CS-204: Design and Analysis of Algorithms

220001038, 220001039, 220001040

February 15, 2024

Lower Bound for Searching:

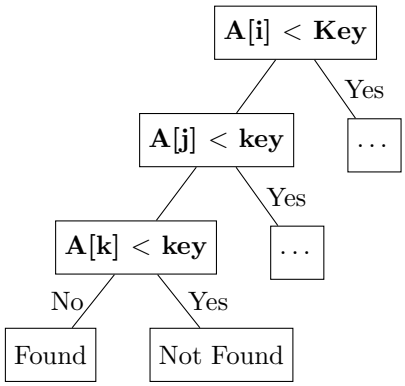
Decision Tree Model:

For searching, each node in the decision tree represents a comparison between elements. Each edge leaving a node corresponds to the possible outcomes of that comparison. Leaves of the tree represent the possible search outcomes (found or not found).

Leaf Nodes:

If there are n elements in the data set, there are atleast n possible outcomes for searching (found at position 1, found at position 2, ..., not found or found in between elements). The height of the decision tree corresponds to the number of comparisons required.

Decision Tree for Searching:



Lower Bound Analysis:

For any binary tree with height h, there are at most 2^h leaves. In our case, we want to find the minimum height h such that $2^h \geq n + 1$. Solving for h, we get $h \geq \log_2(n + 1) - 1$. Therefore, the lower bound for the number of comparisons in a search algorithm is $\Omega(\log n)$.

Table 1: **Decision tree components(Analogy)**

Decision Tree	Algorithm
Internal node	Comparison
Leaf node	Solution
Path from root to leaf	One execution of the algorithm
Path length	Running time
Height of tree	Worst case Running time

Lower Bound for Sorting:

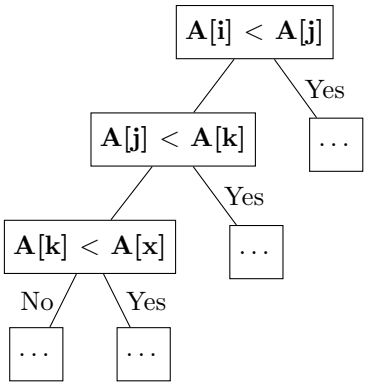
Decision Tree Model:

For comparision based sorting, the decision tree represents the comparisons needed to determine the sorted order of the elements.

Leaf Nodes:

There are atmost $n!$ possible permutations of the n elements, and each leaf corresponds to one possible permutation. Few of those $n!$ permutations are not possible due to the above mentioned constraints in the decision tree.

Decision Tree for Sorting:



Lower Bound Analysis:

The number of leaves in the decision tree must be at least $n!$ (factorial). The height of the decision tree is then at least $\log_2(n!)$, and by Stirling's approximation, it's approximately $\Omega(n \log n)$.

Prove : log(n!) = Ω(n $\log n$)

Proof:

$\log(n!)$

=log(n $\cdot(n-1)\cdot \ldots \cdot 1)$

$= \log(n) + \log(n-1) + \ldots + \log(1)$

$= \sum_{i=1}^{n} \log(i)$

$\sum_{i=1}^{n} \log(i) > \sum_{i=n/2}^{n} \log(i)$

$\sum_{i=1}^{n} \log(i) \geq \sum_{i=n/2}^{n} \log(n/2))$ **or** $\sum_{i=1}^{n} \log(i) \geq \ n/2 \log(n/2)$

Problem 1: Finding Second Minimum in Array

Method 1: Sorting Method

Approach: Sort the array and pick the second element.

Complexity: $O(n \log n)$

Algorithm 1 Finding the Second Minimum Element using Sorting

```
1: procedure SecondMax(arr[1 ... n])
2:     if n < 2 then
3:         error "Array has less than two elements"
4:     end if
5:     Sort arr                                  ▷ Sort the array in ascending order
6:     return arr[1]                             ▷ Return the second element
7: end procedure
```

Comparison Analysis: Less efficient for this specific task due to higher time complexity.

Method 2: Tournament Method

Approach: Pair up elements iteratively to find winners.

Complexity: $O(n)$

Algorithm 2 Finding the Second Minimum Element using Tournament Tree

```
1: procedure SecondMin(arr)
2:     n ← length of arr
3:     if n < 2 then
4:         error: Array size should be at least 2
5:     end if
6:     tournamentTree ← buildTournamentTree(arr)          ▷ Build the tournament tree
7:     return findSecondMin(tournamentTree, arr[0])       ▷ Find the second minimum
8: end procedure
9: function buildTournamentTree(arr)
10:    tree ← initialize an array to represent the tournament tree
11:    for i ← 0 to n − 1 do
12:        tree[i] ← arr[i]
13:    end for
14:    for i ← n to 2n − 2 do
15:        tree[i] ← ∞                                    ▷ Initialize non-leaf nodes with infinity
16:    end for
17:    for i ← 2n − 3 down to 1 do
18:        tree[i] ← min(tree[2i], tree[2i + 1])          ▷ Build the tournament tree
19:    end for
20:    return tree
21: end function
22: function findSecondMin(tree, winner)
23:    min1 ← tree[1]                  ▷ The root of the tournament tree is the minimum
24:    min2 ← ∞                                ▷ Initialize second minimum with infinity
25:    for i ← 2 to length of tree − 1 do
26:        if tree[i] < min2 and tree[i] > winner then
27:            min2 ← tree[i]
28:        end if
29:    end for
30:    return min2
31: end function
```

Comparisons: $n + logn$ **(in the worst case).**

The tournament tree formed during the process is a binary tree. The height of this binary tree is $\lceil \log_2 n \rceil$**.**

At each level of the tree (except the last one), there are pairwise comparisons, and the number of levels is $\lceil \log_2 n \rceil$**.**

So, the order of total number of comparisons C **is proportional to the height of the binary tree, which is** $\lceil \log_2 n \rceil$**.**

The additional n comparisions for finding the second maximum from the runners of the tournament.

