Tareef Dedhar
20621325

Assignment 1: Question 4

Acknowledgments. I have completed this question without outside sources.

Common sum [10 marks]

In the COMMON SUM problem, we are given two arrays A and B of length n containing non-negative (not necessarily distinct) integers, and we must determine whether there are indices $i_1, i_2, j_1, j_2 \in \{1, 2, \ldots, n\}$ for which

$$A[i_1] + A[i_2] = B[j_1] + B[j_2].$$

Design an algorithm that solves the COMMON SUM problem and has time complexity $O(n^2 \log n)$ in the setting where operations on individual integers take constant time.

Your solution must include a description of the algorithm in words, the pseudocode for the algorithm, a proof of its correctness, and an analysis of its time complexity in big-Θ notation.

Solution.

1 Description

First, we take our two arrays of length n (a and b), and new empty array of length ((n)(n+1)/2) named c. We will fill c with all the possible sums in b (which number ((n)(n+1)/2). Then, we will sort c using an efficient sort such a mergesort. Then, for each sum in a, we will check if that sum is located in c, using a binary search. If we get a match, return true, otherwise return false.

2 Pseudocode

```
int[n] a,b;
int[n(n+1)/2)] c;

for (i from 0 to n)
    for (j from i to n)
    c.insert([b[i]+b[j])

sort(c)

for (i from 0 to n)
    for (j from i to n)
    if (binary_search((a[i]+a[j]), c)
    return true

return false
```

3 Proof

First, we will show the number of possible sums in an array numbers (n)(n+1)/2. Intuitively, if you want all the sums possible using b[0], you will add b[0] with b[i], where i = 0 to n. However, if you do this with every index, you will get repeats. For example, if we look at b[1], we don't need to add b[1] with b[0] since we already summed these in the previous step. The same is true for b[2] with b[0] and b[1], so on and so forth. We can express this as a sum $\sum_{i=0}^{n} \sum_{j=i}^{n} b[i] + b[j]$, since each time we start finding sums using b[i], we already added b[i] with b[0] through [i]. So, we can also justify how the array c is populated using this logic.

We sort c to allow for binary search to function, as this allows us to be more efficient.

As for the last loop, we use the same looping logic as above to generate all the sums in array a as we did with array b. However, instead of generating an array containing all these sums, for each of them, we will search our existing array of sums in b for the value of a[i] + a[j]. If it exists, we have a match, and there is a common sum. Otherwise, we return false if all possibilities have been exhausted.

4 Time Complexity

This is $\Theta(n^2 \log n)$, because:

1. Looping from i=0 to n is obviously done n times, and this is executing another loop that is up to n recurrences, where each internal operation is a constant time insertion. This is executing something of constant time up to n times, n times over. So our first step is $\Theta(n^2)$.

2. Sorting something of length ((n)(n+1)/2), which is of the order n^2, using an efficient sort such as mergesort is known to be of time complexity $\Theta(n^2 \log n^2)$, which reduces to $\Theta(n^2 \log n)$.

3. This is the same loop as in step 1, except for the innermost step; instead of a constant time insertion, we are performing a binary search on an array of length n^2. So, each internal step is $\Theta(\log n^2)$, which reduces to $\Theta(\log n)$. So, our total loop time efficiency is $\Theta(n^2 \log n)$.

4. Since our longest step is $\Theta(n^2 \log n)$, and the three steps are independent of each other, we can conclude our total efficiency is $\Theta(n^2 \log n)$.