Assignment1/ProblemSet1\_Solutions.ipynb

**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 (𝑥0,𝑦0),…,(𝑥𝑛,𝑦𝑛), wherein 𝑥0<𝑥1<…<𝑥𝑛, and  𝑦0<𝑦1…<𝑦𝑛 as well.

Furthermore, it is given that 𝑦0<𝑥0 and 𝑦𝑛>𝑥𝑛.

Find a "cross-over" index 𝑖 between 0 and 𝑛−1 such that  𝑦𝑖≤𝑥𝑖 and 𝑦𝑖+1>𝑥𝑖+1.

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

**Example**

𝑖𝑥𝑖𝑦𝑖00−2120242354467578681071012

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

On the other hand, consider the data

𝑖𝑥𝑖𝑦𝑖00−2111.5242354467578681071012

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∈{0,1,…,𝑛−1} such that 𝑥𝑖≥𝑦𝑖 but 𝑥𝑖+1<𝑦𝑖+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.

**def** findCrossoverIndex(x, y):

n **=** len(x)

**assert** n **==** len(y), "Arrays must have the same length"

**assert** x[n**-**1] **<** y[n**-**1], "x[n-1] must be less than y[n-1]"

*# Iterate through the lists and find the crossover index*

**for** i **in** range(n **-** 1):

**if** x[i] **>** y[i] **and** x[i **+** 1] **<=** y[i **+** 1]:

**return** i

**return** **-**1 *# If no crossover happens, return -1 (although the problem assumes there is a crossover)*

*#Define the function findCrossoverIndex that wil*

*# call the helper function findCrossoverIndexHelper*

**def** findCrossoverIndex(x, y):

**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*

**def** findCrossoverIndexHelper(x, y, left, right):

*# Invariant assertions to ensure correctness*

**assert**(len(x) **==** len(y)) *# x and y must have the same length*

**assert**(left **>=** 0) *# left must be non-negative*

**assert**(left **<=** right **-** 1) *# left must be less than or equal to right - 1*

**assert**(right **<** len(x)) *# right must be less than the length of x and y*

​

*# Key properties*

**assert**(x[left] **>** y[left]) *# x[left] must be greater than y[left]*

**assert**(x[right] **<** y[right]) *# x[right] must be less than y[right]*

*# Calculate the middle index of the current search region*

mid **=** (left **+** right) **//** 2

*# Check if the crossover condition holds at mid*

**if** x[mid] **<=** y[mid] **and** x[mid **+** 1] **>** y[mid **+** 1]:

**return** mid

*# Otherwise, narrow down the search region*

**if** x[mid] **>** y[mid]:

*# We need to search the left half*

**return** findCrossoverIndexHelper(x, y, left, mid **-** 1)

**else**:

*# We need to search the right half*

**return** findCrossoverIndexHelper(x, y, mid **+** 1, right)

​

*# Main function to call the helper function*

**def** findCrossoverIndex(x, y):

**assert**(len(x) **==** len(y)) *# x and y must have the same length*

**assert**(x[0] **>** y[0]) *# x[0] must be greater than y[0]*

n **=** len(x)

**assert**(x[n**-**1] **<** y[n**-**1]) *# x[n-1] must be less than y[n-1]*

​

*# Call the helper function with the full range of indices*

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

​

*# Example usage*

x **=** [**-**2, 0, 2, 4, 6]

y **=** [**-**1, 1, 3, 5, 7]

result **=** findCrossoverIndex(x, y)

print("Crossover index:", result) *# Should print: Crossover index: 2*

​

*# 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*

**(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 The time complexity of this binary search approach is O(logn)O(logn), where n is the input number.

**Problem 2 (Find integer cube root.)**

The integer cube root of a positive number 𝑛 is the smallest number 𝑖 such that 𝑖3≤𝑛 but (𝑖+1)3>𝑛.

For instance, the integer cube root of 100 is 4 since 43≤100 but 53>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≥1
* left<right.
* left3<𝑛
* right3>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)

**assert**(cube(right) **>** n)

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*

​

​

**def** integerCubeRoot(n):

**assert**(n **>** 0)

**if** n **==** 1:

**return** 1

**if** n **==** 2:

**return** 1

**return** integerCubeRootHelper(n, 0, n **-** 1)

​

​

*# 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)

*# 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)

**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)')

**(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 :

left3<𝑛andright3>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:

left≤𝑗<right

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

YOUR ANSWER HERE Proof that left≤j<rightleft≤j<rightInductive Invariant: We maintain that left3<nleft 3nright 3 >n during each step of the recursion. Final Condition: The algorithm works by recursively narrowing the range between leftleft and rightright, ensuring that: left3≤nand(left+1)3>nleft 3≤nand(left+1) 3 >n Therefore, at the end of the algorithm, left left will be the integer cube root jj, and right=j+1 right=j+1. Conclusion: Thus, by the properties of the search and the invariant, we conclude that the integer cube root jj lies between leftleft and rightright, i.e., left≤j<right left≤j<right.

