1. (20 points) Suppose there are two sorted arrays A[1 . . . n] and B[1 . . . n], each having n elements (note that they may NOT be integers) sorted in ascending order. Given a number x, we want to find an element A[i] from A and an element B[j] from B such that A[i] + B[j] = x, or report that no such two elements exist.

Design an O(n) time algorithm for the problem.

* Algorithm Description

Using two “key to value data structures” or “maps” my algorithm will solve the problem. The maps will store the complement of the item in the corresponding list. The complement is defined as, x - A[i] or x - B[j]. In other words its the number needed to get to A[i] + B[j]. The value will be the index i or j that will complete the complement. So the key is the complement and the value is the index of the “other number”. Map1 will store the complements from B for A and map2 will store vise versa. Over one pass of both lists the appropriate numbers will be found. If there is no combination then a “none” will be printed out.

* Pseudocode

if n == 0:

print(“none”)

return

for i = 0; i < n; i++:

let a = A[i]

let b = B[i]

map1.put(x - b, i)

map2.put(x - a, i)

if map1.containsKey(a):

print(a, B[map1.get(a)])

return

if map2.containsKey(b):

print(A[map2.get(b)], b)

return

print(“none”)

* Correctness

I check every number from list1 with every number from list2. I only use one for loop.

* Time Analysis

Because there is only one loop with no recursion and in the loop only read, write, and compare operations are being completed we can be sure that the time complexity for my algorithm is O(n).

2. (20 points) You are given k sorted lists L1, L2, . . . , Lk, with 1 ≤ k ≤ n, such that the total number of the elements in all k lists is n. Note that different lists may have different numbers of elements. We assume that the elements in each sorted list Li, for any 1 ≤ i ≤ k, are already sorted in ascending order. Design a divide-and-conquer algorithm to sort all these n numbers. Your algorithm should run in O(n log k) time (instead of O(n log n) time). Note: An O(n log k) time algorithm would be better than an O(n log n) time one when k is sufficiently smaller than n. For example, if k = O(log n), then n log k = O(n log log n), which is strictly smaller than n log n (i.e., n log log n = o(n log n)).

The following gives an example. There are five sorted lists (i.e., k = 5). Your algorithm needs to sort the numbers in all these lists.

L1 : 3, 12, 19, 25, 36

L2 : 34, 89

L3 : 17, 26, 87

L4 : 28

L5 : 2, 10, 21, 29, 55, 59, 61

* Algorithm Description

This problem reminds me of the merge sort algorithm. The key difference is that I am given k lists that are already sorted. Thus the first part of the merge sort algorithm can be skipped. Instead of breaking apart the lists and then bringing them together I can just bring them together.

* Pseudocode

Given L where L is a list of k lists

if L has no lists {

return none

}

if L has one list {

return that one list

}

let C = merge(L.remove(0), L.remove(1))

while L is not empty {

C = merge(L.remove(0), C)

}

return C

merge(A, B) {

let C be a new list

while A is not empty and B is not empty {

if A.get(0) > B.get(0) {

C.add(B.remove(0))

}

else {

C.add(A.remove(0))

}

}

while A is not empty {

C.add(A.remove(0))

}

while B is not empty {

C.add(B.remove(0))

}

return C

}

* Correctness

I have tested my algorithm with an implementation of it in java and it works. It works because I go over every list in L and I merge them together building a combined list one list at a time.

* Time Analysis

My algorithm has as a time complexity of O(n\*log(k)). I know this because I reduce the problem into log(k) levels of merging where at every level I merge n elements.

3. (30 points) Let A[1 · · · n] be an array of n distinct numbers (i.e., no two numbers are equal). If i < j and A[i] > A[j], then the pair (A[i], A[j]) is called an inversion of A.

You are asked to answer the following questions.

(a) List all inversions of the array {4, 2, 9, 1, 7}. (5 points)

Indexes of inversions:

0, 1

1, 3

0, 3

2, 3

2, 4

(b) What array with elements from the set {1, 2, . . . , n} has the most inversions? How many inversions does it have? (5 points)

If the set has n elements then I would say there are about n inversions.

(c) Give a divide-and-conquer algorithm that computes the number of inversions in array A in O(n log n) time. (Hint: Modify merge sort.) (20 points)

* Algorithm Description

This algorithm is a variation of the merge sort algorithm like the hint suggested. The algorithm first splits up the given list until only single elements remain. Then the merging starts. While merging two sublists a comparison is made. If A[0] > B[0] that means that a swap needs to happen. The amount of elements in A is the amount of inversions for that swap. When all the sublists have been recombined in sorted order and all the inversion counts have been totals that is when the total number of inversions for the list is known.

