CS 577 **Assignment 5 – Divide and Conquer** Spring 2023

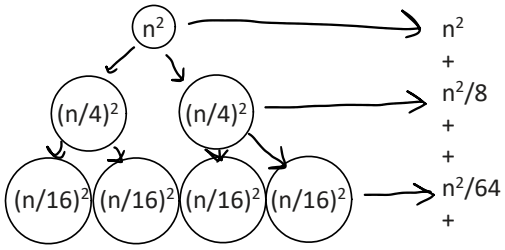
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Carson Miller Wisc id: 9081473028

Divide and Conquer

1. *Erickson, Jeff. Algorithms (p.49, q. 6).* Use recursion trees to solve each of the following recurrences.

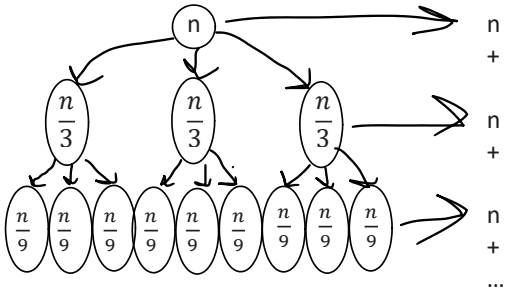
(a) $C(n) = 2C(n/4) + n^2$; $C(1) = 1$.



Each row sums to the value on the right

$$n^2 \sum_{i=0} (\frac{1}{8})^i = \frac{8}{7}n^2 = O(n^2)$$

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



Each row sums to the value on the right

$$Height\ of\ tree * cost\ per\ level\ =\ runtime$$
$$log_3(n) * n\ =\ O(nlog_3(n))$$

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.

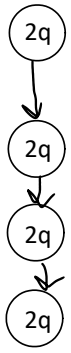
   (a) Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Query both databases at n/2th smallest element, say they return x and y (from databases 1 and 2, respectively).
If each database only has one element, we have found the median and should return (x+y)/2.
If x>y, discard all elements in database 1 greater than x and all elements in database 2 lesser than y.
If x<y, discard all elements in database 1 lesser than x and all elements in database 2 greater than x.
Make recursive call on remaining databases.

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

T(n) = T(n/2) + 2c

$$tree\_height * work\_done = runtime$$
$$\log(n) * 2q = 2q * \log(n) = O(\log(n))$$



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

Assume the recursive call returns the median of both databases passed to it. We will halve each database until there is only 1 element remaining in each. As we do that, we will be removing the half of each database that is the furthest from the median of all 2n elements. Once we have 2 elements remaining (1 from each database), the average of the two numbers remaining will be the median of all 2n elements.

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.

The algorithm is the exact same, just with a 2 multiplied against the "front of B" in MergeCount

Algorithm: CountSort
Input : A list A of n comparable items.
Output: A sorted array and the number of inversions.
if $|A| = 1$ **then return** $(A, 0)$
$(A_1, c_1) :=$ CountSort(Front-half of A)
$(A_2, c_2) :=$ CountSort(Back-half of A)
$(A, c) :=$MergeCount(A_1, A_2)
return $(A, c + c_1 + c_2)$

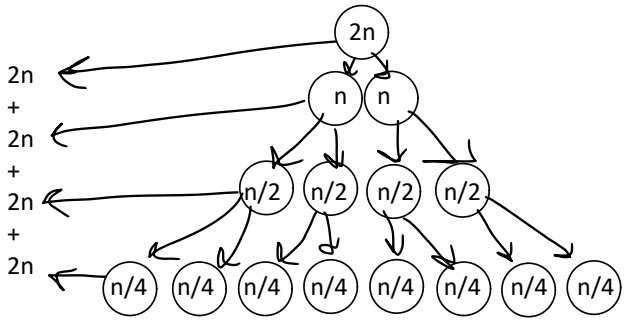
Algorithm: MergeCount
Input : Two lists of comparable items: A and B.
Output: A merged list and the count of inversions.
Initialize S to an empty list and $c := 0$.
while *either A or B is not empty* **do**
  Pop and append min{front of A, front of B} to S.
  if *Appended item is from B* **then**
    $c := c + |A|$.
  end
end
return (S, c)

Replace with (2 * front of B)

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

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

2n*log(n) = O(nlogn)



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

Assume that the algorithm given in lecture is correct. The only change made was multiplying the *j*th element by 2, in pursuit of finding a *significant inversion*. Now the algorithm checks to see if the *i*th element is greater than 2*(*j*th element).

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.

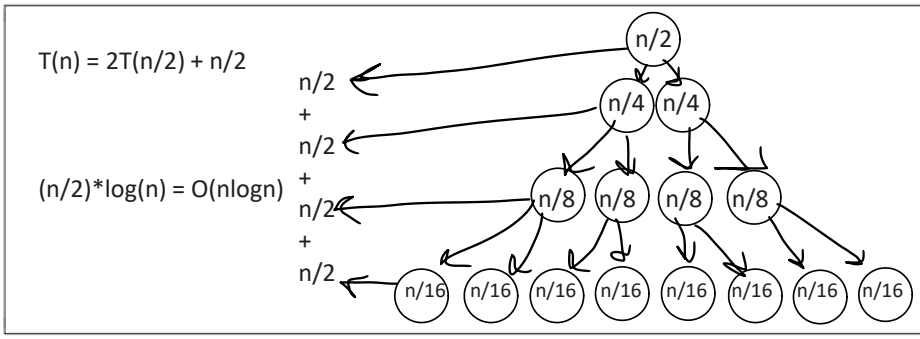
   (a) Give an algorithm to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Arbitrarily pair the cards into duos.
Plug each duo into the equivalence tester.
If |cards|==2, return yes.
If there are no alarms (assume one would go off if two cards corresponded to the same account), return no.
If there are any alarms, separate the cards in half (splitting up each pair into a different half) and make a recursive call on each.

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

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

(n/2)*log(n) = O(nlogn)



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

Assume that the base case (having n/2 piles) returns the correct answer of "yes". If an alarm continues to go off at each recursive call,

5. 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 ranking.

   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 ranking. 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
```