Instructions

> This assignment is to be completed and uploaded as a python3 notebook.

This problem set covers the following topics:

- Basics of algorithms: correctness and running time complexity.
- Time Complexity: O, big-Omega and big-Theta Notations.
- Proving Correctness of Algorithms through Inductive Invariants.
- Merge Sort: Proving Correctness.

Important Note

Although this is a programming assignment, we have asked you to work on the "design" and provided opportunities for you to analyze your solution and describe your design. **However, those parts will not be graded.** You are welcome to compare your answers against our solutions once you have completed the assignments. Our solutions are provided at the very end.

Problem 1: Find Crossover Indices.

You are given data that consists of points $(x_0, y_0), \ldots, (x_n, y_n)$, wherein $x_0 < x_1 < \ldots < x_n $, and $y_0 < y_1 \ldots < y_n$ as well.

Furthermore, it is given that $y_0 < x_0$ and $ y_n > x_n$.

Find a "cross-over" index i between 0 and $n-1$ such that $ y_i \leq x_i$ and $y_{i+1} > x_{i+1}$.

Note that such an index must always exist (convince yourself of this fact before we proceed).

Example

i	0	1	2	3	4	5	6	7
x_i	0	2	4	5	6	7	8	10
y_i	-2	0	2	4	7	8	10	12

Your algorithm must find the index $i = 3$ as the crossover point.

On the other hand, consider the data

i	0	1	2	3	4	5	6	7
x_i	0	1	4	5	6	7	8	10
y_i	-2	1.5	2	4	7	8	10	12

We have two cross over points. Your algorithm may output either $i = 0$ or $i = 3$.

(A) Design an algorithm to find an index $i \in \{0, 1, \ldots, n-1\}$ such that $x_i \geq y_i$ but $x_{i+1} < y_{i+1}$.

Describe your algorithm using python code for a function *findCrossoverIndexHelper(x, y, left, right)*

- `x` is a list of x values sorted in increasing order.
- `y` is a list of y values sorted in increasing order.
- `x` and `y` are lists of same size (`n`).
- `left` and `right` are indices that represent the current search region in the list such that 0 <= `left` < `right` <= `n`

Your solution must use *recursion*.

Hint: Modify the binary search algorithm we presented in class.

```python
#First write a "helper" function with two extra parameters
# left, right that ddedscribes the search region as shown below
def findCrossoverIndexHelper(x, y, left, right):
    # Note: Output index i such that
    #         left <= i <= right
    #         x[i] <= y[i]
    # First, Write down our invariants as assertions here
    assert(len(x) == len(y))
    assert(left >= 0)
    assert(left <= right-1)
    assert(right < len(x))
    # Here is the key property we would like to maintain.
    assert(x[left] > y[left])
    assert(x[right] < y[right])

    # your code here
    if right == left + 1:
        if x[left] >= y[left] and x[right] < y[right]:
            return left  # Found the crossover index
        else:
            return -1  # Should not happen based on problem constraints

    mid = left + (right - left) // 2  # Find the mid-point

    if x[mid] >= y[mid]:
        # The crossover point could be at mid or to the right of mid
        return findCrossoverIndexHelper(x, y, mid, right)
    else:
        # The crossover point is to the left of mid
        return findCrossoverIndexHelper(x, y, left, mid)


#Define the function findCrossoverIndex that wil
# call the helper function findCrossoverIndexHelper
def findCrossoverIndex(x, y):
```

```python
    assert(len(x) == len(y))
    assert(x[0] > y[0])
    n = len(x)
    assert(x[n-1] < y[n-1]) # Note: this automatically ensures n >= 2 why?
    # your code here


    return findCrossoverIndexHelper(x, y, 0, n-1)



# BEGIN TEST CASES
j1 = findCrossoverIndex([0, 1, 2, 3, 4, 5, 6, 7], [-2, 0, 4, 5, 6, 7, 8, 9])
print('j1 = %d' % j1)
assert j1 == 1, "Test Case # 1 Failed"

j2 = findCrossoverIndex([0, 1, 2, 3, 4, 5, 6, 7], [-2, 0, 4, 4.2, 4.3, 4.5, 8, 9])
print('j2 = %d' % j2)
assert j2 == 1 or j2 == 5, "Test Case # 2 Failed"

j3 = findCrossoverIndex([0, 1], [-10, 10])
print('j3 = %d' % j3)
assert j3 == 0, "Test Case # 3 failed"

j4 = findCrossoverIndex([0,1, 2, 3], [-10, -9, -8, 5])
print('j4 = %d' % j4)
assert j4 == 2, "Test Case # 4 failed"

print('Congratulations: all test cases passed - 10 points')
#END TEST CASES
```

```
j1 = 1
j2 = 1
j3 = 0
j4 = 2
Congratulations: all test cases passed - 10 points
```

(B, 0 points) What is the running time of your algorithm above as a function of the input array size n?

This portion is not graded. You are encouraged to answer it as part of your programming assignment however

YOUR ANSWER HERE

Problem 2 (Find integer cube root.)

