CS 3510 Algorithms Last updated: August 28, 2023

Lecture 2: Divide and Conquer

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

1 Introduction

There's no single method to design all algorithms, but there are different algorithmic ideas which are applicable to many problems. For instance, we'll be talking about Divide and Conquer in the upcoming lectures, which will be useful for many problems. Though it won't work for all problems.

Essentially divide and conquer works how it sounds, you'll break your problem into smaller sub-problems and then combine the solutions to the sub-problem to get a solution for the original problem. The hope is that since you have a smaller problem, it will be easier to solve. For instance with sorting, you could break your big problem into multiple sub-problems by splitting the list you're trying to sort into two smaller lists. Obviously it's easier to sort a list of length one or two, than a list of length ten or twenty.

2 MergeSort

MergeSort is a sorting algorithm based on the idea of divide and conquer. One key property of MergeSort is that the runtime of MergeSort is $\mathcal{O}(n \log n)$, where n is the number of elements in a list. [1]

Algorithm 1 MergeSort (Non-rigorous)

1: Break original list into $a[0, \ldots, \frac{n}{2} - 1]$ and $a[\frac{n}{2}, \ldots, n - 1]$;
2: Recursively solve each;
3: Combine sorted arrays into a sorted solution.

Above is an imprecise definition of the algorithm. You can consider this as an initial sketch of the solution to the problem. How do we combine though? Consider the resulting lists from the sub problems, and imagine you have two pointers into each list. You take the smallest number pointed to by one of the pointers, and add that to your resulting array, then you adjust the pointer to move it past the element.

Above we see the pseudocode for the algorithm. However creating an algorithm is not the only step, we need to show that the algorithm is correct and understand it's performance.

2.1 Correctness

Note about correctness on homeworks and exams: depending on the complexity of the algorithm, correctness may not be obvious, but please argue something about correctness

[1] For simplicity, assume that n is even. Technically, we should be putting floors/ceilings on the proof to bound the runtime, but the general concept (and, importantly, the runtime) is the same.

Algorithm 2 MergeSort

```
1: procedure MergeSort(A[1...n])
2:     if n < 1 then
3:         return A
4:     else
5:         return Merge(MergeSort(A[1...⌊n/2⌋]), MergeSort(A[⌊n/2⌋ + 1...n]))
6:     end if
7: end procedure
8: procedure Merge(A[1...k], B[1...l])
9:     if k == 0 then
10:        return B[1...l]
11:    else if l == 0 then
12:        return A[1...k]
13:    end if
14:    if A[1] < B[1] then
15:        return A[1] ∧ Merge(A[2...l], B[1...k])
16:    else
17:        return B[1] ∧ Merge(A[1...l], B[2...k])
18:    end if
19: end procedure
```

in homeworks and exams (at least include a couple of sentences).

Obviously this algorithm is correct though, but we could also rigourusly prove it's correctness using induction. Our induction hypothesis could be the algorithm returns the correct solution on input size i. This is left as an exercise for your curiosity.

2.2 Analysis

The goal is to find the running time ($T(n)$) which is the worst-case running time on an input of size n. We are not finding the exact function of $T(n)$, but rather the $\mathcal{O}$-notation of the family of functions that $T(n)$ belongs to. The claim is that $T(n) = \mathcal{O}(n \log n)$.

The way we are going to analyze this algorithm is to find a recurrence relation for the runtime of the algorithm. In this case, we have

$$T(n) = 2T(\frac{n}{2}) + c \cdot n$$

where each of the two $T(\frac{n}{2})$ comes from the running time of solving the two subproblems, and $c \cdot n$ comes from the merge operation. [2] [3] But can we "solve" this recurrence relation and find a closed form for $T(n)$? We will solve this recurrence relation in two methods. The first is:

[2] We will again assume that n is a power of 2 for simplicity.

[3] We will also assume that c, the time needed to merge each element, is 1. This is also dependent on the implementation of the algorithm, but importantly it is a constant.

