CMPSC 465 Data Structures & Algorithms
Fall 2024 Mingfu Shao and Debarati Das

HW 2

Due September 13th (Friday), 11:59 pm

Formatting: Each problem should begin on a new page. When submitting in Gradescope, try to assign pages to problems from the rubric as much as you can. Make sure you write all your group members' names. For the full policy on assignments, consult the syllabus.

1. **(0 pts.) Acknowledgements.** List any resources besides the course material that you consulted in order to solve the assignment problems. If you did not consult anything, write "I did not consult any non-class materials." The assignment will receive a 0 if this question is not answered.

2. **(20 pts.) Inversions.** Consider a company that maintains a large dataset of financial records, where each record is represented by an integer value. The company wants to identify pairs of records that meet specific comparative criteria for further analysis.

   You are given an array F of n integers, where each integer represents the value of a financial record. Your task is to efficiently identify and count the number of pairs of indices (i, j) in the array such that:

   - $1 \leq i < j \leq n$
   - $F[i] > 2 \times F[j]$

   - **(5 pts.) Simple Approach:** Describe a basic method to solve the problem by examining all possible pairs in the array.
   - **(10 pts.) Optimized Solution:** Develop an optimized algorithm that improves the efficiency of the initial approach. Explain your approach and provide pseudocode.
   - **(2 + 3 pts.) Time Complexity Analysis:** Evaluate the time complexity of both the simple and optimized approaches. Provide a detailed explanation of how you derive the complexity for each.

   Solution.

   a. **Brute Force Solution** A simple approach involves examining all pairs (i, j) and counting those that satisfy $F[i] > 2 \times F[j]$.

Algorithm 1 Brute Force Solution

```
Input: Array F of size n
Output: Count of pairs (i, j) such that 1 ≤ i < j ≤ n and F[i] > 2 × F[j]
count ← 0
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if F[i] > 2 × F[j] then
            count ← count + 1
        end if
    end for
end for
return count
```

b. **Optimized Approach: Merge Sort.** The optimized approach utilizes a modified merge sort to count the valid pairs during the merging process. We can modify the Algorithm #3 introduced in class to solve it. See the pseudo-code given below. We need to call DC-sort-count-inv2 (F) to get the correct number of inversions defined in this problem.

```
Algorithm: DC-sort-count-inv2 (A[1 ··· n])
    if n ≤ 1: return (A, 0);
    (A'_1, I_1) = DC-sort-count-inv2 (A[1 ··· ⌊n/2⌋]);
    (A'_2, I_2) = DC-sort-count-inv2 (A[⌊n/2⌋ + 1 ··· n]);
    (A', I_3) = merge-count-inv2-two-sorted-arrays (A'_1, A'_2);
    return (A', I_1 + I_2 + I_3);
end algorithm;

function merge-count-inv2-two-sorted-arrays (S[1 ··· m], T[1 ··· n])
    // this part counts the number of inversions
    init inv = 0, pointer j = 1;
    for i = 1 → m
        while j ≤ n and S[i] > 2 · T[j]
            j = j + 1;
        end while;
        inv = inv + (j − 1);
    end for;
    // this part merges the two sorted arrays
    init empty array C, reset pointers i = 1, j = 1;
    S[m + 1] = M and T[n + 1] = M;
    for k = 1 → m + n
        if S[i] ≤ T[j]
            C[k] = S[i];
            i = i + 1;
        else
            C[k] = T[j];
            j = j + 1;
        end if;
    end for;
    return (C, inv);
end function;
```

In above merge-count-inv2-two-sorted-arrays, we separate the count of inversions and the construction of the merged sorted array. In the first part, for each $S[i]$, we tranverse T and find the *first* j such that $S[i] \leq 2 \cdot T[j]$. This means that for each $j' = 1 \to j-1$ we have $S[i] > 2 \cdot T[j']$, i.e., there are $j-1$ inversions formed with $S[i]$; therefore we increase "inv" by $j-1$. Note that, for $i+1$, since $S[i+1] \geq S[i]$, *first* j that satisfies $S[i+1] \leq 2 \cdot T[j]$ must be either the same as or larger than the previous j, as T is sorted as well. In other words, j never needs to go back. This is critical to ensure the correctness and that the merge-count-inv2-two-sorted-arrays function runs in $\Theta(m+n)$ time.

