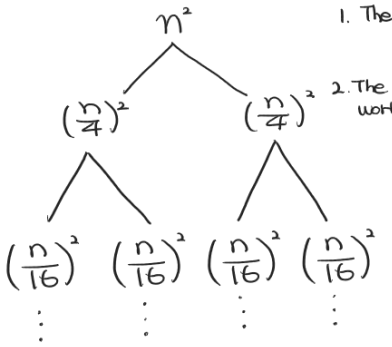
CS 577 # Assignment 5 – Divide and Conquer Spring 2024

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: Ki Min Kang Wisc id: 908 404 5062

Divide and Conquer

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

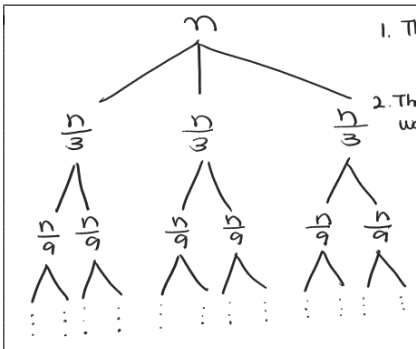


1. The root nodes represents the original problem C(n) with a workload of n^2
2. The next level divides the problem into two C(n/4), each with a workload of $(n/4)^2$
3. This process repeats, and at each level i, the workload is $2^i \times (\frac{n}{4^i})^2 = \frac{n^2}{2^i}$

The tree continues for $O(\log_4 n)$ levels, and the total workload is $\frac{1}{1-\frac{1}{8}} = \frac{8}{7}$, $\frac{8}{7} \times n^2$

Therefore, the recurrence relation has a time complexity of $O(n^2)$

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



1. The root nodes represents the original problem E(n) with a workload of n
2. The next level divides the problem into two $E(\frac{n}{3})$, each with a workload of $\frac{n}{3}$
3. This process repeats, and at each level, the size of the sub problems reduces by a factor of 3

Since the work at each level is O(n), and there are $O(\log_3 n)$ levels, the total work done is $O(n \log_3 n)$.

Therefore, the recurrence relation has a runtime complexity of O(nlogn)

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.

Algorithm to find the median value from two databases with O(logn):
1. Name the databases as DB1 and DB2.
2. Keep track of the lower and upper bounds for each database, initializing as L1 and L2 for lower bounds, and U1 and U2 for upper bounds
3. Let the results from querying DB1 and DB2 be result1 and result2 respectively.

Base Case: If U1 ≤ L1, the median is min(result1, result2)

Recursive Case: Query database DB1 for the middle value using midpoint formula '(Upper bound of DB1 + Lower bound of DB1)/2 and DB2 using '(Upper bound of DB2 + Lower bound of DB2)/2 using integer division.

If the result from DB1 is greater than the result from DB2, then update the upper bound of DB1 to be midpoint of DB1, and update the lower bound of DB2 to be one more than the midpoint of DB2. If the result from DB1 is less than the result from DB2, then update the bounds in the opposite way.

This approach will continue to halve the search space in each database with each query, converging on the value in O(logn)

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

In every recursive step, the algorithm narrows down the search space by splitting either DB1 or DB2 into two halves. This is determined by the comparison between result1 and result2. If result1 is greater than result2, DB1 is split, otherwise, DB2 is split. The division effectively halves the number of elements under consideration.
The recurrence formula for the algorithm's runtime is $T(n) = T(n/2) + O(1)$ with a base case of $T(1) = 1$.

The division process creates a binary tree of recursive calls with logarithmic depth, which means the overall runtime is O(logn), where n is the number of elements in the sets.

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

The algorithm is designed to find the nth smallest element from two sorted arrays (DB1 and DB2).

Let's say: R1 = DB1[L1 ..... U1]
R2 = DB2[L2 ..... U2]

Assume that our algorithm works correcely when the segments are very small.
Specifically, when each segment contains only one element, the algorithm simply picks the smaller of the two, which must be the nth smallest element by definition.

Next, we will use induction to prove if our algorithm works for segments of any size, it must also work when the segments are one step larger.
We compare the medians of segments R1 and R2. The median is the middle value, or if there is an even number of elements, the lower of the two middle values.

