CS 256: Algorithm Design and Analysis

Problem Set 4 (due 3/19/2025)

Instructor: Sam McCauley

Problem 1. Use any of the methods discussed in class (unfolding recurrences, recursion trees, master theorem, guess and check etc.) to solve each of the following recurrences. Give as tight a Big Oh bound as possible: in other words, give a big Θ bound. In all cases, assume that $T(1) = 1$. You must justify (at a high level) your answer in each case—e.g., if using the recursion-tree method, draw the first few levels of the tree and describe which of the three categories does the recurrence lie in, and why that leads to the time bound. You do not need to verify by induction (unless you are using the guess and check method in which case you do need a proof). You may use a latex to draw figures (as done below), or attach a photo/scan of a neatly hand-drawn figure. The first part is solved to guide your approach.

(a) $T(n) = 2T(n/2) + n^2$

Solution. The recursion tree for this recurrence is given in Figure 1. Notice that the cost at each level is decreasing by at least a constant factor. The total cost at the root is n^2, one level down is $n^2/2$, and two levels down is $n^4/4$. In particular we get the following series: $T(n) = n^2(1 + 1/2 + 1/4 + \ldots) = \Theta(n^2)$. Summing, we obtain a total cost of $2n^2 = \Theta(n^2)$. □

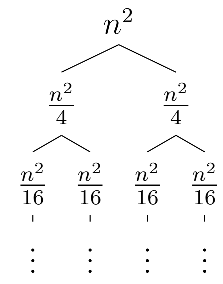

Figure 1: Recursion Tree for Problem 1 (a)

(b) $T(n) = 2T(n/4) + \sqrt{n}$

(c) $T(n) = 3T(n/3) + n^2$

(d) $T(n) = 2T(n/2) + n \log n$

Solution. □

Problem 2. Suppose you are choosing between the following three algorithms:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

2. Algorithm B solves problems of size n by recursively solving two subproblems of size $n - 1$ and then combining the solutions in constant time.

3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in asymptotic notation) and which would you choose?

Solution.

□

Problem 3. You're running an internet poll on a popular website. Each user on the website is only allowed to vote once in the poll. Of course, there's an obvious problem: some people have multiple user accounts on the website.

You want to rule out the worst-case scenario: is there a single person who controls the *majority* of the user accounts on the site?

You receive an array of usernames $U[1 \dots n]$. You want to determine if a single person controls $> n/2$ of the usernames. You don't know which person controls a given username; however you have access to two subroutines:

- SAMEPERSON(i,j) takes two integers i and j and returns whether or not $U[i]$ and $U[j]$ are controlled by the same person, in $O(1)$ time.[1]
- COUNTOCCURRENCES(i,ℓ,r) returns the number of usernames in $U[\ell, \ell+1, \dots, r]$ controlled by the same person that controls $U[i]$, in $O(r - \ell + 1)$ time.[2]

Notice that we can run COUNTOCCURRENCES(i,1,n) for all $i = 1, \dots n$ and learn exactly how many accounts each person controls, in $O(n^2)$ time.

Instead, you are to design a divide and conquer algorithm that determines if a single person controls $> n/2$ of the usernames, in $O(n \log n)$ time.[3] To avoid edge cases, assume n has a nice form, e.g., a power of 2. You must prove that your algorithm is correct (under the assumption that SAMEPERSON and COUNTOCCURRENCES are correct). State and solve the recurrence for the running time of your algorithm.

Solution. □

[1] Let's say this runs some simple analytics on the metadata we've stored for the two users.

[2] This is easy to implement—just call SAMEPERSON(i,j) for $j = \ell, \dots, r$. Having this as a subroutine, rather than a whole loop, may make your algorithm simpler.

[3] There are ways to solve this problem without using divide and conquer, but I am asking you for a divide and conquer solution so that you can get practice.

Problem 4. Suppose we are given two sorted arrays $A[1 \ldots n]$ and $B[1 \ldots n]$. Assume that the arrays do not contain duplicate elements. Describe a divide and conquer algorithm to find the median of $A \cup B$ in $O(\log n)$ time. The median of sorted array A of size n is the middle element (at index $(n+1)/2$) if n is odd; and is the average of the two middle elements (at indices $n/2$ and $(n+1)/2$) if n is even. Remember to justify the correctness of your algorithm and state and solve its running time recurrence.

Hint. Be careful of corner cases. One useful exercise to check your work is: the median consists of either one element, or the average of two elements. Can you be sure that your algorithm will use these elements in its final solution?

Solution. □

Problem 5. Consider the following funky recursive sorting algorithm called FUNKY-SORT. The algorithm is described using a 1-indexed array.

```
FUNKY-SORT($A[1, \ldots, n]$):
    if $n = 2$ and $A[1] > A[2]$:
        swap $A[1] \leftrightarrow A[2]$
    else if $n > 2$:
        $m = \lceil 2n/3 \rceil$
        FUNKY-SORT($A[1, \ldots, m]$)
        FUNKY-SORT($A[n - m + 1, \ldots, n]$)
        FUNKY-SORT($A[1, \ldots, m]$)
```

(a) State and solve the recurrence for the running time of FUNKY-SORT.

Solution. □

(b) Does FUNKY-SORT actually sort the array? Let's prove this in several cases:

(a) Consider an item x that is one of the $n - m$ largest elements of the array. Show that it is placed into the correct slot after the *second* call to FUNKY-SORT.

> **Hint.** What can we say about where x is after the first call to FUNKY-SORT?

Solution. □

(b) Consider an item x that is *not* one of the $n - m$ largest (it is one of the m smallest items). Show that it is placed into the correct slot after the third call to FUNKY-SORT.

> **Hint.** Use your answer above. Since the $n - m$ items are all sorted after the second call to FUNKY-SORT, where must x be after the second call? Why does that mean that it must then be sorted after the third call?

Solution. □

Problem 6. (**Extra Credit (15 pts)**) The company UPS has noticed that left turns are extremely costly—trucks must idle for a long time looking for an opportunity to turn left. In fact, when determining routes for drivers, they eliminate almost all left turns from the route. This saves the company over 300 million dollars per year. See this url for one article on this policy: https://www.cnn.com/2017/02/16/world/ups-trucks-no-left-turns/index.html

Let's apply the same logic to shortest path. Define two consecutive edges along a path $(v_1, v_2), (v_2, v_3)$ to be a **left turn** if:

- v_3 is immediately before v_1 in the adjacency list of v_2, or
- v_3 is the last vertex in the adjacency list of v_2 and v_1 is the first.

Given an undirected graph G where all edges have a positive weight, and two nodes s and t, let the **UPS shortest path** be the shortest path from s to t that does not take a left turn.

Design an efficient algorithm to compute the UPS shortest path. Analyze its running time and briefly explain its correctness. (Note that this is a Dijkstra's question—divide and conquer is unlikely to be helpful here.)

Solution. □