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

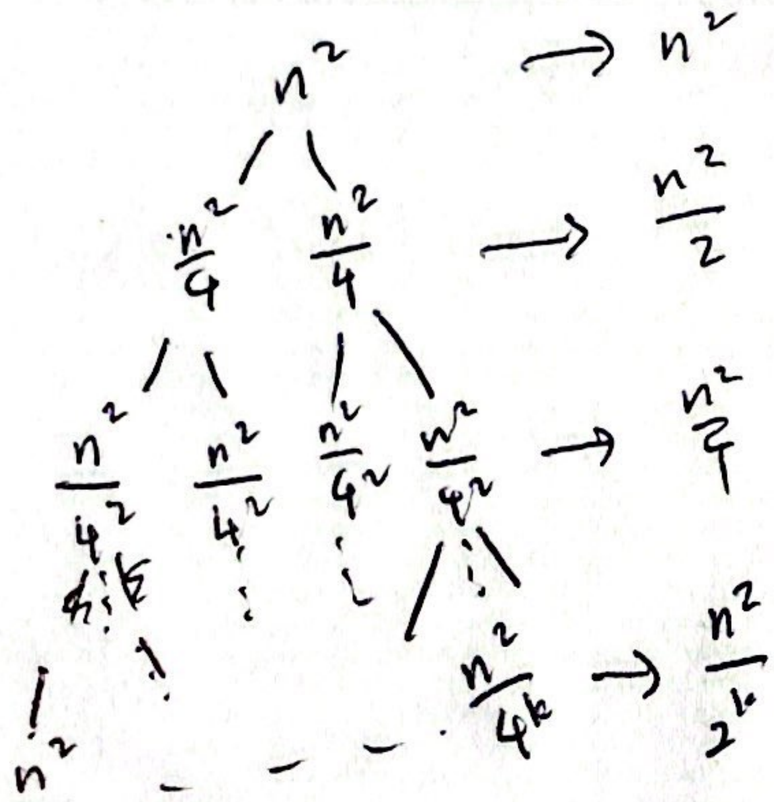
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Ethan Yan Wisc id: 9084049137

Divide and Conquer

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

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



$1 = \frac{n^2}{2^k}$

$2^k = n^2$

$k = 2\log_2(n) =$

$2^k T(\frac{n^2}{2^k}) + kn^2$

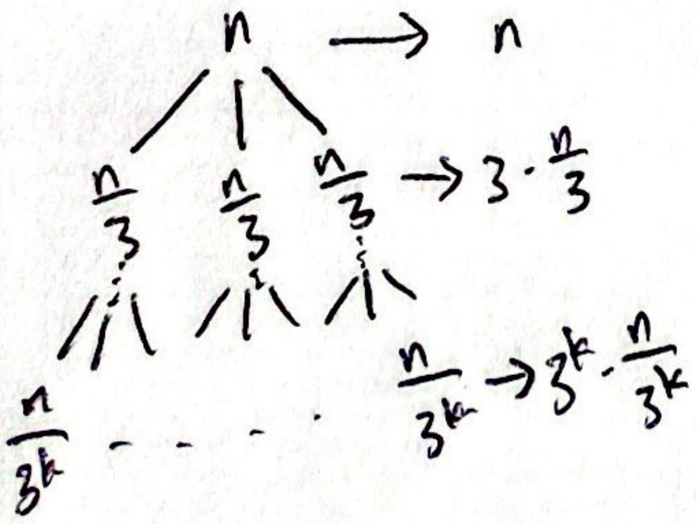
$n^2 + 2n^2\log_2(n)$

$= 2n^2(1-\frac{1}{2^{\log_2 n}})$

$= 2n^2(1-\frac{1}{n})$

$= 2n^2 - 2n \in O(n^2)$

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



$\frac{n}{3^k} = 1$

$3^k = n$

$k = \log_3 n$

$= n \lg n + n$

$= O(n \cdot \lg n)$

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.

Let A & B be both databases.

~~Find value $n/2$ in both database, If $A[\frac{n}{2}] < B[\frac{n}{2}]$, then~~

FindMedian (size, a, b)

| If size=1, return $\min(A[a+\frac{size}{2}], B[b+\frac{size}{2}])$
| If $A[a+\frac{size}{2}] < B[b+\frac{size}{2}]$, then FindMedian $(\frac{size}{2}, a+\frac{size}{2}, b)$
| If $A[a+\frac{size}{2}] > B[b+\frac{size}{2}]$, then FindMedian $(\frac{size}{2}, a, b+\frac{size}{2})$

end

Algorithm

| FindMedian (n, 0, 0)

end

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

$T(n) \leq T(\frac{n}{2}) + c$, $T(1) \leq c$

$k \cdot c = c \log n \in O(\log n)$

k layers. $\begin{cases} \frac{n}{2} \\ \vdots \\ \frac{n}{2^k} \end{cases}$

$\frac{n}{2^k} = 1$

$k = \log n$.

k layers; c work per layer.

