Brigham Michaelis
A00364835
September 20th 2017

Homework #2

Problem 1.
Your are given k sorted lists L1, L2, . . . , Lk, with $1 \le k \le n$, 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 L_i , for any $1 \le i \le k$, 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).

The following gives an example.
There are five sorted lists (i.e., k = 5).
Your algorithm needs to sort the elements in all these lists into a single sorted list.
L1 : 3, 12, 19, 45, 34
L2 : 34, 89
L3 : 17, 26, 87
L4 : 28
L5 : 2, 10, 21, 29, 55, 59, 61
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, when k = O(log n), then n log k = O(n log log n), which is strictly smaller than n log n (i.e. it is o(n log n)).

Description
This algorithm works by performing n "compares" log(k) times. So for example say we had 8 lists of already sorted values. These would be our k lists, or 8 lists. And log(k) = 3. So this would work just like merge sort but instead of merging single values we are merging lists. This is done by merging lists 1 and 2 then 3 and4 then 5 and 6 then 7 and 8, for the "leaves of the tree". Then we merge 1&2 and 3&4 then 5&6 and 7&8. Each time we merge lists we are reducing the number of lists by half. In this fashion we are performing "n" compares log(k) times. Finally we merge the last two lists into one list. This will work for any number of lists where the number of lists is sufficiently smaller than n.

Psuedo Code
This is optional and not required.
Correctness
This algorithm is correct because it can be thought of as doing "n" compares "log k times." For example(in the problem description given above) we are first going to merge lists L1 and L2(We can do this in linear time thanks to merge sort). Then we will merge lists L3 and L4. Now we are left with 3 lists. L1&2 and L3&4 and L5. We will now sort lists L1&2 and L3&4 into one list, finally we will sort the last two lists together into one sorted list. We are reducing the number of lists by half at every level we are merging at.
Runtime
The runtime of this algorithm is O(nlog(k)). We are performing "n" compares log(k) times.

Problem 2.
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. ***Another way to think of this is inversions are a measure of how "unsorted" and array is.***

(a) List all inversions of the array ⟨14, 12, 17, 11, 19⟩.

(14,12),(14,11),(12,11),(17,11)

(b) What array with elements from the set {1, 2, . . . , n} has the most inversions?

An array that is sorted in descending order.

How many inversions does it have?

It has (n/2)*(n-1) 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.)

Description
Start with merge-sort we are going to break the entire array down until we hit unity with every piece. Once we start combining the sorted lists we are going to keep track of something called promotions.

Definition of promotions:

Every time two lists are being merged the size of the list on the left will be the current value of "promotions", this is always true. Once a value from the left list is placed in the new array then the promotion value decreases(to a minimum of zero). Finally every time a value from the right list is added to the new-combined-sorted array the inversion variable is incremented with something like: inversions += promotions.

This will be true every time we are merging two lists. In this fashion we can sort the entire array and when we are finished we will have counted the number or inversions that were in the pre-sorted array.
Psuedo Code
Not required
Correctness
This algorithm is correct because we only modified merge sort to keep track of how often a value was promoted(moved from its current position to somewhere nearer to the beginning of the array). Every time this happened we counted how many values it passed by keeping track of the current size of the "left list" in the merge sort.
Runtime
This code runs O(nlog(n)) time because we only modified Merge-Sort to do some extra bit of constant work roughly half the time.

Problem 3.
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, $T(n)$ is constant for $n \leq 2$. (20 points)
(a) $T(n) = 2 \cdot T(n/2) + n^4$.
(b) $T(n) = 4 \cdot T(n/2) + n$.
(c) $T(n) = 2 \cdot T(n/2) + n \log n$.
(d) $T(n) = T((2/3) \cdot n) + n$.

Problem 4
You are consulting for a small computation-intensive investment company, and they have the following type of problem that they want to solve over and over. 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(n^2): 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.

Description
The first thing we are going to do is scan the initial array. During this initial scan we are going to check every element after the first and if every element after the first is less than or equal to the element preceding it then there was no way to make money, and we report this. If this is not the case then we can make money, and we implement our algorithm.

This problem is very similar to the problem where we had to find the number of inversions.(See problem 2C.)

To make money we are going to sort the array in ***<u>descending</u>*** order using merge sort.

For this problem the first thing we checked was whether or not we could make money. We checked this in the first paragraph. So now that we are here we ***<u>know</u>*** that we can make money. In other words this means that there exists an inversion (with respect to descending order) somewhere in the array.

Through the process of merge sort "inversions" are "corrected". Now, the first time a "correction" occurs(ie… a value takes its rightful place in the array) between two lists we are going to take the difference of that value that was just placed in the sorted array, and the last value in the **other** list that is being sorted. This will give us a psuedo "**max**", and this max value is only replaced when another max value is computed that is greater than it.

Now that we have the max value we need to do a little more work.

We need to return the day to buy and the day to sell. So we need to keep track of the indices.
To keep track of the indices we are going to make a struct that has two member variables:

Value
and
Original_index

We start by writing to a new array of type<new struct>(see above) the value and the original index of every element. Then we perform merge-sort on this new array based on the “value” as mentioned above.

Special Note* We actually need one more struct to hold the “difference” of two values or the “max”. When the program is finished running we just look in this struct and return:

buy on original index of the last item and sell on the original index of the first item.

Psuedo Code
Not needed.
Correctness
This algorithm will work for a number of reasons:
1. This algorithm only happens if there is an inversion(with respect to descending order) in the array. This is the first thing that is checked before we run the algorithm.
2. If this algorithm does run then there is an inversion somewhere in the array. Merge-sort will find this inversion and when it does we just keep track of “max-difference” that we encounter.
3. Because we stored the values in an array of type<new struct> (see above) we know what the original indices are, and because we have another struct that stores two indices or (the difference of two indices) we can keep track of best day to buy and the best day to sell.
Runtime
First off we are doing merge-sort in descending order and that automatically gives us O(nlog(n)) runtime. The only difference is we are adding just two or three instructions every time a “correction” happens.
These instructions are:
1. Take the difference of two values.
2. If that difference is bigger than the max
3. Make it the new max

The worst case scenario is this will run in O(nlog(n) + n),(This is if the list is sorted in ascending order) and because we disregard the lower order terms we are left with O(nlog(n)).