Comparison Analysis: Efficient for finding the second minimum with linear time complexity.

Method 3: Linear Scan

Approach: Scan through the array to find the second minimum.

Complexity: $O(n)$ **Comparisons:** $2n$ **(scan through the array twice).**

Algorithm 3 Finding the Second Maximum Element

```
procedure SecondMax(arr)
    max1 ← −∞
    max2 ← −∞
    for element in arr do
        if element > max1 then
            max2 ← max1
            max1 ← element
        else if element > max2 then
            max2 ← element
        end if
    end for
    return max2                                  ▷ Second maximum element
end procedure
```

Comparison Analysis: Efficient but may involve unnecessary comparisons in some cases.

Comparison:

Tournament method stands out for its linear time complexity, making it efficient for finding the second minimum.

Sorting method and linear scan are efficient but may have higher time complexities in comparison.

Dryrun for tournament Method for Second Maximum Element

The tournament method involves dividing the array into pairs and finding the maximum in each pair until a single winner (maximum element) is determined. Then, the second maximum can be found by excluding the maximum and finding the maximum again in the remaining elements.

Let's consider an array A **with** n **elements:** $A = [a_1, a_2, \ldots, a_n]$.

Let $A = [a_1, a_2, \ldots, a_n]$ **be the array.**

1. Divide the array into pairs and find the maximum in each pair.

$$\textbf{max\_arr} = [\max(a_1, a_2), \max(a_3, a_4), \ldots, \max(a_{n-1}, a_n)]$$

2. Recursively apply step 1 to the max_arr until a single winner is determined.

3. The winner of the tournament is the maximum element in the array.

4. Remove the maximum element from the array and find the maximum in the remaining elements.

$$\textbf{second\_max} = \max(\textbf{remaining elements})$$

Problem 2: Finding Kth Minimum Element in Array

Method 1: Priority Queue Method

Approach: Use a min-heap (priority queue) and extract the k**th minimum.**

Complexity: $O(n \log k)$ **Comparison Analysis: Efficient when pivot is selected wisely.**

Method 2: QuickSelect Algorithm

Approach: Choose pivot, partition array, and recurse based on desired k.

Complexity: $O(n)$ **(average case) because...**

Partitioning: In each step of the QuickSelect algorithm, a pivot element is chosen, and the list is partitioned into two sublists – elements smaller than the pivot and elements larger than the pivot.

Recursive Call: Depending on the position of the k-th element relative to the pivot, the algorithm decides which sublist to recurse into. If the pivot is in the correct position (equal to k), the algorithm is done. Otherwise, it continues the process in the appropriate sublist.

Average Pivot Choice: The average-case analysis assumes that, on average, the chosen pivot is somewhat close to the true median of the list. If this is the case, the partitioning tends to be balanced, meaning both resulting sublists have approximately equal sizes.

Significant Reduction in Problem Size: When the partitioning is balanced, the size of the problem (number of elements to consider) is significantly reduced with each recursive call. In each step, a fraction of the elements is discarded, leading to a fast reduction in the problem size.

Expected Linear Time: Due to the efficient reduction of the problem size in each step, the average-case time complexity of QuickSelect is $O(n)$. It means that, on average, the algorithm processes a constant fraction of the remaining elements in each recursive call, resulting in a linear overall time complexity.

Algorithm 4 Finding the kth Minimum Element using Max Heap

```
procedure KthMin(maxHeap, k)
    build max heap with all the elements of the aray(maxHeap)
    i ← 1
    while i < k do
        pop(maxHeap)                                  ▷ Remove the maximum element
        i ← i + 1
    end while
    return top(maxHeap)                               ▷ Return the kth minimum element
end procedure
```

Algorithm 5 QuickSelect for Finding the kth Minimum Element

```
 1: procedure QuickSelect(arr, low, high, k)
 2:     if low = high then
 3:         return arr[low]                            ▷ Found the kth minimum element
 4:     end if
 5:     pivotIndex ← Partition(arr, low, high)
 6:     if k = pivotIndex then
 7:         return arr[pivotIndex]                     ▷ Found the kth minimum element
 8:     else if k < pivotIndex then
 9:         return QuickSelect(arr, low, pivotIndex − 1, k)
10:     else
11:         return QuickSelect(arr, pivotIndex + 1, high, k)
12:     end if
13: end procedure
14: procedure Partition(arr, low, high)
15:     pivot ← arr[high]
16:     index ← low − 1
17:     for j ← low to high − 1 do
18:         if arr[j] ≤ pivot then
19:             Swap arr[j] and arr[++index]
20:         end if
21:     end for
22:     Swap arr[++index] and arr[high]
23:     return index
24: end procedure
```

Comparison Analysis: Provides average logarthmic-linear time complexity but may degrade in the worst case.

Comparison:

Priority Queue method is efficient when k is small compared to n.

QuickSelect provides average linear time complexity but may degrade in the worst case.

BST method has a worst-case complexity similar to the sorting method but can be efficient in some scenarios.