CS 577 # Assignment 5 – Divide and Conquer Fall 2023

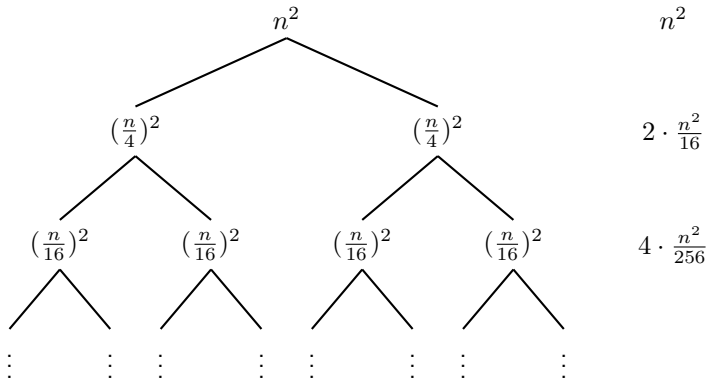
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Divide and Conquer

1. *Erickson, Jeff. Algorithms (p.49, q. 6).* Use recursion trees to solve each of the following recurrences.
   (a) $C(n) = 2C(n/4) + n^2$; $C(1) = 1$.

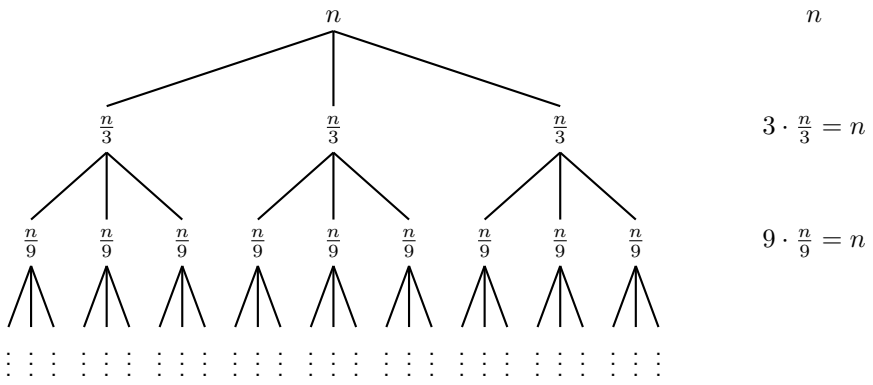
Solution:



Let d be the tree depth. The runtime is $\sum_{i=0}^{d} 2^i \cdot \frac{n^2}{16^i} = n^2 \sum_{i=0}^{d} \left(\frac{2}{16}\right)^i \leq n^2 c$ for some constant c. So the runtime is $O(n^2)$.

   (b) $E(n) = 3E(n/3) + n$; $E(1) = 1$.

Solution:



The tree depth is $\log_3 n$ and each level sums to n, so the runtime is $O(n \log_3 n) \equiv O(n \log n)$.

2. *Kleinberg, Jon. Algorithm Design (p. 246, q. 1).* You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the nth smallest value.

   However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

   (a) Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Solution: Label the databases A and B. Keep track of lower bounds in A_{lower} and B_{lower} (initialized to 1) and upper bounds in A_{upper} and B_{upper} (initialized to n). Let a and b be the results of querying A and B, respectively.

Base Case: If $A_{upper} \leq A_{lower}$ and $B_{upper} \leq B_{lower}$, the median is $min(a, b)$.

Recursive Case:

Query A on $k_1 = \frac{A_{lower}+A_{upper}}{2}$ and B on $k_2 = \frac{B_{lower}+B_{upper}}{2}$ (assume integer division).

If $a > b$, set $A_{upper} = k_1$ and $B_{lower} = k_2 + 1$ (don't change A_{lower} and B_{upper}) and recurse.

If $a < b$, set $A_{lower} = k_1 + 1$ and $B_{upper} = k_2$ (don't change A_{upper} and B_{lower}) and recurse.

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

Solution: Observe that, in each recursive call, the range

$$A[A_{lower}, \ldots, A_{upper}] \cup B[B_{lower}, \ldots, B_{upper}]$$

of values left to search is cut in half, either to

$$A\left[A_{lower}, \ldots, \frac{A_{lower} + A_{upper}}{2}\right] \cup B\left[\frac{B_{lower} + B_{upper}}{2} + 1, \ldots, B_{upper}\right] \tag{1}$$

if $a > b$, or to

$$A\left[\frac{A_{lower} + A_{upper}}{2} + 1, \ldots, A_{upper}\right] \cup B\left[B_{lower}, \ldots, \frac{B_{lower} + B_{upper}}{2}\right]$$

if $a < b$. So the recurrence is $T(n) = T(n/2) + O(1)$; $T(1) = 1$. This yields a recurrence tree with $\log(n)$ layers and $O(1)$ work at each layer, hence the solution is $O(\log n)$.

