CSCI 4470 Algorithms

Part II Sorting and Order Statistics

Chapter 9: Medians and Order Statistics

Order Statistics

Definition

- i**-th Order Statistic**: Element with rank i in set S.
- **Minima**: First order statistic, requires $n-1$ comparisons.
  - **Min value is 1**
- **Maxima**: n-th order statistic.
  - **Max value is n**
- The **Median** is another important order statistic, representing the middle value of a dataset.
  - **Median** is $\lceil \frac{n}{2} \rceil$

Selection Problem

- Given a set A of n distinct numbers and a number i, the task is to find the i-th order statistic of A.
- Used to find the i-th smallest number in an unsorted list in linear time.

Average Case & Worst Case:

- **Average Case**: Typically $O(n)$.
- **Worst Case**: Can be $O(n)$ with optimal algorithms like Median of Medians.
- Could be worse than $O(n)$, such as $O(n^2)$ or $O(n \log n)$, depending on the algorithm used.

Finding Minima and Maxima Together

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
     if min > A[i]
         min = A[i]
return min
```

- **Number of Comparisons to Find Minima**: $n-1$.
- **Number of Comparisons to Find Maxima**: $n-1$.
- **Lower Bound on Number of Comparisons**: $\Omega(n)$.

Method:, To find both minima and maxima together, **compare in pairs**:

- Compare $x_1 : x_2$.
- If $x_1 < x_2$, then compare x_1 with current min and x_2 with current max.
- Otherwise, compare x_2 with current min and x_1 with current max.

```
MIN-AND-MAX(A)
min = max = A[1]
for i = 1 to A.length/2      // assume A.length is even
if A[2i - 1] > A[2i]
   if min > A[2i]    //compare min to smaller value
     min = A[2i]
     if max < A[2i-1]    // compare max to larger value
        max = A[2i-1]
else if min > A[2i-1]     // compare min to smaller value
  min = A[2i-1]
  if max < A[2i]         //compare max to larger value
     max = A[2i]
return min, max
```

- **Number of Comparisons in One Iteration**: 3.
  - In the `MIN-AND-MAX` algorithm, it's not always 3 comparisons in one iteration. It can be 2 or 3 comparisons depending on the elements.
- **Total Number of Iterations**: $\frac{n}{2}$(because we are comparing in pairs: $[x_1 : x_2]$, $[x_3 : x_4]$, ..., etc.)
  - The total number of comparisons is not strictly $3(\frac{n}{2})$. It is at most $3(\frac{n}{2})$ and at least $2(\frac{n}{2})$.
- **Total Number of Comparisons**: $3(\frac{n}{2})$.
- **Optimized Number of Comparisons**: Less than $2(n-1)$ when found separately.

> $O(3\frac{n}{2})$ comparisons

Randomized Select Algorithm

- **Randomized Select Algorithm**: To find any **i-th** order statistics efficiently.
- How can selection be performed in place in expected linear time?
  - Similar to binary search
  - **Approach**: Similar to quicksort, it works by partitioning the array and then recursively searching in one of the partitions.
- **Order statistics**: The elements of a set sorted in order. The i**-th** order statistic is the i**-th** smallest element in the set.

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]    // 1 s is r - p + 1 when p == r means that i=
q = RANDOMIZED-PARTITION(A, p, r) // Partition function from Quicksort Alg.
k = q - p + 1
if i == k
    return A[q]     // found ith value which is the pivot value is the answer
else if i < k
    return RANDOMIZED-SELECT(A, p, q-1, i)      // find smaller value
else
    return RANDOMIZED-SELECT(A, q+1, r, i-k)    // find larger value
    // RANDOMIZED-SELECT which are two recursion calls
    // (i-k) accounts for k values removed from search sub-array
```

Selecting i-th Element

- **Randomized Partition**: The core operation that helps in finding the i-th order statistic.
  - If it returns index i, $A[i]$ is the i-th order statistic.
  - If it returns index $k < i$, the search continues on the right side.
  - If it returns index $k > i$, the search continues on the left side.
- **Function Parameters**:
  - Array A, starting index p, ending index r, and the target order statistic index i.
- **Working**:
  - **Single Element**: If p equals r, return $A[p]$.
  - **Partitioning**: Utilizes randomized partition to find the pivot and accordingly directs the search to the left or right subarray.
  - **Recursive Search**: If the pivot's rank matches i, it is returned; otherwise, a suitable subarray is chosen for a recursive search based on the comparison of i and the pivot's rank.

Analysis of Randomized Select

Expected running time: $O(n)$, where n is the number of elements in the array.
Worst-case running time: $O(n^2)$, but the worst case occurs rarely due to randomization.

- What is the worst-case running time?

