CS 3510: Homework 2B

Due on **Friday** Feb 25

Professor Faulkner

CS 3510 Staff

Problem 1. *(33 points)*
You are given two integer arrays $arr1$ and $arr2$. Your goal is to find the length of the longest common increasing subsequence between these two arrays.

Example:
Input:
$arr1$ = [3, 4, 9, 1]
$arr2$ = [5, 3, 8, 9, 10, 2, 1]
Output: 2 because the longest common increasing subsequence is [3, 9]

a.) (6 points) Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...

The values of $T(i)$ will be the longest common increasing subsequence ending with the arr2(i).

b.) (12 Points) State the recurrence for entries of your table in terms of smaller subproblems. Briefly explain in words why it is correct. Don't forget the base cases here.

For every element in $arr1$ we will search for the element $arr1(i)$ in $arr2$. If we find a match in $arr2$ we will update the value of $T(i)$ to just 1. However, we want to check that if there was a previous smaller common element. During the process of looking for if the elements are in common we are going to take note if we come across an element in $arr2$ that is less than the current $arr1$, if we come across such an element we are going to set a value to the value of that $T(j)$, if we find multiple values less than the current $arr1(i)$ we check if that running value is less than or greater than the $T(j)$ that we subsequentially find. If the $T(j)$ is greater we will set the running value to that. Otherwise, continue otherwise. As a base case if one of the arrays is empty we will return that there is no longest common increasing subsequence and return 0.

c.) (6 points) Use your recurrence relationship to write pseudocode for your algorithm.

```
1  Function LCIS(arr1, n, arr2, m):
       /* base case                                                    */
2      if n or m == 0 then
3          return 0
       /* let T an m long list that is filled with 0s                  */
4      T ← [0]
5      for i in range(n) do
6          current_value = 0
7          for j in range(m) do
               /* We check if both arrays have the common element but does not break loop
                  in case there are multiple instances of the same element */
8              if arr1[i] == arr[j] then
9                  if current + 1 > table[j] then
10                     table[j] = current + 1
               /* Look for previous smaller common elements            */
11             if arr1[i] > arr2[j] then
12                 if T[j] > current then
13                     current = T[j]
       /* return max value of T[i]                                     */
14     return max(T)
```

d.) (9 Points) Determine the running time of your algorithm. Briefly provide justification.

During this algorithm the biggest part we are concerned about are the nested for loops and the contents of those for loops like the if statements more than the initialization of the T array. The for loops are dependent on the size of both $arr1$ and $arr2$ which are n and m respectively. The for loops are $\mathcal{O}(m)$

work $\mathcal{O}(n)$ times. Then the If statements are producing $\mathcal{O}(1)$. Therefore, the full running time of the algorithm is $\mathcal{O}(n)\mathcal{O}(m)\mathcal{O}(1) = \mathcal{O}(nm)$.

Problem 2. *(33 points)*
You and your best friend is playing a game. The game has n stages, each stage can add or deduct points from your score. You and your friend take turns playing the game, starting with your friend. Each person must play exactly 1, 2, or 3 stages on their turn (you cannot skip your turn).

Your goal is to design a dynamic programming algorithm to find the maximum number of points you can earn. You can assume that your friend is helping you maximize your score.

Example:
stages = [1, 3, 10, 6, 5, 2, 4, -5, -2, 3, 7, 9, 8]

Your friend plays the first 2 stages (1, 3)
You play the next 3 stages (10, 6, 5)
Your friend plays the next stage (2)
You play the next stage (4)
Your friend plays the next 3 stages (-5, -2, 3)
You play the next 3 stages (7, 9, 8)

The maximum score you can earn is 49 points.

a.) (6 points) Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...

The values of $T(i, 1)$ will be the greatest value that can be created up to that current index i from the *stages* array and $T[i, 2]$ will store the number of turns you have used on that index ($1, 2$, or 3) and it will store a negative number between -1 and -3 if your friend took their turn last.

b.) (12 Points) State the recurrence for entries of your table in terms of smaller subproblems. Briefly explain in words why it is correct. Don't forget the base cases here.