The integer cube root of a positive number n is the smallest number i such that $i^3 \leq n$ but $(i+1)^3 > n$.

For instance, the integer cube root of 100 is 4 since $4^3 \leq 100$ but $5^3 > 100$. Likewise, the integer cube root of 1000 is 10.

Write a function `integerCubeRootHelper(n, left, right)` that searches for the integer cube-root of `n` between `left` and `right` given the following pre-conditions:

- $n \geq 1$
- $\text{left} < \text{right}$.
- $\text{left}^3 < n$
- $\text{right}^3 > n$.

```
def integerCubeRootHelper(n, left, right):
    cube = lambda x: x * x * x # anonymous function to cube a number
    assert(n >= 1)
    assert(left < right)
    assert(left >= 0)
    assert(right < n)
    assert(cube(left) < n), f'{left}, {right}'
    assert(cube(right) > n), f'{left}, {right}'
    # your code here

    if right == left + 1:
        if (cube(left)<n) and (cube(right)>n):
            return left  # Found the cuberoot
        elif cube(left)==n:
            return left
        elif cube(right)==n:
            return right
        else:
            return -1  # Should not happen based on problem constraints

    mid = left + (right - left) // 2  # Find the mid-point

    if cube(mid)<n:
        # The cuberoot point could be at mid or to the right of mid
        return integerCubeRootHelper(n, mid, right)
    elif cube(mid)>n:
        # The cuberoot point is to the left of mid
        return integerCubeRootHelper(n, left, mid)
    elif cube(mid)==n:
        return mid
```

```python
# Write down the main function
def integerCubeRoot(n):
    assert( n > 0)
    if (n == 1):
        return 1
    if (n == 2):
        return 1
    return integerCubeRootHelper(n, 0, n-1)
```

```python
assert(integerCubeRoot(1) == 1)
assert(integerCubeRoot(2) == 1)
assert(integerCubeRoot(4) == 1)
assert(integerCubeRoot(7) == 1)
assert(integerCubeRoot(8) == 2)
assert(integerCubeRoot(20) == 2)
assert(integerCubeRoot(26) == 2)
for j in range(27, 64):
    assert(integerCubeRoot(j) == 3)
for j in range(64,125):
    assert(integerCubeRoot(j) == 4)
for j in range(125, 216):
    assert(integerCubeRoot(j) == 5)
for j in range(216, 343):
    assert(integerCubeRoot(j) == 6)
for j in range(343, 512):
    assert(integerCubeRoot(j) == 7)
print('Congrats: All tests passed! (10 points)')
```

```
Congrats: All tests passed! (10 points)
```

(B, 0 points)

The inductive invariant for the function `integerCubeRootHelper(n, left, right)` that ensures that the overall algorithm for finding the integer cube root is correct is :

$$\text{left}^3 < n \text{ and } \text{right}^3 > n$$

Use the inductive invariant to establish that the integer cube root of n (the final answer we seek) must lie between `left` and `right`.

In other words, let j be the integer cube root of n.

Prove using the inductive invariant and property of the integer cube root j that:

$$\text{left} \leq j < \text{right}$$

Note that this part is not graded. You are encouraged to answer it for your own understanding.

YOUR ANSWER HERE

(C, 0 points)

Prove that your solution for `integerCubeRootHelper` maintains the overall inductive invariant from part (B). I.e, if the function were called with

$0 \leq \text{left} < \text{right} < n$, and $\text{left}^3 < n$ and $\text{right}^3 > n$.

Any subsequent recursive calls will have their arguments that also satisfy this property. Model your answer based on the lecture notes for binary search problem provided.

Note that this part is not graded. You are encouraged to answer it for your own understanding.

YOUR ANSWER HERE

Problem 3 (Develop Multiway Merge Algorithm, 15 points).

We studied the problem of merging 2 sorted lists `lst1` and `lst2` into a single sorted list in time $\Theta(m+n)$ where m is the size of `lst1` and n is the size of `lst2`. Let `twoWayMerge(lst1, lst2)` represent the python function that returns the merged result using the approach presented in class.

In this problem, we will explore algorithms for merging `k` different sorted lists, usually represented as a list of sorted lists into a single list.

(A, 0 points)

Suppose we have k lists that we will represent as `lists[0]`, `lists[1]`, ..., `lists[k-1]` for convenience and the size of these lists are all assumed to be the same value n.

We wish to solve multiway merge by merging two lists at a time:

```
mergedList = lists[0] # start with list 0
for i = 1, ... k-1 do
    mergedList = twoWayMerge(mergedList, lists[i])
return mergedList
```

Knowing the running time of the `twoWayMerge` algorithm as mentioned above, what is the overall running time of the algorithm in terms of n, k.

Note that this part is not graded. You are encouraged to answer it for your own understanding.