This previously released merge-count-inv2-two-sorted-arrays is incorrect. Can you design an instance S and T such that it will give incorrect count of inversions? Doing so will help you understand why we need to separate counting and merging. Such an instance (that fails this algorithm) can be $S = (3, 7), T = (2, 5)$.

function merge-count-inv2-two-sorted-arrays ($S[1 \cdots m], T[1 \cdots n]$)
    init an emity array C;
    init pointers $i = 1$ and $j = 1$;
    init inv = 0;
    $S[m + 1] = M$ and $T[n + 1] = M$;
    for $k = 1 \to m + n$
        if $S[i] \leq T[j]$
            $C[k] = S[i]$;
            $i = i + 1$;
        else if $S[i] > 2 \cdot T[j]$
            $C[k] = T[j]$;
            $j = j + 1$;
            inv = inv + $(m - i + 1)$;
        else (*# that is,* $T[j] < S[i] \leq 2 \cdot T[j]$)
            $C[k] = T[j]$;
            $j = j + 1$;
        end if;
    end for;
    return (C, inv);
end algorithm;

c. **Time Complexity Analysis**

1. Brute Force Approach: In the brute force approach, we examine all pairs (i, j) where $i < j$, checking whether the condition $F[i] > 2 \times F[j]$ holds.

Analysis:

- The outer loop runs from $i = 1$ to $n - 1$, which takes $\Theta(n)$ iterations.
- The inner loop runs from $j = i + 1$ to n, meaning the number of iterations in the inner loop decreases as i increases. For each i, the inner loop performs $n - i$ comparisons.
- Thus, the total number of comparisons is:

$$\sum_{i=1}^{n-1}(n - i) = \frac{n(n - 1)}{2}$$

- This simplifies to $\Theta(n^2)$.

2. Optimized Merge Sort Approach: The optimized approach uses a modified merge sort to count valid pairs during the merging process.

Analysis:

- The counting part of the merge-count-inv2-two-sorted-arrays function takes $\Theta(m+n)$ time as both i and j will never decrease (although it is two nested loops); its merging part takes $\Theta(m+n)$ time too as we already analyzed in class. Hence, the merge-count-inv2-two-sorted-arrays function takes $\Theta(m+n)$ time.
- The running time of DC-sort-count-inv2 function can be written as a recurrence. Let $T(n)$ be its time complexity of DC-sort-count-inv2 (A) when A contains n integers. Therefore, we can write $T(n) = 2 \cdot T(n/2) + \Theta(n)$. By using the master's theorem, we can conclude $T(n) = \Theta(n \cdot \log n)$.

3. **(20 pts.) Median of Medians.** In the median of median algorithm, Anna and John decide to come up with their own variants. Anna decides to partition the original array into $\lceil n/7 \rceil$ groups of 7 elements each; whereas John decides to partition the original array into $\lceil n/3 \rceil$ groups of 3 elements each. The rest of the algorithm remains the same for both. You may assume that all entries in the original array are distinct.

   - **(7 pts.)** Analyze the bounds for $\max(|A_1|, |A_2|)$ and write the recurrence relation for the time complexity of Anna's algorithm in the following form:

   $$T(n) \leq \begin{cases} ?? & , n > n_0^A \\ \Theta(1) & , n \leq n_0^A \end{cases}$$

   Fill in the ?? in the above part. (You do not need to worry about a value for n_0^A.)
   - **(7 pts.)** Analyze the bounds for $\max(|A_1|, |A_2|)$ and write the recurrence relation for the time complexity of John's algorithm in the following form:

   $$T(n) \leq \begin{cases} ?? & , n > n_0^J \\ \Theta(1) & , n \leq n_0^J \end{cases}$$

   Fill in the ?? in the above part. (You do not need to worry about a value for n_0^J.)
   - **(3 pts.)** Show that for Anna's algorithm, $T(n) = \Theta(n)$.
   - **(3 pts.)** Show that for John's algorithm, $T(n) = O(n \log n)$.

   Conclusion from question: In case you did not see it already, the above question implies that median of medians algorithm runs in linear time for partition-element sizes 5 and 7 (in fact for any odd number ≥ 5), but we can only prove superlinear times for size 3. We only consider odd partiton-elements as they have a unique median.

   Solution. For the simplicity of analysis, we can assume that n is a multiply of 21 (i.e., a multiply of both 3 and 7).

