Design and Analysis of Computer Algorithms

11059005 蕭耕宏

Assignment 2

Due: April 14, 2025 (before class)

Please use A4-sized papers to write your answers for the homework sets.

1. An array A[1 . . . n] is said to have a majority element if more than half of its entries are the same. Your job is to design an efficient algorithm to tell if the given array has a majority element. If so, please also output that element. Note that the given array is not necessarily sorted but you can answer whether two arbitrary elements in the array are equal or not in constant time.

   (a) Show how to solve this problem in O(n log n) time.

   (b) Can you give a linear-time algorithm?

   Solution:

   a. using divide and conquer, the problem can be solved in O(n log n) time. First, divide the array into two halves,

left/right	have majority	do not have majority
have majority	case1	case2
do not have majority	case2	case3

- case1:
  - if two halves have the same majority element, then the majority element is found.
  - otherwise, no majority element is found.
- case2:
  - if one of the halves has a majority element, count the number of occurrences of that element in both halves.
- case3:
  - if both halves don’t have a majority element, then the array does not have a majority element.

To cover all cases, we need to count the number of occurrences of the majority element (if any) in both halves, which takes O(n) time. The recurrence relation is T(n) = 2(T(n/2) + O(n)), which solves to T(n) = O(n log n) by the Master Method.

b. To solve the problem in linear time, we can use the Boyer-Moore Voting Algorithm. The algorithm works as follows:

- Initialize a candidate variable and a count variable.
- Iterate through the array, updating the candidate and count variables based on the current element.
- If the count variable reaches zero, set the candidate to the current element and reset the count to one.
- If the current element is equal to the candidate, increment the count.
- If the current element is different from the candidate, decrement the count.
- After iterating through the array, the candidate variable will hold the majority element (if it exists).
- Finally, we need to verify if the candidate is indeed the majority element by counting its occurrences in the array.

```
FindSignificantInversions(A):
count = 0                          // O(1)
candidate = A[0]                   // O(1)
for i = 1 to n - 1                 // O(n)
    if count == 0                  // O(n)
        candidate = A[i]           // O(n)
        count = 1                  // O(n)
    else if A[i] == candidate      // O(n)
        count = count + 1          // O(n)
    else                           // O(n)
        count = count - 1          // O(n)
    return candidate               // O(1)
```

The time complexity of the Boyer-Moore Voting Algorithm is O(n) for finding the candidate and O(n) for verifying it, resulting in a total time complexity of O(n).

2. Given a sequence X =< x1, x2, . . . , xn > of n distinct numbers, please given an algorithm to find the largest and smallest elements in X. Your algorithm should take only $\lceil \frac{3n}{2} \rceil - 2$ comparisons. Furthermore, please argue that your algorithm is optimal in terms of number of comparisons.

   Solution:

   To find the largest and smallest elements in a sequence of n distinct numbers, we can use a tournament-style approach. The algorithm works as follows:

   when n is even:
   - Pair up the elements and compare each pair.
   - For each pair, keep track of the larger and smaller elements.

   The total comparisons will be (3n)/2 comparisons.

   when n is odd:
   - Pair up the first n-1 elements and compare each pair.
   - for each pair, keep track of the larger and smaller elements.
   - Compare the last element with the current largest and smallest elements.
   - For the last element, we need 2 comparisons.

   The total comparisons will be (3 (n-1))/2 + 2 = (3n + 1)/2 comparisons.

   Additionally, we can remove the comparison of the larger and smaller elements in the first pair by setting the first pair as the initial largest and smallest elements. This will reduce the total comparisons by 2.

   For even n:
   - The number of comparisons is (3n)/2 - 2.

   For odd n:
   - The number of comparisons is (3n + 1)/2 - 2.

   Generally, the number of comparisons is ceil((3n)/2) - 2, which is the optimal solution.

