Weekly Assignment 4: Solution
Total: 100

CS 2500: Algorithms

Due Date: October 29, 2024 at 11.59 PM

Weighted Median

1. Let $x_1, x_2, \ldots, x_n$ be n elements, each assigned an equal weight $w_i = \frac{1}{n}$ for all $i = 1, 2, \ldots, n$. The total sum of the weights is

$$\sum_{i=1}^{n} w_i = \sum_{i=1}^{n} \frac{1}{n} = 1.$$

To find the weighted median, we need an element x_k such that

$$\sum_{x_i < x_k} w_i < \frac{1}{2} \quad \text{and} \quad \sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

Since each weight $w_i = \frac{1}{n}$, each subset of k elements has a total weight $k \cdot \frac{1}{n} = \frac{k}{n}$.

(a) Case of Odd n: For an odd number of elements $n = 2m + 1$, the median x_k is the middle element, where $k = m + 1$.

- The weight of elements less than x_k is $\frac{m}{n} < \frac{1}{2}$.
- The weight of elements greater than x_k is also $\frac{m}{n} \leq \frac{1}{2}$.

Thus, x_k satisfies the conditions for the weighted median when all weights are equal.

(b) Case of Even n: For an even number of elements $n = 2m$, there is no single middle element. Instead, there are two middle elements, located at positions m and $m + 1$ in the sorted list.

To satisfy the weighted (lower) median conditions, we choose the lower of the two middle elements, x_m, as the weighted median. This ensures:

- The weight of elements strictly less than x_m is

$$\frac{m-1}{n} = \frac{m-1}{2m} < \frac{1}{2}.$$

- The weight of elements strictly greater than x_m is

$$\frac{m}{n} = \frac{1}{2}.$$

This choice of x_m meets both conditions required for the weighted (lower) median.

Conclusion: The median of $x_1, x_2, \ldots, x_n$ is the same as the weighted (lower) median when each element has an equal weight $w_i = \frac{1}{n}$, with the lower of the two middle elements chosen if n is even.

Finding the i Largest Elements in a List

1. **Method 1: Sort the Numbers and List the i Largest**

   (a) Steps involved in this method:

   - Sort the array: Start by sorting the entire list of n numbers in descending order (or ascending order and then take the last i elements).
   - Select the i largest elements: After sorting, the i largest elements will be the first i elements in the sorted list if sorted in descending order, or the last i elements if sorted in ascending order.
   - Output the sorted i largest elements: Simply extract and output these i elements in the order they appear.

   (b) The worst-case time complexity of this approach is $O(n \log n)$.

   - Sorting a list of n numbers using a comparison-based sorting algorithm (such as Merge Sort or Heap Sort) takes $O(n \log n)$ time.
   - After sorting, selecting the top i elements is an $O(i)$ operation, but this is dominated by the $O(n \log n)$ sorting time.

   Therefore, the overall worst-case time complexity for this method is $O(n \log n)$.

2. **Method 2: Use an Order-Statistic Algorithm**

   (a) An order-statistic algorithm, such as Quickselect, can be used to find the i-th largest element in the list. The steps are as follows:

   - Find the i-th largest element: Use Quickselect (a selection algorithm) to find the i-th largest element in the list. Quickselect has an average time complexity of $O(n)$, but in the worst case, it can be $O(n^2)$. However, with median-of-medians or similar optimizations, it can be made worst-case $O(n)$.
   - Partition the list: Once the i-th largest element is found, partition the list into elements that are greater than or equal to this i-th largest element and those that are less.

   (b) Steps involved in this method after identifying the ith largest element:

   - Select elements greater than or equal to the i-th largest element: After finding the i-th largest element, we have a partitioned list where one part contains all elements greater than or equal to this element.
   - Sort the subset of i largest elements: Sort the subset of i largest elements to ensure they are in descending order (or ascending order, depending on preference).
   - Output the sorted i largest elements: Output these sorted i largest elements.

   (c) Overall worst-case time complexity:

   - Finding the i-th largest element: Using Quickselect, the worst-case time complexity to find the i-th largest element is $O(n^2)$ due to poor pivot choices in the worst case.
   - Partitioning and sorting the i largest elements: Once we have identified the i-th largest element, partitioning the list around this element is an $O(n)$ operation. Sorting the i largest elements then takes $O(i \log i)$ time.

   Therefore, the overall worst-case time complexity of Method 2 is:

   $$O(n^2) + O(i \log i) = O(n^2)$$

   - **Comparison with Method 1:** In the worst case, Method 2 does not offer better performance than Method 1, as Method 1 has a worst-case time complexity of $O(n \log n)$, which is better than $O(n^2)$.
   - **Average-case comparison:** In practice, Quickselect performs closer to $O(n)$ on average. Thus, if we consider the average case, Method 2 could be more efficient than Method 1, especially for small values of i, where $i \log i$ remains small.

3. **Comparison and Analysis**

   (a) Two methods:

- Method 1 (Sorting): This has a time complexity of $O(n \log n)$, regardless of i.
- Method 2 (Order-Statistic): This has a worst-case time complexity of $O(n^2)$ due to Quickselect's possible poor pivot choices.

Comparison:

- In the worst case, Method 1 is generally more efficient than Method 2 because $O(n \log n)$ is better than $O(n^2)$.
- However, in the average case, Quickselect is expected to perform closer to $O(n)$. When i is small relative to n, $O(i \log i)$ is also small, making Method 2's average case $O(n+i \log i)$ potentially faster than Method 1's $O(n \log n)$. For example, if $i = O(\log n)$, Method 2's average-case complexity $O(n + i \log i) \approx O(n)$, which can be better than $O(n \log n)$.
- As i grows closer to n, the $i \log i$ term approaches $n \log n$, reducing Method 2's advantage. Therefore, Method 2 is more efficient in the average case when i is relatively small compared to n.

(b) **Scenario:** Method 1 would be preferable when we need all or nearly all elements in sorted order, or when we don't know the exact value of i in advance.

Reasoning: If i is very close to n (say, $i = n-1$), then Method 1's time complexity $O(n \log n)$ becomes comparable to Method 2's time complexity in the average case, $O(n+(n-1)\log(n-1)) = O(n \log n)$.

Additionally, if a fully sorted list is needed for other operations, Method 1 avoids the need for multiple steps to first find the i-th largest element and then sort a subset. Method 1 thus provides a fully sorted list at the outset, which can be advantageous in cases where we might need more than just the top i elements.