> In the worst case, the running time of the algorithm is $O(n^2)$. This occurs when each recursive call decreases the size of the input by only 1, leading to a series of recursive calls that take a considerable amount of time.

- When does it occur?

  > This scenario happens when the RANDOMIZED-PARTITION function consistently picks the worst possible pivot, either the smallest or the largest element in the array, resulting in the most unbalanced partitions. In your example array $\{5, 10, 7, 3, 15, 9, 2\}$, it would occur if, while searching for the smallest element, the largest element is consistently chosen as the pivot, and vice versa.

Practical Considerations

- **Usage**: Useful in scenarios where we need to find median or other order statistics quickly.
- **Optimization**: Can be optimized further by using a hybrid approach with other efficient sorting algorithms for small arrays.

Conclusion

- **Randomized Select**: A powerful tool for finding order statistics in expected linear time, offering a good average case performance.

Worst-Case Analysis of RANDOMIZED-SELECT

In the worst-case scenario for `RANDOMIZED-SELECT`, the `RANDOMIZED-PARTITION` function always picks an element that results in the most skewed partition possible. This means that one partition has $n-1$ elements and the other has 0 elements.

Recurrence Relation:

Given this skewed partitioning, the recurrence relation for the worst-case scenario is:

$$T(n) = T(n-1) + O(n)$$

Here's a breakdown:

1. $T(n-1)$: This term represents the time taken by the recursive call on the partition with $n-1$ elements.
2. $O(n)$: This term represents the time taken to partition the array of size n. $O(n)$:

Base Case:

$$T(1) = O(1)$$

When the array has only one element, the algorithm simply returns that element, which takes constant time.

Solving the Recurrence:

To solve the recurrence relation $T(n) = T(n-1) + O(n)$, we can expand it:

$$T(n) = T(n-1) + O(n)$$

$$= [T(n-2) + O(n-1)] + O(n)$$

$$= [T(n-3) + O(n-2)] + O(n-1) + O(n)$$

$$= \dots$$

$$= T(1) + O(n) + O(n-1) + O(n-2) + \dots + O(2) + O(1)$$

$$T(n) = O(n) + O(n-1) + O(n-2) + ... + O(2) + O(1)$$

To find the Big O notation for this series, we sum up all the terms:

$$O(n) + O(n-1) + O(n-2) + \dots + O(2) + O(1) = O\left(\sum_{i=1}^{n} i\right) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2+n}{2}\right) = O(n^2)$$

$$T(n) = O(n^2)$$

So, in the worst-case scenario, the `RANDOMIZED-SELECT` algorithm has a time complexity of $O(n^2)$.

Average Case Runtime Analysis

Let $X_k = I\{$ pivot rank $= k\}$, $1 \leq k \leq n$

what is $E[X_k]$?

Step 1: Define $E[X_k]$ (步驟 1：定義 $E[X_k]$)

- $E[X_k]$ is the expected value of the random variable X_k, which indicates whether the k-th element is chosen as the pivot.
- It is calculated as $E[X_k] = \frac{1}{n}$ because the pivot is chosen randomly.

Step 2: Worst Case in Partition

- In the worst case during partition, the i-th element ends up on the side with the greater number of elements, resulting in partition sizes of $k-1$ and $n-k$.

Step 3: Recurrence Relation

- The recurrence relation for the expected runtime $E[T(n)]$ is derived considering the worst case in each partition, leading to the relation:

$$E[T(n)] \leq \sum_{k=1}^{n} X_k \cdot \max\{T(k-1), T(n-k)\} + O(n)$$

Step 4: Independent Variables

- Note that X_k and $T(\max\{k-1, n-k\})$ are independent; the runtime of a partition of size $k-1$ or $n-k$ does not depend on whether that size is selected.
- Also, T is an increasing function, so we can simplify $\max\{T(k-1), T(n-k)\}$ to $T(\max\{k-1, n-k\})$.

Step 5: Simplifying the Relation

- Now, we can simplify the relation further to find the expected runtime:

$$E[T(n)] \leq \sum_{k=1}^{n} E[X_k] \cdot E[T(\max\{k-1, n-k\})] + O(n)$$

$$T(n) \in O(n)$$

A linear time algorithm

SELECT Algorithm Example Analysis

Initial Array

A = {10, 38, 87, 55, 47, 5, 3, 9, 12, 88, 19, 22, 53, 98, 17, 37, 35, 63, 72, 33, 93, 87, 15, 66}

G1	G2	G3	G4	G5
10	5	19	**37**	93
38	3	**22**	35	**87**
87	**9**	53	63	15
55	12	98	72	66
47	88	17	33	

Step 1: Grouping and Finding Medians

Grouping: Divide array A into groups of 5 elements each, and one group with 4 elements. This takes linear time, $O(n)$.

