CSCI 3104: Algorithms
Spring 2022
Recitation #8 - Divide & Conquer

SOLUTIONS

Midterm TA reviews. Please take take a few minutes to fill out TA FCQs! They're very helpful for us TAs. You can find the link in your email under the subject line "Computer Science Midterm TA FCQ's."

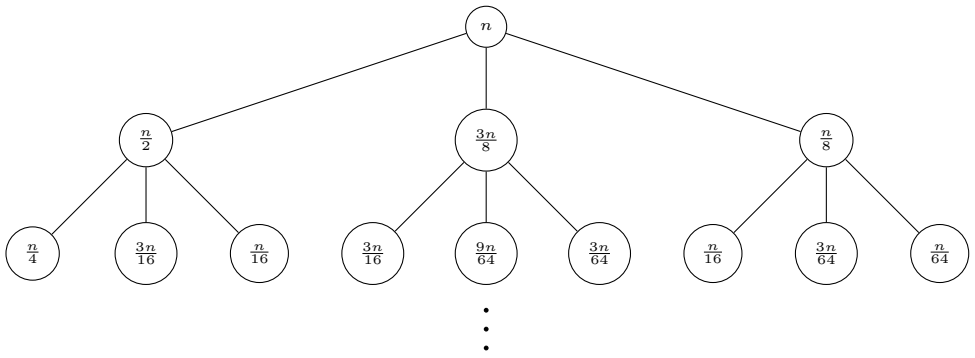
Problem 1

Say we have an algorithm which divides the input of size n into three sets of sizes $n/2$, $3n/8$, and $n/8$, with base cases when $n \leq 8$.

a. Write down the recurrence for the runtime of the algorithm.

$$T(n) = \begin{cases} \Theta(1) & n \leq 8 \\ T(n/2) + T(3n/8) + T(n/8) & n > 8 \end{cases}$$

b. Draw the first few levels of the recursion tree, labelling each vertex with its input size.



c. Is this a balanced or unbalanced recursion?

This tree is balanced, since each branch is a constant fraction of the input size. In an unbalanced tree, we remove a fixed number of elements at each recursion, rather than a fraction.

d. Give an upper bound on the runtime for our algorithm, as tight as you can.

We compute an upper bound as follows:

(a) *Compute the amount of work done at each level.* We write out the total work at several levels, noting that the work is $O(m)$ for input of size m:

$$n = n$$
$$n/2 + 3n/8 + 5n/8 = n$$
$$(n/4 + 3n/16 + n/16) + (3n/16 + 9n/64 + 3n/64) + (n/16 + 3n/64 + n/64) = n$$
$$\vdots$$

We note the pattern that each row requires total work n (as long as no branches have reached a base case). Thus, we do at most $O(n)$ work in each layer.

(b) *Compute the longest path.* We upper bound the number of possible layers by the depth of the deepest root-base case path in the tree. Let k denote the depth of the longest path. Since the largest recursion at each level is of size $n/2$, the path where we always take $n/2$ is the longest path, we stop when $n/(2^k) \leq 8$. Thus, the path must terminate by depth $\log_2(n) - 3$.

(c) *Upper-bound the runtime.* We upper bound the runtime by assuming that we reach no leaves (base cases) before the longest path terminates. Thus, our runtime is

$$T(n) \leq n\,(\log_2(n) - 3) \in O(n \log n)$$

(*Note:* we could probably achieve a better runtime bound, but this is good enough.)

Problem 2

In merge-sort, we sort a list by dividing it into two halves, sorting the halves recursively, then merging them together. What's so special about breaking things into two lists? Why not three? you could think about what we'll call 3-Mergesort, where you split the list into three lists of size roughly $n/3$, and sort the three sub-lists recursively, then merge them. 3-Mergesort works as follows, assuming n is always nicely divisible (to leave out floors):

Algorithm 1 3-Mergesort

```
1: procedure 3-MERGESORT(Integer array L)
2:     n ← |L|
3:     A ← 3-MERGESORT(L[0 : n/3 − 1])
4:     B ← 3-MERGESORT(L[n/3 : 2n/3 − 1])
5:     B ← 3-MERGESORT(L[2n/3 : n])
6: return 3-MERGE(A, B, C)
```

a. Write pseudocode for 3-MERGE. It should take as input three sorted lists and output them as a single sorted list containing the elements of all three lists.

Algorithm 2 3-Merge

```
1: procedure 3-MERGE(Integer arrays A,B,C)
2:     L = ∅
3:     while A, B, or C is nonempty do
4:         a ← A[0], b ← B[0], c ← C[0]
5:         x = min(a, b, c), X = array containing x
6:         remove x from X, append x to L
       return L
```

b. Give the recurrence relation of 3-MERGESORT, and give an asymptotic bound. Is our new sorting algorithm better than regular MERGESORT?

$$T(n) = 3T(n/3) + O(n)$$

We do $O(n)$ merge work on our arrays, and have to recursively sort three arrays of size $n/3$.

We unroll $T(n)$:

$$\begin{aligned} T(n) &= 3T(n/3) + O(n) \\ &= 9T(n/9) + 3O(n)/3 + O(n) \\ &= 27T(n/27) + O(n) + O(n) + O(n), \end{aligned}$$

and observe that our size is cut in three at each recursion. Thus, we reach a base case on layer k when $n/(3^k) \leq 1$, or $\log_3 n \leq k$. Thus, we recurse for depth $\log_3 n$ and do $O(n)$ work at each level, giving total runtime upper bound $O(n \log n)$.

This is the same asymptotic bound as regular MERGESORT, meaning as the input size increases we don't do significantly better using 3-MERGESORT than we would using regular MERGESORT.

c. Is the recurrence tree balanced or unbalanced?

This tree is balanced, since each branch is a constant fraction of the input size. In an unbalanced tree, we remove a fixed number of elements at each recursion, rather than a fraction.

d. How might we implement (deterministic) 3-QUICKSORT? Do you think this be an improvement?

(You don't need to give a formal statement of the algorithm, just give a description.)

We can select 2 pivots a and b from our input (say the first and second elements), and assume $a < b$. We write a new PARTITION function which creates 3 segments of our input list, in order: elements smaller than a, elements between a and b, and those larger b. This could be implemented in $O(n)$ time by running a pointer through the list twice: once to order all elements with regard to a and a second time to order all elements greater than a with regard to b. We can then sort each of the three sections recursively to give a fully sorted list.

This isn't really much better than regular QUICKSORT with regard to asymptotics, particularly for the deterministic case. In the deterministic case, the worst case is identical: we pick $a = b$ as the minimum element in our array. We also have to make two passes through our array to set the pointers, as opposed to one pass in regular QUICKSORT.