If the median of R1 is greater than the median of R2, we know that the nth smallest element cannot be in the upper half of R1. Similarly, it cannot be in the lower half of R2. So, we eliminate these parts. The same logic applies if the median of R1 is less than the median of R2.

We repeat the process, halving the size of our search space each time, until we find the nth smallest element. Since we halve the search space each time, it takes O(log n) steps.

In conclusion, the algorithm is correct

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.

Algorithm to count significant inversions:

Process:

0) Base Case:
If a sequence has only one element, return zero inversions and the original sequence.

1) Divide:
Break the sequence into two parts, and recursively determine significant inversions and sort each part.

2) Merge:
With inversion counts for each part, calculate cross-inversions.
Let L= Left and R= Right
Traverse L backwards with index i and R backwards with index j.
If an element in L is more than twice any in R, increment the count of significant inversions by j. Then move to the next element in L and repeat until all elements are compared or the condition no longer holds.

3) Combine:
Use a mergesort-like merge step to combine the sorted halves into one sorted sequence, and return this with the total inversion count.

Output:
Returns the count of significant inversions and the sorted sequence.

(b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

The algorithm divides the problem into two parts by making two recursive calls, each operation on half of the original array ($2T(n/2)$)

Then, during the merging step, it runs through both halves of the array once to combine them back together in a sorted manner, which takes linear time $O(n)$.

So, the runtime recurrence for this algorithm is:
$T(n) = 2T(n/2) + O(n)$

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

We want to show that for any sequence of numbers of length K, our algorithm will correctly count the number of significant inversions and sort the sequence.

Base Case:
When K=1, there are no inversions, and it is trivially sorted.

Inductive Step:
Assume that our algorithm works perfectly for sequences shorter than K.
We split the sequence into two halves and apply our algorithm to each half, which are shorter than K.
By inductive hypothesis, we can count on them being sorted and the inversions in them counted correctly after the algorithm is applied.
Next, we need to merge these two sorted halves together and count significant inversions.
Let's call the number in the first half L[i] and the second half R[j]

While merging, if L[i] is not more than twice R[j], then (i, j) is not a significant inversion.
But if L[i] is more than twice R[j], then (i, j) is a significant inversion.
As we merge, whenever we move a number from the right half, we can count inversions, because the left half is sorted, if L[i] is more than twice R[j], it will also be more than twice all the numbers up to R[j].
So, we can just add j to count of significant inversions
By correctly counting these inversions and merging the two halves, the algorithm correctly counts from both halves.

Therefore, the algorithm is correct for sequences of any length.

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.

Algorithm to find out if there is a bank card that represents more than half of the n cards.

Process:

0) Base Case:
If there is only one card, it's considered the majority element by default since there is nothing to compare it with

1) Divide:
Split the set of cards into two groups

2) Merge:
Recursively apply the algorithm to each half to find the majority element of each.
If both groups return the same card, that's the majority.
If there is no majority in both groups, return null.
If groups return different cards, compare each with all cards:
If one card is the majority, return it.
If neither is, return null.

Output:
The card that is the majority, or null if there is no majority

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence. Show your work and do not use the master theorem.

The algorithm splits into two parts and makes a recursive call on each half, doing the same operation twice on half the number of elements ($2T(n/2)$)

For merge step, in the worst-case, it checks every element in the array against both possible majority candidates, which requires a linear number of comparisons ($2n$ comparisons).

So, the recurrence formula of operations $T(n)$ is:

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

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

We will show the algorithm always can find a majority element in a set of cards, if one exists.

Base Case:
When we have just one card, it's automatically the majority since there's nothing else to compare it to.

Inductive Step:
Assume that the algorithm works perfectly when we find the majority element for any number of cards When we split the cards into two halves and recursively, we're left with two elements, a and b, each possibly being the majority of their respective halves.

- If a and b are the same, and they together make up more than half of the set, then that card is the majority
- If neither half has a majority element, then there can't be a majority in the combined set.
- If a and b are different, but neither appears more than half the time in their combined halves, then there's no majority

Therefore, our algorithm can find the majority element.

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, k, giving the number of instances that follow. For each instance, there will be a positive integer, j, giving the number of elements in the list.

Input constraints:

- $0 \leq$ k ≤ 1000
- $0 \leq$ j ≤ 100000
- $0 \leq$ list elements $\leq 2^{31} - 1$

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
```