   - Anna's algorithm: We first try to lower bound the number of elements greater than the median of medians i.e. $|A_2|$. Now Anna's algorithm will divide n elements into $n/7$ groups, out of which about half of these groups, i.e., $n/14$, would have their median greater than the pivot (i.e., the median of medians). In each of such group, there are exactly 4 elements (its median and another 3 larger than the median) that are guaranteed greater than the pivot. Hence, we can give a lower bound as: $|A_2| \geq 4 \cdot n/14 = 2n/7$. Consequently, we can have an upper bound for the number of elements less than the median of medians: $|A_1| = n - |A_2| \leq 5n/7$. By a symmetrical similar argument, we can find the exact same upper bound on the number of elements greater than the median of medians i.e. an upper bound on $|A_2|$, that is $|A_2| \leq 5n/7$. Combined, we have $\max(|A_1|, |A_2|) \leq 5n/7$.

So $T(n)$ is upper bounded by the time it takes to recurse on the larger of $|A|_1$ and $|A_2|$ added to the time it takes to find the median of the $n/7$ medians. Hence, we have:

$$T(n) \leq \begin{cases} T(n/7) + T(5n/7) + \Theta(n) & , n > n_0^A \\ \Theta(1) & , n \leq n_0^A \end{cases}$$

- John's algorithm: We can analyze it in a similar fashion as above. We first try to lower bound the number of elements greater than the median of medians i.e. $|A_2|$. Now John's algorithm will divide n elements into $n/3$ groups, out of which about half of these groups, i.e., $n/6$, would have their median greater than the pivot (i.e., the median of medians). In each of such group, there are exactly 2 elements (its median and another one larger than the median) that are guaranteed greater than the pivot. Hence, we can give a lower bound as: $|A_2| \geq 2 \cdot n/6 = n/3$. Consequently, we can have an upper bound for the number of elements less than the median of medians: $|A_1| = n - |A_2| \leq 2n/3$. By a symmetrical similar argument, we can find the exact same upper bound on the number of elements greater than the median of medians i.e. an upper bound on $|A_2|$, that is $|A_2| \leq 2n/3$. Combined, we have $\max(|A_1|, |A_2|) \leq 2n/3$.
  So $T(n)$ is upper bounded by the time it takes to recurse on the larger of $|A|_1$ and $|A_2|$ added to the time it takes to find the median of the $n/3$ medians. Hence, we have:

$$T(n) \leq \begin{cases} T(n/3) + T(2n/3) + \Theta(n) & , n > n_0^J \\ \Theta(1) & , n \leq n_0^J \end{cases}$$

- Recall the theorem is for the following recurrence relation:

$$T(n) = \begin{cases} T(an) + T(bn) + \Theta(n) & , n > n_0 \\ \Theta(1) & , n \leq n_0 \end{cases}$$

  If $a + b < 1$, then $T(n) = \Theta(n)$; but if $a + b = 1$, then $T(n) = \Theta(n \log n)$. Now for Anna's time complexity, let $U(n)$ be a function satisfying the following recurrence equation:

$$U(n) = \begin{cases} U(n/7) + U(5n/7) + \Theta(n) & , n > n_0^A \\ \Theta(1) & , n \leq n_0^A \end{cases}$$

  Then by using the theorem, $a + b = 1/7 + 5/7 = 6/7 < 1$, we have $U(n) = \Theta(n)$. If $T(n)$ is Anna's time complexity, you can trivially show using induction on n that $T(n) \leq U(n)$ for all n. This implies that $T(n) = O(n)$. Now to show $T(n) = \Omega(n)$, simply observe that the algorithm partitions the array into A_1 and A_2 which takes $\Theta(n)$ time; also in the find-pivot function calculating the median of each subarrays takes $\Theta(n)$ time as well. Thus $T(n) = \Omega(n)$. Combined, we know $T(n) = \Theta(n)$.
- For John's algorithm, similarly, let $U(n)$ be a function satisfying the following recurrence:

$$U(n) = \begin{cases} U(n/3) + U(2n/3) + \Theta(n) & , n > n_0^A \\ \Theta(1) & , n \leq n_0^A \end{cases}$$

  Since $a + b = 1/3 + 2/3 = 1$, by using above the theorem, $U(n) = \Theta(n \log n)$. If $T(n)$ is John's time complexity, you can trivially show using induction on n that $T(n) \leq U(n)$ for all n. This implies that $T(n) = O(n \log n)$. This is the end of the story, as we cannot show that $T(n) = \Omega(n \log n)$; none of the steps in the selection algorithm takes $\Theta(n \log n)$ time.