Michelle Lewis, Dylan Raymond, Tej Singh
February 1, 2024
CS 325 - Amir Nayyeri

Group Assignment 1

Algorithm A: number_of_pairs

Description:

number_of_pairs takes two arrays (A, B) and a min/max range and outputs the number of pairs from both arrays whose sum are within the min/max range. It utilizes `mergeSort` to recursively sort the arrays. It then iterates through the elements of A and adds each to "some" elements of B, to determine the number of sums that fall within the range. The sum comparisons are conducted recursively through a divide and conquer mechanism that compares the B elements to the minimum and maximum values of the range (`left_sum_index` and `right_sum_index` respectively). In each function, it splits the array in half to examine smaller and smaller pieces near the min or max to determine the correct index. Once B's min index and max index are known, the function counts the number of values for that element of A and progresses to the next element until complete.

Run-time analysis:

```
Number_of_pairs(A[1…m], B[1…n], min_max_range)
```
$O((n+m)\log(n+m))$

```
    mergeSort(A[1…m])
```
$O(m \log m)$

```
    mergeSort(B[1…n])
```
$O(n \log n)$

```
    for element in A[1…m]):
        left_sum_index(B[1…n])
        right_sum_index(B[1…n])
```
$2m\ O(\log n) = O(m \log n)$

```
left/right_sum_index(B[1…n])                O(log n) - each

      m = middle index of array

      if                                    O(1)

      elif                                  O(1)

      elif:

            left/right_sum_index(B[m+1…n])  T(n/2)

      else:

            left/right_sum_index(B[1…m])    T(n/2)
```

Proof of correctness:

The algorithm starts with 2 merge sorts, one of array A and one of array B. These calls take time $O(m \log m)$ and $O(n \log n)$ respectively. Additionally, a for loop is used to loop through each element of A, containing a call to `left_sum_index` and `right_sum_index`. Both `sum_index` functions search recursively through array B where each recursive call halves the range of indices to search for. This gives the recursion $T(n) = T\left(\frac{n}{2}\right) + O(1)$ where $T(n)$ is the running time of `sum_index` on an array of n. This recursion runs a total of $\log n$ times, each with additional time complexity of $O(1)$. Thus, the time complexity of both `sum_index` functions are $O(\log n)$. Since these are run twice for each value of A, the time complexity of the for loop is $2mO(\log n) = O(m \log n)$. The resulting time complexity of `number_of_pairs()` comes out to

$$O(n \log n) + O(m \log m) + O(m \log n)$$

Let $k = \max\{n, m\}$, then since the logarithmic function is an increasing function, we have

$$n \log n + m \log m + m \log n \leq k \log k + k \log k + k \log k = 3k \log k$$

Additionally, since $k \leq n + m$ given that $n, m > 0$,

$$3k \log k \leq 3(n + m) \log(n + m) = O\big((n + m) \log(n + m)\big)$$

Therefore we have proven that `number_of_pairs()` is of time complexity $O\big((n + m) \log(n + m)\big)$ as desired.

Algorithm B: count_valid_contiguous_subarrays

Description:

number_of_allowable_intervals uses *count_valid_contiguous_subarrays*, which takes a list, the list size, a minimum, and a maximum and outputs the number of non-empty contiguous subarrays of the list that add up to a value that is greater than or equal to the minimum and less than or equal to the maximum. It utilizes divide and conquer techniques to recursively split the list into lists consisting of only one value. After each split, it creates two lists. One list holds the progressive sums of the left half and the other holds the progressive sums of the right half. It then passes these lists as arguments of the `number_of_pairs` algorithm (the algorithm for part A) to get the number of valid contiguous subarrays that overlap the left and right half. When the program recursively reaches a list consisting of one value, that value is checked to make sure it resides within the valid range. If so, it adds one to the current number of valid contiguous subarrays found. The total amount of valid contiguous subarrays is then written to the output file.

Run-time analysis:

```
count_valid_contiguous_subarrays(list[...], min, max)        → O(n · log² n + n)
      valid_contiguous_subarrays = 0
      if size > 1:                                            → O(1)
            left_sublist = left half of list[...]
            right_sublist = right half of list[...]
            left_sum_list = [empty]
            right_sum_list = [empty]

            for element in left_sublist:                      → O(n/2)
            left_sum_list.append(left_sublist progressive sums)
            for element in right_sublist:                     → O(n/2)
                  right_sum_list.append(right_sublist progressive sums)
```

```
valid_contiguous_subarrays +=
count_valid_contiguous_subarrays(left_sublist, left_size, min,
max)
```
$\rightarrow T\left(\frac{n}{2}\right)$

```
valid_contiguous_subarrays +=
count_valid_contiguous_subarrays(right_sublist, right_size, min,
max)
```
$\rightarrow T\left(\frac{n}{2}\right)$

```
valid_contiguous_subarrays +=
number_of_pairs(left_sum_list, right_sum_list, [min, max])
```
$\rightarrow O(n \log n)$

Proof of correctness:

We can prove that this algorithm provides the desired $O(n \cdot poly \log(n))$ time complexity by first writing a recurrence relation for the count_valid_contiguous_subarrays function:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n + n \log n; \qquad T(1) = 1$$

To prove that count_valid_contiguous_subarrays has the desired $O(n \cdot poly \log(n))$ time complexity, let's assume that:

$$T(n) \leq n \cdot \log^2 n$$

This will be our induction hypothesis. For the base case $T(1) = 1$, the recurrence sets

$$T(1) = 1 \leq 1 \cdot \log^2 1 = 0.$$

$$T(n) = T(n) + n + n \cdot \log n$$

$$\leq n \cdot \log^2 n + n + n \cdot \log n$$

$$\leq n + (\log^2 n + \log n + 1)$$

$$\leq n \log^2 n$$

When solving this equation, we can simplify the equation by dropping constants and any non-dominant terms. Since $\log^2 n$ is the fastest-growing term in the parentheses as the input gets larger, we can eliminate all other terms in the parentheses. This leaves us with

$n \cdot \log^2 n$. Thus, count_valid_contiguous_subarrays has a runtime complexity of $O(n \cdot \log^2 n)$.