**(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≤left<right<𝑛, and  left3<𝑛 and right3>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 HEREThe inductive invariant is:

left3<nleft 3nright 3 >n. Proof: Base Case: When left=right−1left=right−1, the function returns the correct integer cube root, satisfying left3<nleft 3nright 3 >n. Recursive Step: If mid3>nmid 3 >n, we recurse with the range [left,mid][left,mid]. The condition left3<nleft 3<n still holds.If mid3≤nmid 3 ≤n, we recurse with the range [mid,right][mid,right]. The condition right3>nright 3 >n still holds. Conclusion: At every step, the invariant left3<nleft 3 nright 3 >n is preserved. Therefore, the final result will satisfy left≤j<rightleft≤j<right, where jj is the integer cube root of nn.

**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 Θ(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 The overall running time of the algorithm is: ( \mathcal{O}(n \cdot k) )

Explanation:

* Each merge operation in twoWayMerge(mergedList, lists[i]) takes ( \mathcal{O}(n) ) time since we are merging two lists of size ( n ).
* We perform ( k-1 ) such merge operations.

Thus, the total time complexity is ( \mathcal{O}(n \cdot (k-1)) ), which simplifies to ( \mathcal{O}(n \cdot k) ).

**(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(list1, list2):

i, j **=** 0, 0

merged **=** []

**while** i **<** len(list1) **and** j **<** len(list2):

**if** list1[i] **<** list2[j]:

merged.append(list1[i])

i **+=** 1

**else**:

merged.append(list2[j])

j **+=** 1

merged.extend(list1[i:])

merged.extend(list2[j:])

**return** merged

​

**def** oneStepKWayMerge(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[i]) *# Last unmerged list if odd*

**return** ret\_list\_of\_lists

​

**def** kWayMerge(list\_of\_lists):

**while** len(list\_of\_lists) **>** 1:

list\_of\_lists **=** oneStepKWayMerge(list\_of\_lists)

**return** list\_of\_lists[0] *# Final merged list*

*# 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])

**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)

*# 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*

**(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 HEREThe overall running time of the algorithm in **(B)**, which performs a multiway merge using repeated two-way merges, can be analyzed as follows:

Key points:

* **Step 1**: At each step, we merge pairs of lists. The merging of two lists of size ( n ) takes ( \mathcal{O}(n) ) time.
* **Step 2**: After the first round of merging, we halve the number of lists. Thus, after each round of merging, the number of lists is reduced by half.
* **Step 3**: The process continues recursively until only one list remains.

Time Analysis:

* In the first round, we merge ( k ) lists. Since there are ( k/2 ) pairs of lists to merge, each merge takes ( \mathcal{O}(n) ) time, so the total time for this round is: [ \mathcal{O}(n \cdot k/2) ]
* In the second round, we merge ( k/2 ) lists. There are ( k/4 ) pairs to merge, so the total time for this round is: [ \mathcal{O}(n \cdot k/4) ]
* This process continues, and after ( \log\_2 k ) rounds, the number of lists will reduce to one.

Thus, the total time complexity is the sum of the work done at each round: [ T(n, k) = \mathcal{O}(n \cdot k/2) + \mathcal{O}(n \cdot k/4) + \mathcal{O}(n \cdot k/8) + \dots + \mathcal{O}(n) ]

This is a geometric series with the sum approximately equal to ( \mathcal{O}(n \cdot k) ).

Final Time Complexity: Therefore, the overall running time of the algorithm is: [ \mathcal{O}(n \cdot k \cdot \log k) ]

Explanation:

* ( n ) is the size of each list.
* ( k ) is the number of lists to merge.
* The ( \log k ) factor comes from the number of rounds of merging, as we reduce the number of lists by half at each step.

Thus, the overall time complexity is ( \mathcal{O}(n \cdot k \cdot \log k) ).

**Solutions to the Conceptual (Non Coding) Questions**

**Problem 1B**

Note that the running time of *findCrossOverIndexHelper* for inputs 𝑥,𝑦 of size 𝑛 is Θ(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 Θ(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≤𝑗<right**is :**

We note that since 𝑗 is assumed to be integer cube root of 𝑛, we have 𝑗3≤𝑛 and  (𝑗+1)3>𝑛. We have left<𝑗+1 and likewise right>𝑗. Therefore, left≤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 Θ(𝑛×((𝑘−1)+⋯+1))=Θ(𝑛×𝑘2)

**Problem 3C**

At iteration 𝑖, the list of lists has size 𝑘/2𝑖−1 with each element of size 𝑛×2𝑖−1. The number of merge operations is 𝑘/2𝑖 with each merge operation taking 𝑛×2𝑖 time. The overall work done at the 𝑖𝑡ℎ iteration remains 𝑘×𝑛. There are Θ(log(𝑘)) iterations in all. Therefore, the overall complexity is Θ(𝑛𝑘log(𝑘)).

**That's All Folks!**

Thank you for stopping in -Sulay Cay ☺