#Weekly Assessment 4

Instructions:

1. Sign the Honor Pledge.
2. In addition to your own notes and books, online resources can also be used but you have to provide reference(s).

Honesty Pledge:

I pledge on my honor that I have not given or received any unauthorized assistance on this exam/assignment.

Indicate your Name / Date: Kyle Matthew Cayanan / March 14, 2023

---

1. Linda claims to have an algorithm that takes an input sequence $S$ and produces an output sequence $T$ that is a sorting of the $n$ elements in $S$.

(a) Give an algorithm, `is_sorted` , that tests in $O(n)$ time if $T$ is sorted. [10 pts]

###Answer:

def is_sorted(input):

```
    i=0

    end=len(input)

    while i<end:
        if i==0:
            if input[0]>input[1]:
                return False
        elif i==end-1:
            if input[i-1]>input[i]:
                return False
        else:
            if input[i-1]>input[i]:
                return False
        i+=1

    return True
```

#check T using is_sorted()

print(is_sorted(T))

(b) Explain why the algorithm `is_sorted` is not sufficient to prove a particular output $T$ to Linda's algorithm is a sorting of $S$. [10 pts]

###Answer:

In question 1.a, I wrote a code that simply checks if an input T is sorted. It succesfully runs in O(n) time. However, assuming that we need to also verify if T is a sequence from S, that means we need to check if the validity of T. Every variable in T must exist in S identically.

You would need a for-loop to get every index of T, then you would need a second for loop to compare a single variable to every element in S. This means that you would have a nested loop that has run time of O(n^2).

(c) Describe what additional information Linda's algorithm could output so that her algorithm's correctness could be established on any given $S$ and $T$ in $O(n)$ time. [10 pts]

```
###Answer:
```

```
Linda could create a dictionary that contains the original position of every element. They key should be the
final position of every element. This way, we can just compare every element [i] to the value of the
dictionary.

The dictionary will contain the index positions, and these index positions will be attached to S.

For i in T:
    if i!=S[dictionary[i]]
```

2. Suppose we are given an $n$-element sequence $S$ such that each element in $S$ represents a different vote for president, where each vote is given as an integer representing a particular candidate, yet the integers may be arbitrariily large (even if the number of candidates is not). Design an $O(n \log n)$- time algorithm to see who wins the election $S$ represents, assuming the candidate with the most votes wins. [20 pts]

```
###Answer:

For this problem, the best solution that has O(nlog(n)) time is the merge sort. The reference I have used
for merge sort is this website: https://www.programiz.com/dsa/merge-sort.

Afterwards, I modified the code to return the largest number after the merge sort is completed.

Remarks: the tutorial taught me how to use recursion using array modificaiton instead of array addition.
This is the reason why there are no "return" functions in the code. Instead of return, the array object is
modified directly.

This is the actual code:

#variable for testing
merger=[2312,4143,653,133]

#actual merge sort function
def mergeSort(array):

    #while array is still greater than two elements, sort them into half
    #this will repeat until list size of two. The two elements will be sorted to left side and right side
    if len(array) > 1:

        #  r is midpoint
        r = len(array)//2
        L = array[:r]
        M = array[r:]


        #recursion to sort the two halves
        #if the input is a single element, no array modification will occur
        mergeSort(L)
        mergeSort(M)

        #reset counters
        i = j = k = 0

        # left side and right side is compared
        # they will be compared using index i and index j
        while i < len(L) and j < len(M):

            #if left side is smaller, it will be sorted to the output array
            if L[i] < M[j]:
                array[k] = L[i]

                #i will only iterate if the left side is the lower element
                i += 1

            #if right side is smaller, it will be sorted to the output array
            else:
                array[k] = M[j]
                #j will only iterate if the right side is the lower element
                j += 1

            #k is the index for the output array
            #it will only iterate if an element is added to the final output
```

```
            k += 1


        #if either L or M runs out early, the following loops will add the remaining elements to the array
        #recall that this is under the assumption that L and H are already sorted

        #while Left still has remaining elements, add them to the array
        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

        #while Right still has remaining elements, add them to the array
        while j < len(M):
            array[k] = M[j]
            j += 1
            k += 1

#function for testing
#calls merge sort using the test list "merger"
def vote_checker(input):
    #fetch the sorted votes using merge sort
    win_list=merge_sort(input)

    #get the largest vote from the end of the list
    winner=win_list[-1]

    #return the winning vote
    return winner

print(vote_checker(merger))
```

3. Suppose we are given two $n$-element sorted sequences $A$ and $B$ each with distinct elements, but potentially some elements that are in both sequences.