Finding Medians: Find the median of each group through insertion sort. For instance, the first group will be {10, 38, 87, 55, 47} and its median is 47. This process is repeated for all groups.

> The medians of groups $= \{47, 9, 22, 37, 87\}$, $O(n)$

Step 2: Finding Median of Medians

Recursive Call: We find the median of the medians obtained in step 1. Let's assume the medians are $\{47, 9, 22, 37, 87\}$. The median of this set is **37**.

> the median of medians $= 37$, $O(n)$

Step 3: Partitioning Around x

Partition: We partition the array around the median of medians, 37, resulting in two subarrays: one with elements less than 37 and one with elements greater than 37.

Step 4: Recursive Calls to Find the i-th Smallest Element

Recursion: Depending on the value of i relative to k, recursively find the i-th smallest element. For instance, if we are looking for the 7th smallest element, we would proceed with the elements less than 22.

Finding a Good Pivot

- **Grouping and Sorting**: The array is divided into groups and sorted to find a good pivot, taking $O(n)$ time.
- **Median of Medians**: The median of medians is found recursively, serving as a good pivot.

Modifying Partition

- **Partition with Median of Medians**: The partition is modified to use the median of medians as the pivot, guiding the next steps based on the rank k sought.

Recurrence Relation

- **Formulation**: The recurrence relation for the worst-case running time is formulated as $T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$, where $\Theta(n)$ represents a linear time complexity.
- **Solving**: The recurrence is solved using the substitution method, proving that $T(n) = O(n)$.

Analysis of Elements Relative to the Pivot

- **Smaller Elements**: At least $3n/10 - 6$ elements are smaller than the pivot, derived from the analysis of medians and their respective groups.
- **Larger Elements**: At most $7n/10 + 6$ elements are larger than the pivot, ensuring a balanced partition most of the time.

Solution of The Recurrence Relation

The Recurrence Relation for the worst-case running time

> $T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + \Theta(n)$, where $\Theta(n)$ represents a linear time complexity.

Step 1: Aim to prove $T(n) = O(n)$ using the substitution method

> Assume $T(n) \leq cn$, aiming to find c and a.

Step 2: Substitute $T(n) \leq cn$ into the recurrence:

> $T(n) \leq c\left(\frac{n}{5} + 1\right) + c\left(\frac{7n}{10} + 6\right) + an$

Step 2.1: Distribute c in the term $c\left(\frac{n}{5} + 1\right)$ and $c\left(\frac{7n}{10} + 6\right)$

> $c\left(\frac{n}{5}\right) + c$, and $c\left(\frac{7n}{10}\right) + 6c$

> $T(n) \leq c\left(\frac{n}{5}\right) + c + c\left(\frac{7n}{10}\right) + 6c + an$

Step 2.2: Combine the n terms by adding $\frac{n}{5}c$ and $\frac{7n}{10}c$

> $T(n) \leq c\left(\frac{n}{5} + \frac{7n}{10}\right) + 7c + an$

> $T(n) \leq c\left(\frac{9n}{10}\right) + 7c + an$

Step 3: Simplify

> $T(n) \leq cn\left(\frac{9}{10}\right) + 7c + an$

Step 4: To satisfy $T(n) \leq cn$,

> $cn\left(\frac{9}{10}\right) + 7c + an \leq cn$

Step 4.1: Bring the term $cn\left(\frac{9}{10}\right)$ to the right to have it with cn:

> $7c + an \leq cn - cn\left(\frac{9}{10}\right)$, then $7c + an \leq cn\left(\frac{10}{10} - \frac{9}{10}\right)$

> Further simplify $7c + an \leq cn\left(\frac{1}{10}\right)$

Step 5: Rearrange inequality to find the terms involving a

> $7c + an \leq cn\left(\frac{1}{10}\right)$, then $an \leq cn\left(\frac{1}{10}\right) - 7c$

> $cn\left(\frac{1}{10}\right) \geq an - 7c$,

Step 6: Divide both sides by n and then by c to get:

> $\frac{1}{10} \geq \frac{a}{c} - \frac{7}{n}$

Step 7: Rearrange to find a condition involving a and c:

> $\frac{c}{10} - a \geq \frac{7}{n}$

Step 8: Ensure a positive left-hand side:

> $\frac{c}{10} - a > 0$

Step 9: find a condition on n, rearrange the inequality from step 7:

> Step 7, $\frac{c}{10} - a \geq \frac{7}{n}, n \cdot \left(\frac{c}{10} - a\right) \geq n \cdot \left(\frac{7}{n}\right)$

Step 9.1 Isolate n

> $n\left(\frac{c}{10} - a\right) \geq 7$, then $n \geq \frac{7}{\left(\frac{c}{10} - a\right)}$

Step 9.2 Making it a Strict Inequality

> $n > \frac{7c}{\frac{c}{10} - a}$