2.2.1 The Tree Method

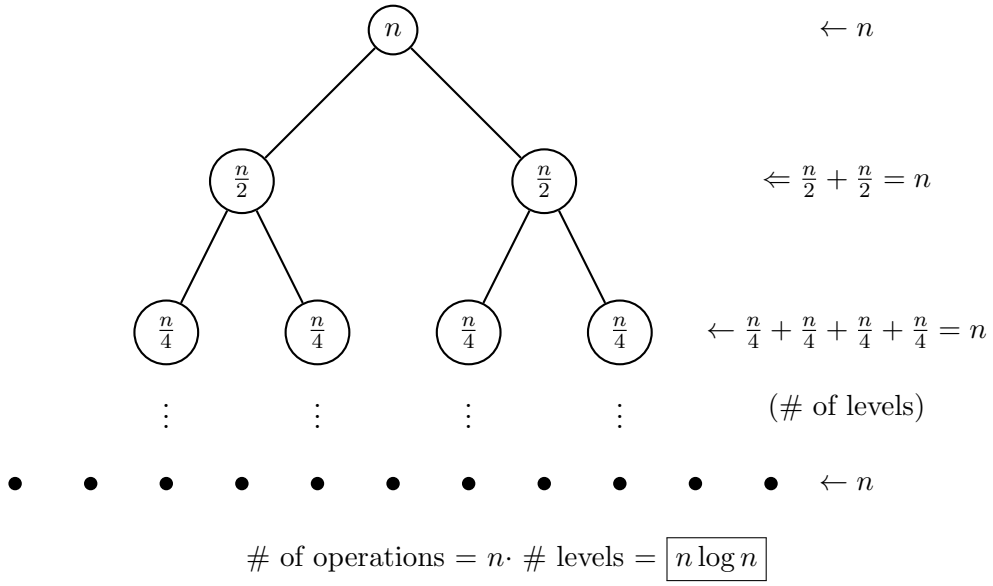

Figure 1: Tree Method of finding runtime

The key idea is that we count the number of operations on each level. In the first (top) level, we perform the merge operation once on a list of length n; thus, n operations are performed. In the second level, we perform the merge operation twice on lists of length $n/2$; thus, still n operations are performed. In general, on the kth level, we perform the merge operation 2^{k-1} times on lists of length $\frac{n}{2^{k-1}}$, which in total produce n operations. In fact, exactly n operations are performed on each level, and with $\log n$ levels, $n \log n$ operations are done.

With trees, however, we have to be careful in how we count up the number of operations. While this is technically rigorous, is there a better method to find the runtime of some divide and conquer algorithms? [4]

2.2.2 Guess and Verify by Induction

In an induction statement you need to carefully guess at the runtime. For MergeSort, we could guess that the recurrence is $T(i) \leq C \cdot i \log i$. Then you'll need to do two parts to prove correctness: the base case and the inductive step.

To show the base case, we have that $i = 1$. Here, we have that $T(1) = 0$.

The next step is the inductive step. Assuming the inductive hypothesis is true for $k \in \{1, \ldots, \text{ı}\}$, we need to prove that the induction hypothesis is true for $i + 1$:

$$T(i+1) = 2T(\frac{i+1}{2}) + (i+1)$$

$$\leq 2 \cdot c \cdot \frac{i+1}{2} \log(\frac{i+1}{2}) + (i+1)$$

[4] Some recurrence relations are actually very complicated, and no general solutions are known.

$$= c(i+1)\log(i+1) - c(i+1)\log 2 + (i+1)$$
$$\leq c(i+1)\log(i+1)$$

The first inequality follows by the inductive hypothesis on $\frac{i+1}{2}$; the second equality, by properties of the logarithm; and the third inequality is true when $c \geq 2$. Thus, the inductive hypothesis is true for $i+1$, and by induction we have shown that the inductive hypothesis is always true. It follows that $T(n) = \mathcal{O}(n \log n)$, and we are done.

3 Master Theorem

It turns out though, that for many simple recurrences we can use the "Master's Theorem" to find the overall equation. The Master's Theorem works for recurrences of the form $T(n) = aT(\frac{n}{b}) + \mathcal{O}(n^d)$. [5]

$$T(n) = \begin{cases} \mathcal{O}(n^d) & d > \log_b a \\ \mathcal{O}(n^d \log n) & d = \log_b a \\ \mathcal{O}(n^{\log_b a}) & d < \log_b a \end{cases} \qquad \text{(Master Theorem)}$$

[5] If you're interested in how to prove the Master Theorem which may be useful, you can find the proof in DPV or CLRS.