(a) Describe an $O(n)$-time method for computing a sequence representing the union $A \cup B$ (with no duplicates) as a sorted sequence. [10 pts]

```
###Answer:

For this problem, I will implement a similar concept that was applied in merge sort. I will use three
variables to use as counters that will compare A and B. However, I modified this function to have a third
case. In a case where A and B have an equal element, only 1 instance will be added. Then both i and j will
be iterated. Then finally I will sort the list to pop a variable whenever a duplicate exists.

This will be a linear iteration, therefore it will have O(n) time.
```

(b) Implement your method in (a) for integer elements. Perform an experimental performance test of its running times and show a summary graph. [20 pts]

In [ ]:
```
# YOUR CODE HERE

The actual code:

testA=[1,2,3,4,9]
testB=[1,3,4,5,6,7,8]

def Linear_Sort(A,B):
    array=A+B
    i = j = k = 0

    while i < len(A) and j < len(B):

        if A[i] < B[j]:
            array[k] = A[i]

            i += 1

        elif A[i] == B[j]:
            array[k] = A[i]
            array.pop(-1)

            i += 1
            j += 1


        else:
            array[k] = B[j]

            j += 1


        k += 1

    while i < len(A):
        array[k] = A[i]
        i += 1
        k += 1

    #while Right still has remaining elements, add them to the array
    while j < len(B):
        array[k] = B[j]
        j += 1
        k += 1

    return array

print(Linear_Sort(testA,testB))
```

Benchmarking Results here.

4. Look at the two different algorithms for finding the maximum contiguous subarray below: `maxSubArraySum` and `maxSubArraySum2` .

In [3]:
```
# Function to find the maximum contiguous subarray
import sys

def maxCrossingSum(arr, l, m, h):

    # Include elements on left of mid.
    sm = 0
    left_sum = -10000

    for i in range(m, l-1, -1):
        sm = sm + arr[i]

        if (sm > left_sum):
            left_sum = sm

    # Include elements on right of mid
    sm = 0
    right_sum = -1000
    for i in range(m, h + 1):
        sm = sm + arr[i]

        if (sm > right_sum):
            right_sum = sm

    # Return sum of elements on left and right of mid
    # returning only left_sum + right_sum will fail for [-2, 1]
    return max(left_sum + right_sum - arr[m], left_sum, right_sum)


# Returns sum of maximum sum subarray in arr[l..h]
def maxSubArraySum(arr, l, h):
    #Invalid Range: low is greater than high
    if (l > h):
        return -10000
    # Base Case: Only one element
    if (l == h):
        return arr[l]

    # Find middle point
    m = (l + h) // 2

    # Return maximum of following three possible cases
    # a) Maximum subarray sum in left half
    # b) Maximum subarray sum in right half
    # c) Maximum subarray sum such that the
    #     subarray crosses the midpoint
    return max(maxSubArraySum(arr, l, m-1),
               maxSubArraySum(arr, m+1, h),
               maxCrossingSum(arr, l, m, h))

def maxSubArraySum2(a, size):

    max_so_far = -sys.maxsize - 1
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if (max_so_far < max_ending_here):
            max_so_far = max_ending_here

        if max_ending_here < 0:
            max_ending_here = 0
    return max_so_far


# Driver function to check the above function

a = [-2, -3, 4, -1, -2, 1, 5, -3]
print ("Maximum contiguous sum is "+ str(maxSubArraySum(a, 0, len(a)-1)))
print ("Maximum contiguous sum is "+ str(maxSubArraySum2(a, len(a))))
```

```
Maximum contiguous sum is 7
Maximum contiguous sum is 7
```

(a) Which algorithms uses a Divide-and-conquer approach and which one does not? Explain your answer. [10 pts]

```
###Answer:
The function that uses divide and conquer is "maxSubArraySum2". This is because it solves for the midpoint.
Afterwards, the midpoint is used to recursively call itself. This is repeated for the left half and right
half. The recursion continues until the leftmost (first) element is equal to the rightmost (last) element.
This means that the recursion repeats until the array is size 1.
```

(b) The function `maxSubArraySum2` is an implementation of Kadane's Method. Explain what it does to solve the maximum contiguous subarray problem? What is the big Oh time complexity of this algorithm? [10 pts]

```
###Answer:

I tried to interpret this function using this online source: https://www.interviewbit.com/blog/maximum-
subarray-sum/#:~:text=Kadane%27s%20Algorithm%20is%20an%20iterative,ending%20at%20the%20previous%20position.

How it solved the problem is to first set max_so_far to the largest negative integer, then max_ending_here
to zero. Because max_so_far is set to the lowest integer, it will always be less than any sum so far.

For the actual iteration, it will use a for loop to iterate through all n inputs. Max_ending_here simply
solves for the sum of inputs. If its sum is larger than max_so_far, max_so_far will be updated to that new
value.

Any negative sum will become max_so_far new value. Then max_ending_here will reset to zero. This will repeat
everytime a negative input is entered.

When a positive input is entered, it will be added to zero, and that will become the new max_so_far. Repeat
until all n complete.

To summarize: max_ending_here gets the sum of each element between iterations. Max_so_far keeps track of the
largest sum ever reached.

Finally, this action is only repeated for the entire length of the list. This means that the runtime is
O(n).
```