YOUR ANSWER HERE

(B) Implement an algorithm that will implement the k way merge by calling `twoWayMerge` repeatedly as follows:

1. Call `twoWayMerge` on consecutive pairs of lists `twoWayMerge(lists[0], lists[1])`, ... , `twoWayMerge(lists[k-2], lists[k-1])` (assume k is even).
2. Thus, we create a new list of lists of size `k/2`.
3. Repeat steps 1, 2 until we have a single list left.

```
def twoWayMerge(lst1, lst2):
    # Implement the two way merge algorithm on
    #          two ascending order sorted lists
    # return a fresh ascending order sorted list that
    #          merges lst1 and lst2
    # your code here
    tmp_store = []
    #for i,j in zip(range(len(lst1)),range(len(lst2))):
    i = 0
    j = 0
    while (i <= len(lst1)-1) and (j<=len(lst2)-1):
        if lst1[i]<lst2[j]:
            tmp_store.append(lst1[i])
            i+=1
        else:
            tmp_store.append(lst2[j])
            j+=1
    while i <= len(lst1)-1:
        tmp_store.append(lst1[i])
        i+=1
    while j <= len(lst2)-1:
        tmp_store.append(lst2[j])
        j+=1
    return tmp_store


# given a list_of_lists as input,
#   if list_of_lists has 2 or more lists,
#        compute 2 way merge on elements i, i+1 for i = 0, 2, ...
#   return new list of lists after the merge
#   Handle the case when the list size is odd carefully.
def oneStepKWayMerge(list_of_lists):
    if (len(list_of_lists) <= 1):
        return list_of_lists
    ret_list_of_lists = []
    k = len(list_of_lists)
    for i in range(0, k, 2):
        if (i < k-1):
            ret_list_of_lists.append(twoWayMerge(list_of_lists[i], list_of_lists[i+1]))
        else:
            ret_list_of_lists.append(list_of_lists[k-1])
```

```python
    return ret_list_of_lists

# Given a list of lists wherein each
#    element of list_of_lists is sorted in ascending order,
# use the oneStepKWayMerge function repeatedly to merge them.
# Return a single merged list that is sorted in ascending order.
def kWayMerge(list_of_lists):
    k = len(list_of_lists)
    if k == 1:
        return list_of_lists[0]
    else:
        new_list_of_lists = oneStepKWayMerge(list_of_lists)
        return kWayMerge(new_list_of_lists)
```

```python
# BEGIN TESTS
lst1= kWayMerge([[1,2,3], [4,5,7],[-2,0,6],[5]])
assert lst1 == [-2, 0, 1, 2, 3, 4, 5, 5, 6, 7], "Test 1 failed"

lst2 = kWayMerge([[-2, 4, 5 , 8], [0, 1, 2], [-1, 3,6,7]])
assert lst2 == [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8], "Test 2 failed"

lst3 = kWayMerge([[-1, 1, 2, 3, 4, 5]])
assert lst3 == [-1, 1, 2, 3, 4, 5], "Test 3 Failed"

print('All Tests Passed = 15 points')
#END TESTS
```

```
All Tests Passed = 15 points
```

(C, 0 points)

What is the overall running time of the algorithm in (B) as a function of n and k?

Note that this part is not graded. You are encouraged to answer it for your own understanding.

YOUR ANSWER HERE

Solutions to the Conceptual (Non Coding) Questions

Problem 1B

Note that the running time of *findCrossOverIndexHelper* for inputs x, y of size n is $\Theta(\log(n))$. This is because, each iteration of the algorithm halves the search region and the algorithm terminates when the search region has size 2. This requires at most $\Theta(\log(n))$ iterations by the same argument as that presented for binary search in the lecture video.

Problem 2B

The reason we can conclude left $\leq j <$ right is : We note that since j is assumed to be integer cube root of n, we have $j^3 \leq n$ and $(j+1)^3 > n$. We have left $< j+1$ and likewise right $> j$. Therefore, left $\leq j <$ right.

Problem 2C

```
mid = (left + right)//2
if (cube(mid)  <= n and cube(mid+1) > n):
    return mid
elif (cube(mid) > n):
    return integerCubeRootHelper(n, left, mid) # Call 1
else:
    return integerCubeRootHelper(n, mid, right) # Call 2
```

If Call 1 happens, we note that `cube(mid) > n`. However, `cube(left) < n` is already true since the value of `left` is unchanged. Thus Call 1 satisfies the invariant.

Note that Call 2 will satisfy the property because `cube(right) > n` and the call will only happen if `cube(mid+1) <= n`. This implies that `cube(mid)< n`. Therefore, we conclude that `Call 2` will satisfy the invariant, as well.

Problem 3A

The overall running time is $\Theta(n \times ((k-1) + \cdots + 1)) = \Theta(n \times k^2)$

Problem 3C

At iteration i, the list of lists has size $k/2^{i-1}$ with each element of size $n \times 2^{i-1}$. The number of merge operations is $k/2^i$ with each merge operation taking $n \times 2^i$ time. The overall work done at the i^{th} iteration remains $k \times n$. There are $\Theta(\log(k))$ iterations in all. Therefore, the overall complexity is $\Theta(nk\log(k))$.

That's All Folks!