CS5050 Advanced Algorithms

Spring Semester, 2021

Assignment 2: Divide and Conquer

Due Date: 11:59 p.m., Thursday, Feb. 18, 2021

Note: The assignment is expected to be much more difficult and time-consuming than Assignment 1, so please start early. As for Question 4 of Assignment 1, for each of the algorithm design problems in all assignments of this class, you are required to clearly describe the main idea of your algorithm. Although it is not required, you are encouraged to give the pseudo-code (unless you feel it is really not necessary). You also need to briefly explain why your algorithm is correct if the correctness is not that obvious. Finally, please analyze the running time of your algorithm.

1. **(20 points)** Suppose there are two **sorted arrays** $A[1...n]$ and $B[1...n]$, each having n elements. 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. You may assume that both arrays A and B are sorted in ascending order.

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

Main Idea: Have two index pointers, one called i that starts at the first element of A[ ] and another called j that begins at the last element of B[ ]. Then add A[i] to B[j] and check if the outcome is equal to x, if it is not then check is the outcome is less than x increase the index i by one, and if not then move the index j down by one.

Pseudo-code:

```
i = 1;

j = n;

  while (i < j) {

    if(A[i] + B[j] == x)      { return 1;}

    else if(A[i] + B[j] < x)    {i = i+1;}

    else {j = j-1;}

  }

return -1
```

Correctness Explanation: This algorithm is correct because it checks the largest number of the second array against the smallest number of the first array. Since the algorithm returns true when A[i] + B[j] = x we either find an answer or return false.

Running Time: The running time of the algorithm is O(n) because in the worst possible case this solution will only loop a maximum of n times. Each time a sum is checked one index moves closer to the middle which will end the loop.

2. **(20 points)** Your are given k sorted lists $L_1, L_2, \ldots, L_k$, 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).

   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.

   L_1 : 3,12,19,25,36

   L_2 : 34,89

   L_3 : 17,26,87

   L_4 : 28

   L_5 : 2,10,21,29,55,59,61

Main Idea: Divide the problem by finding the mid-point between I = 1 (this is the first list) and k (this is the last list) and then split the lists there. Conquer by do this recursively until I and k are equal, which will separate all the lists individually. Combine by going back up and sorting the lists next to each other into one combined list.

Pseudo Code:

```
mergeklists(lists){

If (lists == null){ return null; }

Return mergeklists(lists, 1, lists length);



mergeklists(lists, int start, int end){

        if (start == end){ return lists[start];} (base case)

        int mid = start + end / 2;

        left = mergeklists(lists, start, mid);

        right = mergeklists(lists, mid + 1, end);

        return merge (left, right);

}

Merge(list1, list2){
```

Merge together the lists recursively}

Running time: The running time of this algorithm will be O(n log k) because we are merging k lists recursively and the n comes from checking the n values of each list.

3. **(30 points)** Let $A[1\cdots n]$ 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.

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

(4,2);(4,1);(2,1);(9,1);(9,7)

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

The array with the most elements from the set {1,2,..,n} is {n,n-1,…,1} and it has (n(n-1)/2) 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)**

Main Idea: Modify merge sort so that it checks for inversions. This algorithm goes sorts through an array and in the merge method uses the if else chain to check if a > n1 and b > n2, which are the index variables for the arrays, and if the left array value at variable a is greater less than or equal to the right array at variable b.

Pseudo Code: Initially: MERGE-SORT(A, 1, n);

```
Merge-sort(A,i,k)
 if (i < k){
    inversions = 0
    mid = (i + k) / 2;
    inversions += merge_sort(A, i, mid);
    inversions += merge_sort(A, mid+ 1, k);
    inversions += merge(A, i, mid, k);
    return inversions;}
 else{ return 0;}
MERGE(A, i, mid, k){
 n1 = mid - i + 1;
```

```
n2 = k – mid;
let L[1..n1] and R[1..n2] be new arrays
for (a = 1 to n1)
   L[i] = A[i + a - 1];
for (b = 1 to n2)
   R[j] = A[mid + b];
a = 1;
b = 1;
for (r = i to k)
   if (a > n1)
      A[r] = R[b];
      b = b + 1;
   else if (b > n2)
      A[r] = L[a]
      a = a + 1
   else if (L[a] ≤ R[b])
      A[r] = L[a]
      a = a + 1
   else
      A[r] = R[b]
      b = b + 1
      inversions += n1 - a
return inversions;}
```

Correctness Explanation: The algorithm uses the if else statements in merge to check the elements of the arrays against the inversion definition of $i < j$ and $A[i] > A[j]$, then the pair $(A[i],A[j])$ is called an *inversion* of A. This is also a divide and conquer algorithm because the algorithm is

being initially divided into smaller subarrays then the conquer logic occurs and finally is combined through merge.

Running Time: The running time for this algorithm is O(n log n) because it is a modified merge sort.

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 $n \leq 1$.

   (a) $T(n) = 2 \cdot T(\frac{n}{2}) + n^3$.

   (b) $T(n) = 4 \cdot T(\frac{n}{2}) + n\sqrt{n}$.

   (c) $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$

   (d) $T(n) = T(\frac{3n}{4}) + n$.

   a.) O(n^3)

   b.) O(n^2)

   c.) O(nlog(n))

   d.) O(n^$log_3 4$)

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 $O(n^2)$: 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 (but no bonus point this time).

Main Idea: Use the Divide and Conquer approach to find the maximum subarray in order to find the best possible day to buy and then sell. First we divide the subarray into two equal size subarrays, A[low..mid] and A[mid+1..high]. Then, conquer by finding max of subarrays A[low..mid] and

A[mid+1..high]. Finally, we combine by finding best solution from the two solutions found in conquer step and check them against the solution of subarray crossing the midpoint.

Pseudo Code: Initially, call Max-subarray (A, 1, n)

```
Max-subarray (A, I, k){

if I = k  return A[i];}(Base Case)

Else{ mid = I + k / 2;

Max1 = Max-subarray(A, I, mid);

Max2 = Max-subarray(A, mid + 1, k):

Max-cross = Max-Cross-Subarray(A, I, mid, k);

Check which max is largest and then return that index.}

Max-Cross-Subarray (A, I, mid, k) {

Left-sum = 0;

Sum = 0;

For I = from mid to low;

        sum = sum + A[i];

        if (sum > left-sum)

                left-sum = sum;

                 max-left = I;

right-sum = 0;

        sum = 0;

        for( j = mid+1 to high)

                sum = sum + A[j];
```

```
        if (sum > right-sum)

                right-sum = sum;

                max-right = j;

return(max-left, max-right, left-sum + right-sum);
```

Correctness Explanation: This algorithm should be correct because it divides up the problem into two smaller pieces, then conquers by checking for the maximum value subarray in each side, and finally combines by checking the two maximums against each other and then against the maximum subarray found in the middle cross array. This should check the array for all the best possible solutions.

Running Time: The time complexity for the algorithm is $T(n)= 2 * T(n/2) +n$, so the worst case run time can be solved to be $O(n \log n)$.

Total Points: 110