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

Soundness

Base case: $n=1$,

Means only 1 value in each database

FindMedian will return minimum of these values, which will be the median regardless since there are only 2 values.

Using strong induction: assume $n=k$ holds.

Show $n=k+1$ holds. When $n=size=k+1$ enters FindMedian, we know $k+1>1$, so we will do either FindMedian($\frac{k+1}{2}$, a, b+$\frac{k+1}{2}$) or ($\frac{k+1}{2}$, a, b+$\frac{k+1}{2}$) which will return the median since $1<\frac{k+1}{2}<k$ by the inductive hypothesis.

Correctness:

In each recursive call, we pass in $\lfloor \frac{size}{2} \rfloor$ given size is initially n, the next recursive call would be $\frac{size-1}{2}$ if odd or $\frac{size}{2}$ if even which is less than size, thus it makes its way down approaching 1.

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 CountSort

if |A|=1 then return (A,0)

(A_1, c_1) = Countsort (1st half of A)

(A_2, c_2) = countsort (2nd half of A)

(A, c) = MergeCount(A_1, A_2)

return $(A, c+c_1+c_2)$

end.

Algorithm MergeCount

Initialize S to an empty list and c=0.

while either A or B is not empty:

pop and append min{front A, front B} to S.

if popped item is from B & more than twice of B then:

c := c+|A|

end

end

return (S, c)

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

$T(n) \leq 2 \cdot T(\frac{n}{2}) + cn$; $T(1) \leq c$.

$T(n) \leq 2T(\frac{n}{2}) + cn$

$\leq 2(2T(\frac{n}{4}) + c\frac{n}{2}) + cn$ $\quad | = \frac{n}{2^k}$

$\vdots$

$\leq 2^k T(\frac{n}{2^k}) + kcn$ $\quad 2^k = n$

$= nT(1) + cn\log(n)$ $\quad k = \log_2 n$

$= c_1 + cn\log n$

$= O(n \log n)$.

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

Soundness:

Base case: $n=1$, no inversions, countsort returns 0 which is correct.

IH: when $n=k$, # of inversions returned is correct.

Strong induction: for $n=k+1$, we do countsort of 1st & 2nd half of A, which has size $1 < \frac{k+1}{2} < k$, so by IH, both halfs would return correct # of inversions —— ① & ②

~~Find~~ Final MergeCount for 1st & 2nd of A would iteratively go through each element & count ~~sth~~ significant inversions for this merge.. —— ③

Final count will hold as we sum all 3 counts from ①, ②, ③

Correctness:

In each recursive call, we pass in $\frac{size}{2}$ into countsort, given size is initially n, next recursive call would be less than n, it will be $\frac{n}{2}$, it makes way down to 1.

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.

CountEquivalence

c = null

If size = 1, return & ~~False~~ True.

If size = 2, ~~return~~

If S_1 & S_2 are equivalent, return S_1.

CountEquivalence (1st half of S)

if c != null:

count = 0

loop through S, for match(S[i], c), count++

if count > $\frac{size}{2}$: return True.

c = countEquivalence (2nd half of S)

if c != null:

count = 0

loop through S, for match(S[i] c), count++, if count > $\frac{size}{2}$; return True.

return False

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

$T(n) \le 2 \cdot T(\frac{n}{2}) + cn$; $T(1) \le c$

$T(n) \le 2T(\frac{n}{2}) + cn$

$\le 2(2T(\frac{n}{4}) + c\frac{n}{2}) + cn$

$\vdots$

$\le 2^k T(\frac{n}{2^k}) + kcn$

$= nT(1) + cn\lg(n) = cn + cn\lg n$

$= O(n \log n)$.

$k = \frac{n}{2}k$

$2^k = n$

$k = \log_2 n$

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

Soundness:

Base case: $n=1$, bfd true since only 1 card, matches itself

~~$n=2$, f~~

IH: ~~$n=k$ holds~~ $1 \le n \le k$ holds.

Strong induction: CountEquivalence recurrence calls ~~k~~ with size of array $\frac{k+1}{2}$ which is $1 < \frac{k+1}{2} < k$, so by IH, it would return card match if there is a match.

If there is no match in either half, False is returned, if there is a match, card matched would then be compared with every card in cards list to see if it matches $> \frac{n}{2}$ cards and return True if so.

Thus, ~~$n=k$~~ $n=k+1$ holds.

Correctness:

In each recursive call, we pass $\frac{size}{2}$ into countequivalence since $\frac{size}{2}$ works its way down to 1 & will terminate.

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
```