(c) Prove correctness of your algorithm in part (a).

Solution: Let x be the desired nth smallest value, and let $R_A = A[A_{lower}, \ldots, A_{upper}]$ and $R_B = B[B_{lower}, \ldots, B_{upper}]$. We prove the following statement $(*)$ by (strong) induction on the size $A_{upper} - A_{lower} + 1 + B_{upper} - B_{lower} + 1$ of the remaining search range $R_A \cup R_B$.

$(*)$ If the (smaller) median of $R_A \cup R_B$ is x, then the algorithm returns x.

Initially, $A_{lower} = B_{lower} = 1$ and $A_{upper} = B_{upper} = n$, so $(*)$ asserts our algorithm is correct.

For the base case $A_{upper} - A_{lower} = 0$ and $B_{upper} - B_{lower} = 0$ (equivalently $A_{upper} = A_{lower}$ and $B_{upper} = B_{lower}$), $R_A \cup R_B$ contains just two values, the smaller of which, by assumption, is x. The algorithm returns the min of the two values, which is x, so $(*)$ holds.

Now assume $(*)$ holds for all ranges smaller than $R_A \cup R_B$. Observe that a and b are the medians of R_A and R_B, respectively.

Suppose $a > b$. Then $x \leq a$. Otherwise, if $x > a$, then x is larger than more than half the elements of R_A, and, since $x > a > b$, x is also larger than more than half the elements of R_B. This contradicts the assumption that x is the median of $R_A \cup R_B$. Similarly, $x \geq b$. In this case, the algorithm restricts to the range $R'_A \cup R'_B$ in (1). This removes $(A_{upper} + A_{lower})/2 - 1$ elements smaller than a, hence smaller than x, and $(B_{upper} + B_{lower})/2 - 1$ elements larger than b, hence larger than x. By symmetry of the A and B indices, $(A_{upper} + A_{lower})/2 - 1 = (B_{upper} + B_{lower})/2 - 1$, so, to create $R'_A \cup R'_B$, we have removed from $R_A \cup R_B$ an equal number of elements larger and smaller than x, which by assumption is the median of $R_A \cup R_B$. Hence x is also the median of $R'_A \cup R'_B$. By induction, $(*)$ also holds for the smaller range $R'_A \cup R'_B$, and x is the median of $R'_A \cup R'_B$, so by $(*)$ the algorithm returns x.

A symmetric proof applies for the case $a < b$.

3. *Kleinberg, Jon. Algorithm Design (p. 246, q. 2).* Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, ..., a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

   We motivated the problem of counting inversions as a good measure of how different two orderings are. However, this measure is very sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$.

   (a) Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

Solution:

Return Values: The number of significant inversions and sorted version of input sequence.

Divide: Split the input sequence into two halves and recursively calculate the number of significant inversions in each half as well as the sorted version of the input sequence.

Merge: Since we have the result of inversions in the left and right halves, we just need to calculate inversions between halves. Let the left and right half be L and R respectively. Let i and j be initialized to the size of L and R respectively. Initialize result N to be the sum of the counts of inversions exclusively on the left and right halves. If $L[i] \leq 2R[j]$, then, if $j > 1$, set $j \leftarrow j - 1$, and, if $j = 1$, stop. If $L[i] > 2R[j]$, set $N \leftarrow N + j$. Then, if $i > 1$, set $i \leftarrow i - 1$, and, if $i = 1$, stop. Finally, use the mergesort merge step to generate the sorted sequence to return. Return N and the sorted sequence.

Base case: If the length of the input sequence is 1, just return 0 inversions and the input sequence.

(b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

Solution: In the Divide step, the algorithm makes two recursive calls, each on half of the array ($2T(n/2)$). The loop in the Merge step goes through the left and right half exactly once, then runs the mergesort merge procedure, each of which takes time linear in n.

$$T(n) = 2T(n/2) + O(n).$$

This is identical to the recurrence for mergesort, and has solution $O(n \log n)$.

(c) Prove correctness of your algorithm in part (a).

Solution: We prove the following statement $(*)$ by (strong) induction on the length k of the input sequence.

$(*)$ On input sequences of length k, the algorithm returns the correct number of inversions and the correctly sorted sequence.

For $k = n$, $(*)$ asserts that our algorithm is correct.

