(n - 197):
        let (a,b) = Special(n - 197)
        return (a + 1, b)
    else if checkIsSpecial(n - 232):
        let (a,b) = Special(n - 232)
        return (a, b + 1)
    else: "no solution"
}
```

Proof of Correctness for Recursive Solution

```
Base Cases:
1. n = 0: Algorithm returns (0,0) which is correct since 0*197 + 0*232 = 0
2. n < 0: Algorithm returns "no solution", which is accurate as you cannot have
a negative number as an answer if you have non-negative coefficients
```

```
Inductive Case:
We assume that the algorithm is correct for all values less than some k > 0
Then for n = k:
    1. If k can be represented with a*197 + b*232:
        k - 232 can be represented as a * 197 + (b - 1)*232
        k - 197 can be represented as (a - 1)*197 + b * 232
        The algorithm will be able to find these solutions by the inductive hypothesis

    2. If k cannot be represented in the above manner, then:
         k - 197 or k - 232 both cannot be represented
         The algorithm will proceed to return "no solution" for both those cases.

By a proof of induction, we have proved that the algorithm is correct for all n
greater than or equal to 0
```

Dynamic Programming (Bottom-up) Algorithm: Pseudcode

```
def isSpecial(n) {
    //initialize a dp array where dp[i] stores {possible:bool, a:int, b:int}
    dp_array = new array[n + 1] where {possible: false}
    dp_array[0] = {possible: true, a:0, b:0}

    for i from 0 to n:
        if dp_array[i].possible:
            if i + 197 <= n:
                dp_array[i + 197] = {
                    possible: true,
                    a: dp_array[i].a + 1,
                    b: dp_array[i].b
                }

            if i + 232 <= n:
                dp_array[i + 232] = {
                possible: true,
                a: dp_array[i].a,
                b: dp_array[i].b + 1
                }

    if dp_array[n].possible:
        return (dp_array[n].a, dp_array[n].b)
    else:
        return "no solution"
}
```

Runtime Analysis

Recursive Solution (without memoization)

```
    Each call branches into two recursive calls.
    The maximum depth would be O(n/min(197,232)).
    This would lead to a runtime of O(2^n).
```

Dynamic Programming Solution

The algorithm processes each number from 0 to n only once and for each number, we perform a constant number of operations. This would lead to a time complexity of O(n).

Correctness of Dynamic Programming Solution

In order to show that the DP solution works as intended, we need to prove two things:
1. Completeness of the Solution: If dp_array[n] = True, then the algorithm can correctly identify the non-negative integers a and b such that n = a * 197 + b * 232

We know that if dp_array[n] is True, then n can be represented by a * 197 + b * 232.
Thus, if dp_array[i] is True, then it must be reachable by either doing dp_array[i - 197] or dp_array[i - 232] or both.

If we start from n, the algorithm starts by checking if dp_array[n - 197] is True.
If it is true, we do n - 232, which means we decrement the count of 232 in the sum by 1.
This process continues until n becomes 0 and the counts of how many times we subtracted 197 and 232 gives us the values of a and b.

Since we only subtract values guaranteed by the algorithm from n, we know that n will be represented in terms of 197 and 232, thus, the algorithm will always return a correct solution when dp_array[n] is True.

2. Correctness of the DP formulation: if dp_array[i] is True, then i can be represented as i = (a * 197) + (b * 232)

Base Case: dp_array[0] is True because 0 can be trivially represented with a = 0 and b = 0

Inductive Step: We assume that dp_array[k] = True correctly represents that k can be written as l = (a * 197) + (b * 232) for all k < i

If dp_array[i - 197] is True, then i - 197 can be written as (a - 1) * 197 + b * 232.
Thus, we we add one more value of 197 then i can be written as i = a * 197 + b * 232.

This same logic could be used for dp_array[i - 232] being True, then i - 232 can be written as a * 197 + (b - 1) * 232. Thus, if we add one more value of 232, i can be written as i = (a * 197) + (b * 232).

By a proof of induction, we have shown that if dp_array[i] is True, then the algorithm correctly indicates that i can be represented as a sum of non-negative multiples of 197 and 232.

Edge Cases

- Small Values of n

  dp_array[i] is only set to True if i is greater than or equal to 197 or greater than or equal to 232. Thus, dp_array[i - 197] or dp_array[i - 232] is True. Thus, any value smaller than 197 or 232 will not be set to True unless there is a way they can be represented as multiples of these numbers

- No Solution Case