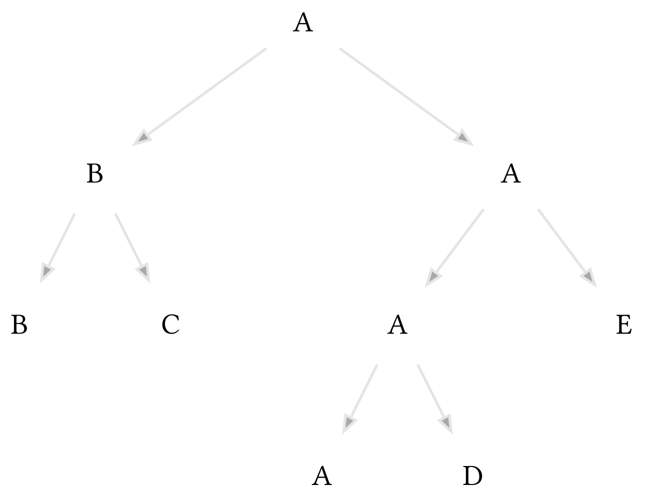
3. Show that to find the second largest one of a list of n numbers, we need at least $n - 2 + \lceil \log n \rceil$ comparisons.

Solution:

To count the number of comparisons needed to find the second largest element in a list of n numbers, we can use a tournament-style approach.

To find the second largest element, we have to find the largest element first. This requires n - 1 comparisons, because all elements except winner must lose once . At the same time, we will keep track of the elements that lost to the largest element in the tournament. The number of elements that lost to the largest element is $\lceil \log n \rceil$, as each round eliminates half of the elements. After finding the largest element, we need to find the largest element among the elements that lost to it. This requires $\lceil \log n \rceil - 1$ comparisons. Therefore, the total number of comparisons needed to find the second largest element is: $n - 1 + \lceil \log n \rceil - 1 = n - 2 + \lceil \log n \rceil$.

Below is a simple example of a tournament tree to illustrate the process of finding the second largest element.



In this example, the secondn largest element is B. The second child has two leaves, A and D. The tree structure represents the comparisons made during the tournament. The number of comparisons needed to find the second largest element after finding the largest element is $\lceil \log 5 \rceil - 1 = 2$ (comparison between B, D and E). Therefore, the total number of comparisons needed to find the second largest element is $n - 2 + \lceil \log n \rceil = 5 - 2 + 3 = 6$.

4. Suppose that you are given n red and n blue jugs, all of different shapes and sizes. All red jugs hold different amount of water, as do the blue ones. Moreover, for every red jugs, there is a blue jugs that holds the same amount of water, and vice versa. Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

   (a) Describe a deterministic algorithm that uses Θ(n^2) comparisons to group the jugs into pairs.

   (b) Prove a lower bound of Ω(n log n) for the number of comparisons that an algorithm solving this problem must make.

   Solution:

a. To group the jugs into pairs, we can use a brute-force approach. The algorithm works as follows:

For each red jug, compare it with each blue jug iteratively. Since there are n red jugs and n blue jugs, we will have n^2 comparisons in total.

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

The algorithm can be implemented as follows:

```
GroupJugs($redJugs$, $blueJugs$):
1  for i = 1 to n                                  // O(n)
2      for j = 1 to n                              // $O(n^2)$
3          compare(redJugs[i], blueJugs[j])        // $O(n^2)$
4  return pairs                                    // O(1)
```

The time complexity of the algorithm is O(n^2) because we are performing n comparisons for each red jug, and there are n red jugs. The total number of comparisons is O(n^2). The algorithm is deterministic because it systematically compares each red jug with each blue jug.

b. To prove a lower bound of Ω(n log n) for the number of comparisons, we can use a decision tree argument.

By treating each comparison as a decision node in a binary tree, we can see that the number of comparisons needed to group the jugs into pairs is at least the height of the decision tree. Since we can infer the number of leaf nodes in the tree is at least n!, the height of the tree must be at least log(n!). By Stirling’s approximation, we have:

$\log(n!) = n \log n - n$

Therefore, the height of the decision tree is at least Ω(n log n). This implies that any algorithm solving this problem must make at least Ω(n log n) comparisons.