Base case: For $k = 1$, the algorithm correctly asseses that a sequence of length 1 has no inversions and is already sorted.

Induction hypothesis: $(*)$ holds for sequences of length less than k.

Induction step: The Divide step recurses on the first and second half of the sequence, each of which have length $< k$, so, by our induction hypothesis, the algorithm obtains a correct value N and the two half-sequences are properly sorted. It remains to show that the Merge step is correct. If $L[i] \leq 2R[j]$, then (i, j) do not form a significant inversion. If $L[i] > 2R[j]$ then (i, j) are a significant inversion, and furthermore, since the right half is sorted in increasing order, (i, ℓ) are a significant inversion for every $1 \leq \ell \leq j$. Hence the algorithm correctly adds j significant inversions. Since every previous j satisfied $L[i] \leq 2R[j]$, there are no more significant inversions involving i, so the algorithm correctly decrements i. Since the left half is sorted, we don't miss any significant inversions by maintaining the same j value while decrementing i.

Thus the Merge steps correctly computes the number of significant inversions. We know from lecture that the mergesort merge step is correct, so $(*)$ holds.

4. *Kleinberg, Jon. Algorithm Design (p. 246, q. 3).* You're consulting for a bank that's concerned about fraud detection. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud.

   It's difficult to read the account number off a bank card directly, but the bank has an "equivalence tester" that takes two bank cards and determines whether they correspond to the same account.

   Their question is the following: among the collection of n cards, is there a set of more than $\frac{n}{2}$ of them that all correspond to the same account? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester.

   (a) Give an algorithm to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

   Solution:

   Return values: Card that represents majority element in a group.

   Divide: Split the cards into two halves and recursively find the majority element of each half.

   Merge: Since the majority element in a subproblem must also be the majority element in at least one of its halves, consider the majority element (if any) returned by each half.

   - If both sides have the same majority element, return this as the majority element.
   - If neither side has a majority element, return "none".
   - Otherwise, let a and b be the majority elements of the left and right halves, respectively. Compare a and b with every element in both halves to obtain numbers n_a and n_b of elements matching a and b in the combined array. If $n_a > \frac{n}{2}$, return a; if $n_b > \frac{n}{2}$, return b, and if both $n_a, n_b \leq \frac{n}{2}$, return "none".

   Base case: Only one card to compare. No need to invoke equivalence tester, but this card is the majority element of its group (size 1).

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

   Solution: In the Divide step, the algorithm makes two recursive calls, each on half of the array ($2T(n/2)$). The Merge step, in the worst case, compares every element in the array to both a and b, requiring $2n$ steps. So the recurrence is

   $$T(n) = 2T(n/2) + O(n).$$

   This is identical to the recurrence for mergesort, and has solution $O(n \log n)$.

(c) Prove correctness of your algorithm in part (a).

Solution: We prove the following statement $(*)$ by (strong) induction on the length k of the input sequence.

$(*)$ On input of k cards, the algorithm correctly determines whether a majority exists and, if so, returns a member of the majority.

For $k = n$, $(*)$ asserts that our algorithm is correct.

Base case: For $k = 1$, the algorithm correctly returns the only card as representing a majority.

Induction hypothesis: $(*)$ holds for sets of fewer than k cards.

Induction step: The Divide step recurses on the first and second half of the cards, each of which is a set of $< k$ cards, so, by our induction hypothesis, the recursive call correctly determines the majority. It remains to show that the Merge step is correct. Let a, b, n_a, n_b be defined as in the solution to part (a).

1. If $a = b$ and $n_a, n_b > (n/2)/2$ (both sides have the same majority element), then $n_a + n_b > n/2$, so indeed $a = b$ is a majority element.
2. If neither side has a majority, then for any x in the left half and y in the right half, $n_x + n_y \leq (n/2)/2 + (n/2)/2 = n/2$, so the combined list has no majority.
3. If the sides have different majority elements a and b, then by similar reasoning to case 2, no element other than a or b can form a majority, so the algorithm behaves correctly.

Thus the algorithm returns the correct majority element, so $(*)$ holds.

5. **Inversion Counting:**
Implement the optimal algorithm for inversion counting in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(n \log n)$ time, where n is the number of elements in the list.

The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of elements in the list.

Note that the results of some of the test cases may not fit in a 32-bit integer.

A sample input is the following:

```
2
5
5 4 3 2 1
4
1 5 9 8
```

The sample input has two instances. The first instance has 5 elements and the second has 4. For each instance, your program should output the number of inversions on a separate line. Each output line should be terminated by a newline. The correct output to the sample input would be:

```
10
1
```