The base case for this problem will be if the array *stages* has only one element or zero elements in it then we are going to return that the maximum score you can earn is 0 points. For further instances we are going to first check if the ith number in the *stages* array is positive of negative, if it is negative we are going to attempt to give it to our friend and if it is positive we are going to try and take it for ourselves. In our attempt to do this we are going to check $T[i-1, 2]$ if the value is either -3 or 3 we are going to check the numbers in the last three to see if the current $stages[i]$ is either greater than or less than those numbers. If we are comparing trying to add to our score, if we find a number less than the one we are trying to add within the last three we are going to try to give it to our friend, if it is in the $i-3$ position, we are going to have to check $T[i-4, 2]$ to see the j value to see if we are able, if not we are going to have to keep checking back to see if we are able to adjust somewhere. We do the opposite if we are trying to add it to our friend's turn. If the number is not on the $stages[i-3]$ then we are going to give that lowest value to our friend and restart our turn. Lastly, if it is not less than or greater than any of the previous three values we are going to give it to the other person regardless.

c.) (6 points) Use your recurrence relationship to write pseudocode for your algorithm.

```
1  Function BIGSCORE(stages, n):
       /* base case                                                          */
2      if n ≤ 1 then
3          return 0
       /* let T an nx2 long list that is filled with 0s                      */
4      T ← [0, 0]
5      for i in range(n) do
6          if stages[i] > 0 then
7              if T[i-1, 2] < 3 then
8                  T[i, 1] = T[i-1, 1] + stages[i]
9                  T[i, 2] = T[i-1, j] + 1
10             else
                   /* we are going to look for smaller element in last 3     */
11                 if stages[i-2] < stages[i-1] < stages[i] then
12                     T[i-2, 1] = T[i-3, 1]
13                     T[i-2, 2] = -1
14                     T[i-1, 1] = T[i-2, 1] + stages[i-1]
15                     T[i-1, 2] = 1
16                     T[i, 1] = T[i - 1, 1] = stages[i]
17                     T[i, 2] = T[i - 1, 2] + 1
18                 else if stages[i-1] < stages[i - 2] < stages[i] then
19                     T[i-1, 1] = T[i - 2, 1]
20                     T[i - 1, 2] = -1
21                     T[i, 1] = T[i - 1] + stages[i]
22                     T[i , 2] = 1
23                 else if stages[i-3] > stages[i] then
24                     T[i, 1] = T[i-1, 1]
25                     T[i, 2] = -1
26                 else
                       /* I am actually not sure how to pseudo code this but this is where
                          we would see if we can add the T[i-3,1] element to our friend's
                          set of turns etc.                                  */
27         else
               /* same as last If but opposite for friend instead of us.     */
28     return T[n,1]
```

d.) (9 Points) Determine the running time of your algorithm. Briefly provide justification.

In this algorithm we are only iterating over each of the elements and making *several comparisons and ifs, so the runningtime of this algorithm is a brutal* $\mathcal{O}(n)$. *I am sorry I honestly got really stuck on this question and probably came up with an awful answer, but I guess it is better than nothing.*

Problem 3. *(34 points)*
You are given an array of non-negative integers named *arr*. Split *arr* into two subsets that produce the minimum absolute value of the difference between the total sum between the subsets.

Hint: instead solve the following problem: does some subset of the elements of *arr* sum to $\frac{\text{sum}(arr)}{2}$. Then, use the resulting table to solve the original problem.

Example:
Input: *arr* = [10, 7, 2, 7, 1]
Output: 1 because subsets [7, 7] and [10, 2, 1] produce the minimum absolute difference: |(7 + 7) - (10 + 2 + 1)| = 1

a.) (6 points) Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...

The values of $T(i)$ is going to be the running difference of the two subarrays.

b.) (12 Points) State the recurrence for entries of your table in terms of smaller subproblems. Briefly explain in words why it is correct. Don't forget the base cases here.

If there are no elements in the array then we are going to return 0. We are first going to sort the array from greatest to least. We are going to look at $T(i-1)$ if it is negative, we are going to add the next value, but if $T(i-1)$ is nonnegative we are going to subtract the next value. This works because it would in theory add to the now smaller array which would change the difference over time by smaller amounts until all values are used up.

c.) (6 points) Use your recurrence relationship to write pseudocode for your algorithm.

```
1  Function MINABSDIFF(arr, n):
       /* base case                                                    */
2      if n == 0 then
3          return 0
4      Sort arr from greatest to least.
       /* let T an n long list that is filled with −1s                 */
5      T ← [−1]
6      for i in range(n) do
7          if T[i-1] < 0 then
8              T[i] = T[i-1] + arr[i]
9          else
10             T[i] = T[i-1] - arr[i]
11     return T[i]
```

d.) (10 Points) Determine the running time of your algorithm. Briefly provide justification.

Other than sorting the array, we are only running a single for loop over the values of the array. Other than depending on the sorting array the algorithm only has a running time of $\mathcal{O}(n)$ where n is the length of *arr*.