* Pseudocode

Given L where L is a list of elements

L, count = mergesort()

print(L, count)

mergesort(L) {

if L is empty or L has one element return L, 0

H1 = the first half of L

H2 = the second half of L

H1, count1 = mergesort(H1)

H2, count2 = mergesort(H2)

return merge(H1, H2, count1+count2)

}

merge(A, B, count) {

C is a new list

while (A has elements and B has elements) {

if (A.get(0) > B.get(0)) {

count += A.size()

C.add(B.remove(0))

}

else C.add(A.remove(0))

}

while (A has elements) C.add(A.remove(0))

while (B has elements) C.add(B.remove(0))

return C, count

}

* Correctness

I have tested my algorithm with an implementation of it in java and it works. I also know that it works because my algorithm is just mergesort with a count variable being accounted for. Everytime I am merging two sublists I check if A[0] > B[0] if this is true that means that B[0] is an inversion of every element in A.

* Time Analysis

The time complexity for my algorithm is O(n\*log(n)) because my algorithm has n levels of merging and it goes over n elements in total. Thus n\*log(n). This also makes sense because my algorithm is basically mergesort.

4. (20 points) Solve the following recurrences (you may use any of the methods we studied in class). Make your bounds as small as possible (in the big-O notation). For each recurrence, T(n) = O(1) for n ≤ 1.

(a)

Master Theorem case 3.

(b)

Master Theorem case 1.

(c)

Master Theorem case 2.

(d)

Master Theorem case 1.

5. (20 points) You are consulting for a small computation-intensive investment company, and they have the following type of problem that they want to solve. A typical instance of the problem is the following. They are doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let’s number the days i = 1, 2, . . . , n; for each day i, they have a price p(i) per share for the stock on that day. (We’ll assume for simplicity that the price was fixed during each day.) Suppose during this time period, they wanted to buy 1000 shares on some day and sell all these shares on some (later) day. They want to know: When should they have bought and when should they have sold in order to have made as much money as possible? (If there was no way to make money during the n days, you should report this instead.)

For example, suppose n = 5, p(1) = 9, p(2) = 1, p(3) = 5, p(4) = 4, p(5) = 7. Then you should return “buy on 2, sell on 5” (buying on day 2 and selling on day 5 means they would have made $6 per share, the maximum possible for that period).

Clearly, there is a simple algorithm that takes time O(n2): try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better.

Design an algorithm to solve the problem in O(n log n) time. Your algorithm should use the divide-and conquer technique.

Note: The divide-and-conquer technique can actually solve the problem in O(n) time. But such an algorithm is not required for this assignment. You may think about it if you would like to challenge yourself (but no bonus point this time).

* Algorithm Description

My algorithm uses the mergesort algorithm as a basis but when sorting I keep track of what numbers are swapped as well as what the lowest and highest numbers and days are. This is important because if for example I have the list 5 4 3 2 1 (indexes represent days and entries represent prices). All numbers would be swapped so there would be no best buy and sell days. On the other hand if I have 1 2 3 4 5 this could be thought of as the best case scenario because the prices go nothing but up thus there are no swaps. So my algorithm sorts the list of prices but keeps track of what numbers were swapped then it finds the lowest and highest days and assigns the best buy and sell days accordingly.

* Pseudocode

Given P where P is a list of day objects, a day object as a price, day, and isValid fields.

# [Day,Price]

Global lowest = [null,null]

Global highest = [null,null]

P = mergesort()

print(P)

if lowest or highest has any nulls:

print(‘no solution’)

else:

print(‘Best day to buy ‘ + lowest[0] + ‘, Best day to sell ‘ + highest[0])

print(‘You would have made $’ + (highest[1] - lowest[1]) + ‘ per share’)

mergesort(P) {

if P is empty or P has one element return P, 0, 0

H1 = the first half of P

H2 = the second half of P

H1 = mergesort(H1)

H2 = mergesort(H2)

return merge(H1, H2)

}

merge(A, B, lowest, highest) {

C is a new list

while (A has elements and B has elements) {

if (A.get(0) > B.get(0)) {

A.get(0).isValid = false

C.add(B.remove(0))

}

else C.add(A.remove(0))

}

while (A has elements) C.add(A.remove(0))

while (B has elements) C.add(B.remove(0))

for p in C:

if p.isValid:

if p.price < lowest[1]:

Lowest[0] = p.day

lowest[1] = p.price

if p.price > highest[1]:

highest[0] = p.day

highest[1] = p.price

return C

}

* Correctness

I have tested my algorithm with an implementation of it in java and I have given it many random stock prices per day and it is finding the best buy and sell days. This also makes sense given the structure of my algorithm.

* Time Analysis

My code uses the merge-sort algorithm but adds extra data being tracked. It also added more comparisons in the merge function. However neither of these would change the time complexity. Also there are log(n) calls and there is an n amount of work done at each level of recursion. These facts together I am sure the time complexity is n\*log(n).