```
If dp_array[n] is false, then the algorithm correctly returns "no solution" because there
is no way you can express n as a sum of non-negative multiples of 197 and 232.
```

Problem 5. For the most part, we haven't been optimizing for space in our dynamic programs. But for many of the problems we've discussed it is possible to get away with much less space. The goal of this problem is for you do out one such example.

Recall the subsetsum problem from class. INPUT: set $S = \{s_1, \ldots, s_n\}$ of non-negative integers and a target number B. OUTPUT: set $S' \subseteq S$ with $\sum_{x \in S'} = B$ or FALSE if no such S' exists.

The algorithm we showed in class used $O(nB)$ time and $O(nB)$ space. Write pseudocode for an algorithm that solves subset sum in $O(nB)$ time but only needs $O(B)$ space in addition to the input itself. Your algorithm must return the actual set S'. For this problem you ONLY need to write pseudocode.

Solution. Pseudocode:

```
function findSubsetSum(S, B):

        dp_array = [False] * (B + 1)
        dp_array[0] = True

        trail = [None] * (B + 1)

        for i in range(len(S)):
            current = S[i]

            for j in range(B, current - 1, -1):
                if dp_array[j - current]: -
                    dp_array[j] = True
                    trail[j] = i

        if not dp_array[B]:
            return False

        result = []
        sum_needed = B
        while sum_needed > 0:
            index = trail[sum_needed]
            result.append(S[index])
            sum_needed -= S[index]

        return result
```

This is a dynamic programming approach used to solve the subset sum problem. We maintain two arrays: dp_array which tracks the achievable sums and trail[] which stores the element that was last used to achieve each sum.

The algorithm works to process each element one at a time from the input set S. For each element, it works backward, iterating from the target sum B to the current element's value. It checks if adding the current

element to the previous achievable sum creates new achievable sums. We avoid counting the same element multiple times in a sum by iterating backwards. When an achievable sum is found, the algorithm stores the element that was used (in trail[ ]) and the fact that the sum is achievable (in dp_array).

If we find the target sum B, it is marked as achievable and the algorithm works to reconstruct the actual subset. It works backwards through the trail array, starting from the target sum B, it checks which element was last used to achieve that sum, adds it to the result, subtracts it from the remaining sum needed. This process repeats until it reaches zero. If the target sum is never marked as achievable, the algorithm returns the value of False.

Time complexity: $O(n * B)$ This is because our algorithm has one main loop that runs for n elements. It also has an inner loop up to B for each element. Thus, the total run time would be $O(n * B)$.

Problem 6. ImportExport(A,B)

Input: Two arrays $A[0 \ldots n-1]$ and $B[0 \ldots n-1]$, both of length n, representing the prices of tulips in country A and country B at different times, respectively. All entries in both arrays are positive integers.

Output: Return the maximum possible profit, defined as:

$$\max_{j>i}\{B[j] - A[i]\}$$

or return 0 if all such $B[j] - A[i]$ are negative. You only need to return the value of the maximum profit, not the actual indices i and j.

Example:

Given the input:

$$A = [40, 18, 20, 25, 12, 13, 19, 20, 5, 7, 3]$$
$$B = [15, 50, 5, 30, 34, 19, 28, 33, 20, 20, 20]$$

The maximum possible profit is achieved by buying tulips at time $i = 4$ (when $A[4] = 12$) and selling them at time $j = 7$ (when $B[7] = 33$), which gives a profit of:

$$B[7] - A[4] = 33 - 12 = 21$$

Therefore, the output is 21.

- Give pseudocode for a recursive $O(n)$-time algorithm for ImportExport(A,B).
- Give pseudocode for a $O(n)$-time dynamic programming algorithm for ImportExport(A,B).

Solution. Recursive Approach Pseudocode:

```
function ImportExport(A, B):
    return getMaxProfit(0, A[0])

function getMaxProfit(i, minA):
    if i == length(B):
        return 0

    profit = B[i] - minA
```

```
    currentMaxProfit = max(profit, findMaxProfit(i + 1, min(minA, A[i])))

    return currentMaxProfit
```

The recursive approach has a base case. When i reaches the end of the array B, there are no more prices left to compare, so the algorithm returns 0.

For the recursive step, we take each index i in B. For each index, the algorithm computes the potential profit by buying at the minimum priced observed to date in A and by selling at B[i]. It also makes sure that it updates the minimum track price in A up to the current index. This makes sure that we always have the lowest price to buy from A at any given point. Lastly, the algorithm also calls getMaxProfit recursively with i + 1 which allows it to continue checking the future prices in array B.

Proof by Correctness: The base case is when i is equal to the length of B, which means we have checked all the possible prices in B. At this point, the algorithm will return 0 which indicates no profit, since there are no further prices to consider in B.

The profit is calculated based on buying at the minimum price seen so far and selling it at B[i]. The algorithm updates minA by comparing it with A[i], which means it will always contain the minimum price up to index i.

For the recursive case, we have each i in the range (0, length(B)). The recursive function always maintains two things. The first thing is minA, which is the minimum value of A[0] through A[i]. This ensures that we always consider the lowest possible buying price up to each index i. The second thing that the algorithm tracks using recursive returns is the maxProfit. This stores the highest profit possible by checking all the valid B[j] - A[i] pairs as the function progresses. The approach finds the maximum profit for each subarray of B and recursively builds up the maximum profit for the entire array. The algorithm keeps every selling price in B against the lowest buying price in A up to the index i. Thus, maxProfit will always contain the maxProfit from selling in A and buying in B, which proves the correctness of the algorithm.

Dynamic Programming Approach Pseudocode:

```
function ImportExport(A, B):
    minA = A[0]
    maxProfit = 0

for j = 1 to length(B) - 1:
    profit = B[j] - minA
    maxProfit = max(maxProfit, profit)
    minA = min(minA, A[j])

return maxProfit
```

In order for the algorithm to efficiently find the maximum profit that can be achieved by buying tulips in country A and selling them in country B, it keeps track of two price arrays: A and B. It iterates through the two arrays and keeps track of two main variables: minA which represents the best price to buy as it would be the lowest price seen so far in array A and maxProfit which represents the highest profit seen so far as the algorithm continues to run.

minAA is set to the first element of A, as it is the price observed initially and maxProfit is set to 0, as there are no transactions to keep in mind. As the algorithm proceeds, for each index j in array B, it calculates the potential profit if the tulips are brought at minA and sold at B[j]. This current profit is compared with maxProfit, and maxProfit is updated if the current profit is greater. At the same time, minA is compared with A[j], and is updated to the lowest price up to index j. This makes sure that it represents the best buying

price before each potential sale made. By the end of the loop in the algorithm, maxProfit stores the value of the maximum profit possible from the two arrays (stores 0 if there is no profit possible). This algorithm achieves this in $O(N)$ time.

Proof by Correctness:

Base Case: loop has not yet started to run so minA = A[0] and maxProfit = 0, which is accurate as no comparisons have been made yet

Inductive Step: We assume that maxProfit is correct after processing B[j - 1]. When we start running the algorithm on B[j], the algorithm correctly updates the profit as B[j] - minA. If profit is greater, it also updates maxProfit, thus making sure that the algorithm stores the maximum possible profit up to index j. minA is also updates to make sure it stores the minimum price of A[0] through A[j].

If all the profits are negative, then maxProfit would store the value of 0, thus the algorithm would return 0 when there are no profitable sales possible.

Thus, by induction, we have proved that maxProfit will always store the highest possible profit for the entire array after processing all elements of array B.

Runtime: Both algorithms take $O(n)$ time to run. The recursive algorithm calls getMaxProfit(i, minA) and the function is called once for every index i in the range (0, n-1). This leads to n number of calls and each recursive call is performing a constant amount of work. Since each element is processed once with constant work at each step, the recursive approach has an overall time complexity of $O(n)$

The dynamic programming algorithm also runs through the arrays only once as the loop runs from j= 1 to n-1, making that have n - 1 iterations. In each iteration, there is a constant amount of work being done. Thus, the dynamic programming algorithm also has a time complexity of $O(n)$ as it processes each element in the arrays A and B only once and is performing a constant number of work per element.

Problem 7. Suppose we have n persons with their heights given in an array $P[1:n]$ and n skis with heights given in an array $S[1:n]$. Design and analyze a greedy algorithm that in $O(n \log n)$ time assign a ski to each person so that the average difference between the height of a person and their assigned ski is minimized. The algorithm should output a permutation σ of $\{1, \ldots, n\}$ with the meaning that person i is assigned the ski $\sigma(i)$ such that

$$\frac{1}{n} \cdot \sum_{i=1}^{n} |P[i] - S[\sigma(i)]|$$

is minimized.

Example: If $P = [3, 2, 5, 1]$ and $B = [5, 7, 2, 9]$, we can output $\sigma = [2, 1, 4, 3]$ with value

$$\frac{1}{4} \cdot \sum_{i=1}^{4} |P[i] - B[\sigma(i)]| = \frac{1}{4} \cdot (|P[1] - B[2]| + |P[2] - B[1]| + |P[3] - B[4]| + |P[4] - B[3]|) = \frac{1}{4} \cdot (4 + 3 + 4 + 1) = 3,$$

which you can also check is the minimum value.

Solution. This purpose of this question is to find a permutation so that every person assigned a ski will have the minimal absolute distance between their own height and the ski's height. Because the question asks us to use the greedy algorithm, when we are trying to implement the greedy approach, the first thing that comes to our mind is to think that the biggest heights will correspond to the biggest ski heights, and the smallest person heights will correspond to the smallest person heights. This logically makes sense to us because if the heights of the person and the ski are closer together, they will have a lower absolute difference between them. Taking this information, let us try to devise an algorithm that will tackle this. The first thing

that we can do is to sort both arrays in ascending order, arranging the heights of the skis and the person from smallest to greatest. Then, because they will be sorted, intuitively ski [i] can be assigned to match up with person [i]. Because the question is asking for the "distance between the height of the ski and the person is minimized", we can calculate the difference between those two height differences. In this question, the best permutation refers to the "pairing" of the heights and the skis. So now that we talked about the algorithm, let us write the pseudo code:

```
public int[] skiHeightPairs(int []S, int[]P){

    int people=P.length
    int skis=S.length

    //new arrays to keep track of indices
    int []pIndex= new int [P.length]
    int []sIndex= new int [S.length]

//populate the arrays w/ their indices
    for (int i=0; i<n; i++){
        pIndex[i]=i;
        sIndex[i]=i;
    }

    //use merge sort to sort pIndex, and sIndex arrays
    based on the persons's height from array P, and the ski heights from array S.

    mergeSort on (pIndex, P)
    mergeSort on (sIndex, S)

    this is a black box algorithm that we learned in class.

    int [] permutations= new int [P.length]
    for (int i=0; i< P.length; i++){
        permutations[pIndex[i]]=sIndex[i]
        //makes it so that the person at index i gets assigned the ski at index i.
    }

    return permutations;

}
```

Now, let us analyze the proof of correctness for this greedy algorithm. In the beginning, we talked about how the greedy algorithm ensures that we assign the lowest height ski to the lowest height person, and vice versa. Our base case is when there is only one person and on ski. In other words, when the length of P is 1. Then, the greedy algorithm that we wrote would sort both array P, and array S, (but since there's only 1 element there's no real sorting that happens). Then, assigns the indices to each other since it is the minimal and only difference. Now, we can talk about the inductive step which aims to prove that this algorithm works for k+1 as well. The greedy algorithm will sort all the elements , so that P[k+1] will be assigned to S[k+1]. Intuitively, if there are 2 people one with height of 2cm, and on with height of 3 cm. And two skis, one with height 5 cm and the other 6 cm. If the 2cm person gets assigns to the 5 cm height and the 3 cm person gets assigned to the 6 cm height, the absolute difference between the heights and the ski heights will be minimized, and that is what our greedy algorithm does.

Problem 8. Given a set of jobs, each with a deadline and a processing time, schedule the jobs to minimize the maximum lateness, where lateness is defined as the time difference between a job's completion and its deadline (if it's completed after the deadline). Write an algorithm and prove its correctness.

Solution. To solve this problem and minimize the maximum lateness of a set of jobs with given deadlines and processing times, we can first sort the jobs by their deadlines in ascending order (the earliest deadline would be placed first). The jobs would then also be scheduled in that order.

We need the following information for the pseudocode for the algorithm:
Input: array of jobs where each job j has a processing time t and deadline dl
Pseudocode:

```
def minimizeMaximumLateness(Array jobs)
    #Step 1: Sort the jobs by earliest deadline first
    sort jobs using Merge Sort in ascending order

    current_time = 0
    max_lateness = 0
    schedule = []

    for j = 0 to n-1:
        job_start_time = current_time
        job_finish_time = current_time + jobs[j].processing_time
        lateness: max(0, job_finish_time - jobs[j].deadline)

        schedule.append({
        'job_id': j + 1,
        'start': job_start_time,
        'finish': job_finish_time,
        'lateness': lateness
        })

        current_time = job_finish_time
        max_lateness = max(max_lateness, lateness)

    return schedule, max_lateness
```

Proof of Correctness by Induction

Let each job j have:

- p_j: processing time,
- d_j: deadline.

The **lateness** L_j for job j is defined as:

$$L_j = C_j - d_j,$$

where C_j is the time needed to complete the job j. We want to minimize **maximum lateness** which can be represented by:

$$L_{\max} = \max_j L_j.$$

The algorithm works by sorting and scheduling the jobs in ascending order.

Base Case: $n = 1$

If there is only one job, scheduling it at time $t = 0$ is optimal:

- The completion time C_1 is p_1,
- The lateness $L_1 = C_1 - d_1 = p_1 - d_1$,

and since there is only one job, this schedule minimizes the maximum lateness.

Thus, the algorithm is correct for $n = 1$.

Inductive Hypothesis

Assume that for any set of n jobs, scheduling them in **ascending order** minimizes the maximum lateness.

Inductive Step: Prove for $n + 1$ Jobs

Now, let's consider a set of $n + 1$ jobs. We need to show that scheduling these $n + 1$ jobs in **ascending order** also minimizes the maximum lateness.

1. Sort the $n + 1$ jobs by their deadlines in ascending order $d_1 \leq d_2 \leq \cdots \leq d_{n+1}$.
2. According to the inductive hypothesis, scheduling the first n jobs (i.e., $\{1, 2, \ldots, n\}$) in this order minimizes their maximum lateness.
3. The job $n + 1$ has the latest deadline in the sequence $d_{n+1} \geq d_j$ for all $j \leq n$. Thus, we add job $n + 1$ after the first n jobs.
4. **Calculate the New Lateness**:
   - Let C_n be the completion time of the first n jobs (using the order defined by their deadlines).
   - The start time for job $n + 1$ will be C_n, and its completion time will be $C_{n+1} = C_n + p_{n+1}$.
   - The lateness of job $n + 1$ will be:

$$L_{n+1} = C_{n+1} - d_{n+1} = (C_n + p_{n+1}) - d_{n+1}.$$

5. Scheduling job $n + 1$ last minimizes its contribution to the maximum lateness since it has the latest deadline. The maximum lateness $L_{\max}$ for the $n + 1$ jobs will either be the maximum lateness of the first n jobs or the lateness of job $n + 1$. Thus, $L_{\max}$ for $n + 1$ jobs is minimized by this schedule.

Conclusion

By the principle of mathematical induction, scheduling jobs in **ascending order** minimizes the maximum lateness for any set of n jobs. This completes the proof of correctness.

Problem 9. You are given two integer arrays gas$[0 \ldots n - 1]$ and cost$[0 \ldots n - 1]$, where:

- gas$[i]$ represents the amount of gas available at the i-th gas station.

- cost[i] represents the amount of gas required to travel from the i-th gas station to the $(i+1)$-th gas station.

Your car starts with an empty gas tank, and you want to travel around the circular route once in the clockwise direction, beginning at one of the gas stations. The car has an unlimited gas tank, so it can collect all the gas available at each station as it passes.

The goal is to determine whether there exists a starting gas station from which you can travel around the circuit once. If such a starting gas station exists, return its index. Otherwise, return -1. If there exists a solution, it is guaranteed to be unique. Design an algorithm that runs in $O(n)$ time.

Solution. Our goal in this problem is to see whether there "exists a starting gas station" index so that you ar allowed to complete a full loop around all of the stations. When looking at this question, we can think of the edge cases that exist in order to rule out possible stations/answer choices. The first edge case that comes to my mind is looking at the cost array vs the gas array. In order to be able to make your way around the entire circuit, we need sufficient fuel so that the car can travel. If we ever have a situation where the sum of the gas that you are getting at the individual stations is less than the sum of the gas required to travel from one station to another, then it will be highly impossible to make it from one station to another, let alone complete a full circuit. In cases like this, we can return -1 to signify this. For this algorithm, we can use the greedy algorithm to approach the task at hand. We want to keep track of our total gas across all the stations as well as our gas balance to account for how much gas we have from a starting station if we were to start our journey there. As we traverse through each station, we ensure to update our two variables, and if our gas balance is ever less than zero, then we know that we did not have enough gas with that starting station index, and thus, we must increment it to point to the next index as a starting station. We can return out of the program with our updated index when our cumulative gas is a positive number. Let us look at some pseudo code that would help us further break down this problem:

```
function fullCircuit(gas, cost):
     //our function takes in the gas array
     and the cost array
     (amount of gas it takes to travel
     from one station to the next)

totalGas=0  //setting both our pointers to be 0 as traversing through
the stations has not yet started

     gasBalance=0
     startingStation=0

     for (int i=0; i<gas.length; i++) //traversing through the stations

        gasBalance+= gas [i]- cost [i];  //update gas tank values
        totalGas += gas[i] -cost [i];

        if gasBalance<0:  //if we don't have enough gas to travel to the next station
                startingStation++ //update our starting station
                gasBalance=0
                //set our balance back to 0


     if totalGas >=0 :
                return starting station
                //we had enough gas to make it around the stations
     else:
```

```
return -1;
//we didn't have enough gas so return -1
could complete circuit
```

Proof of Correctness: We can see that this is an O(n) runtime because we are just doing a single for loop to traverse through the subsequent arrays, and this for loop is dependent on the size of the arrays. While explaining this proof before the pseudo code, the proof of correctness was seen because we talked about the 2 conditions where we must ensure to check to see if our solution is feasible. We must check to make sure that if our total gas is less than the amount of gas (cost) needed to complete for the circuit, then we return -1, ensuring it is not. This is where the base of our algorithm is built. On top of this, we use the Greedy Algorithm to make sure that if our gas balance hits a negative number or 0, we reset it with a new index, letting us know that if we start at that station, we won't have enough gas to make it around the stations (circuit).In this situation our base case would be the failed condition where we are returning -1, implying that we cannot complete the circuit. We are then using an iterative process to check for the gas balance between every additional statement and if it is negative, then we know that this is not the station that is a viable option, and so we increment our starting station to the next one. Our inductive hypothesis would be that we are assuming that for a station b, where b¡ i, then if our gas Balance is greater than or equal to 0, then it is feasible for us to start our journey at the starting station and reaching until station b+1. If this is not true, then we will reset our stating index itself to b+1, and reset our gas Balance to 0, as there was not enough gas in the tank to go from the original starting station to station b+1. Our inductive step in this process is to continuously update our gas balance variable and check if is a negative or positive number, taking the appropriate course of action afterwards.

Challenge Yourself. You are given a string consisting of just the characters $'a'$, $'b'$, or $'c'$. The task is to sort the string in lexicographical order ($a < b < c$) by finding the minimum number of swaps needed to achieve this. Each swap can exchange two characters in the string.

Solution. To find an efficient solution to this problem, you can start off by counting the occurrences of the letters: `'a'`, `'b'`, `'c'`. Then, you would proceed to create three substrings: a substring for all the `a`'s, a substring for all the `b`'s, and a substring for all the `c`'s. Once those substrings are created, you would proceed to count the number of misplaced characters in each substring. This would look like counting the number of `b`'s and `c`'s in the substring for `a`, the number of `a`'s and `c`'s in the substring for `b`, and the number of `a`'s and `b`'s in the substring for `c`.

This would get us to the process of actually sorting the string now. There will be two types of swaps that we have to deal with. The first one is a **direct swap**, which can be resolved in one swap. This would mean that you swap the elements between the pairs right next to each other (i.e. `'a'` in `b`'s substring with a `b` in `a`'s substring). The other type of swap is **cyclic swap**, where all three characters need to swap within themselves. These **cyclic swaps** need two swaps each to place the character where it belongs.

The pseudocode for the algorithm is as follows:

```
function minimumSwapsToSortLexicographically(string S):
    count_a = count of 'a' in S
    count_b = count of 'b' in S
    count_c = count of 'c' in S

    substring_a = S[0:count_a]
```

```
substring_b = S[count_a:count_a + count_b]
substring_c = S[count_a + count_b:]

count_ab = count of 'b' in substring_a
count_ac = count of 'c' in substring_a
count_ba = count of 'a' in substring_b
count_bc = count of 'c' in substring_b
count_ca = count of 'a' in substring_c
count_cb = count of 'b' in substring_c

direct_swaps = min(count_ab, count_ba) + min(count_ac, count_ca) + min(count_bc, count_cb)

remaining_mismatches = abs(count_ab - count_ba) + abs(count_ac - count_ca)
+ abs(count_bc - count_cb)

cyclic_swaps = (remaining_mismatches // 3) * 2

total_swaps = direct_swaps + cyclic_swaps

perform_swaps(S)

return total_swaps, sorted S
```

Helper Method for the Sorting

```
def perform_swaps(String S) {

#convert string to a list so that it's easier to manipulate
S = list(S)

for i in range(len(S)):
    if S[i] == 'a':
        for j in range(i+1, len(S)):
            if S[j] == 'b':
                S[i], S[j] = S[j], S[i]  # Swap 'a' with 'b'
                break

for i in range(len(S)):
    if S[i] == 'b':
        for j in range(i+1, len(S)):
            if S[j] == 'c':
                S[i], S[j] = S[j], S[i]  # Swap 'b' with 'c'
                break

for i in range(len(S)):
    if S[i] == 'c':
        for j in range(i+1, len(S)):
            if S[j] == 'a':
                S[i], S[j] = S[j], S[i]  # Swap 'c' with 'a'
                break

#for cyclic swaps
for i in range(len(S)):
    if S[i] == 'a':
```

```
            for j in range(i+1, len(S)):
                if S[j] == 'b':
                    for k in range(j+1, len(S)):
                        if S[k] == 'c':
                            S[i], S[j] = S[j], S[i]
                            S[j], S[k] = S[k], S[j]
                            S[i], S[k] = S[k], S[i]
                            break

    return ''.join(S)
    }
```

Proof Of Correctness: Our base case would be for a string with length $n = 1$. In this case, the string only has one character, which means the minimum number of swaps needed is zero, which is correct.

Our inductive hypothesis assumes that for any string of length k, where $k \leq n$ and only has the letters 'a', 'b', and 'c', the algorithm works as intended and correctly sorts the string in lexographically order using the minimum number of swaps (both direct and cyclic swaps).

Using our inductive hypothesis, we need to show that for a string S of length $n = k + 1$, the algorithm also works as intended to sort the string lexographically with the minimum number of swaps. The algorithm will first count the occurrences of each letter and define three respective substrings for each letter: 'a', 'b', and 'c'. If the sorting is incorrect for S, we know that there must be AT LEAST one pair of misplaced characters in different substrings. If this is not present, then the substring would already be sorted.

Our algorithm correctly handles direct swaps for strings up to length k, based on our inductive hypothesis. For strings with length $k + 1$, this strategy still works and minimizes swaps within the substrings, as it is optimal to correct two mismatch characters with a direct swap whenever feasible.

If no more direct swaps can be done and the string is still not sorted, the other misplaced characters need cyclic swaps. By the inductive hypothesis, we are guaranteed that: $(remainingMismatches/3) * 2$ swaps is the minimal number of swaps we need to put the remaining characters in their correct positions.

Each direct or cyclic swap reduces the number of mismatches in the string S by moving the characters towards their respective substrings/positions. By induction, we have shown that at each stage, the algorithm minimizes the total number of swaps needed to fully sort any string up to length $n = k + 1$.