CS 161 (Stanford, Winter 2026) Lecture 3

Adapted From Virginia Williams' lecture notes. Additional credits: Albert Chen, Juliana Cook, Ofir Geri, Sam Kim, Gregory Valiant, Aviad Rubinstein, Nima Anari.
Please direct all typos and mistakes to Moses Charikar and Ellen Vitercik.

Solving Recurrences and the Selection Problem

1 Introduction

Today we will continue to talk about divide and conquer, and go into detail on how to solve recurrences.

Recall that divide and conquer algorithms divide up a problem into several subproblems that are the smaller instances of the same problem, solve those problems recursively, and combine the solutions to the subproblems into a solution for the original problem. When a subproblem size is small enough, the subproblem is solved straightforwardly. In the past lectures, we have seen two examples of divide-and-conquer algorithms: MergeSort and Karatsuba's algorithm for integer multiplication.

The running time of divide and conquer algorithms can be naturally expressed in terms of the running time of smaller inputs. Today we will show two techniques for solving these recurrences. The first is called the *master method* to solve these recurrences. This method can only be used when the size of all the subproblems is the same (as was the case in the examples). We will also see a surprising algorithm that does not fall into this category, and how to analyze its running time using another method, the *substitution method*.

2 Recurrences

Stated more technically, a divide and conquer algorithm takes an input of size n and does some operations all running in $O(f(n))$ time for some f and runs itself recursively on $k \geq 1$ instances of size $n_1, n_2, ..., n_k$, where $n_i < n$ for all i. To talk about what the runtime of such an algorithm is, we can write a runtime **recurrence**. Recurrences are functions defined in terms of themselves with smaller arguments, as well as one or more base cases. We can define a recurrence more formally as follows:

Let $T(n)$ be the worst-case runtime on instances of size n. If we have k recursive calls on a given step (of sizes n_i) and each step takes time $O(f(n))$, then we can write the runtime as $T(n) \leq c \cdot f(n) + \sum_{i=1}^{k} T(n_i)$ for some constant c, where our base case is $T(c') \leq O(1)$.

Now let's try finding recurrences for some of the divide-and-conquer algorithms we have seen.

2.1 Integer Multiplication

Recall the integer multiplication problem, where we are given two n-digit integers x and y and output the product of the two numbers. The long multiplication/grade school algorithm runs in $O(n^2)$ time. In lecture 1 we saw two divide-and-conquer algorithms for solving this problem. In both of them, we divided each of x and y into two $(n/2)$-digit numbers in the following way: $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$. Then we compute $xy = ac \cdot 10^n + 10^{\frac{n}{2}}(ad + bc) + bd$.

In the first algorithm, which we call Mult1, we simply computed the four products ac, ad, bc, bd. Karatsuba found that since we only need the sum of ad and bc, we can save one multiplication operation by noting that $ad + bc = (a + b)(c + d) - ac - bd$.

Algorithm 1: Mult1(x, y)

Split x and y into $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$
$z_1 = \text{Mult1}(a, c)$
$z_2 = \text{Mult1}(a, d)$
$z_3 = \text{Mult1}(b, c)$
$z_4 = \text{Mult1}(b, d)$
return $z_1 \cdot 10^n + 10^{\frac{n}{2}}(z_2 + z_3) + z_4$

Algorithm 2: Karatsuba(x, y)

Split $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$
$z_1 = \text{Karatsuba}(a, c)$
$z_2 = \text{Karatsuba}(b, d)$
$z_3 = \text{Karatsuba}(a + b, c + d)$
$z_4 = z_3 - z_1 - z_2$
return $z_1 \cdot 10^n + z_4 \cdot 10^{\frac{n}{2}} + z_2$

We now express the running time of these two algorithms using recurrences. Adding two n digit integers is an $O(n)$ operation, since for each position we add at most three digits: the ith digit from each number and possibly a carry from the additions due to the $(i-1)$th digits.

Let $T_1(n)$ and $T_2(n)$ denote the worst-case runtime of Mult1 and Karatsuba, respectively, on inputs of size n. Then, the runtime of Mult1 can be written as the recurrence

$$T_1(n) = 4T_1\left(\frac{n}{2}\right) + O(n),$$

and Karatsuba's runtime can be written as the recurrence

$$T_2(n) = 3T_2\left(\frac{n}{2}\right) + O(n).$$

Note that the constant "hidden" in the $O(n)$ term in T_2 may be greater than in T_1, but for the asymptotic analysis of the running time, these constants are not important.

2.2 MergeSort

Consider the basic steps for algorithm MergeSort(A), where $|A| = n$.

1. If $|A| = 1$, return A.
2. Split A into A_1, A_2 of size $\frac{n}{2}$.
3. Run MergeSort(A_1) and MergeSort(A_2).
4. Merge(A_1, A_2)

Steps 2 and 4 each take time $O(n)$. In step 3, we are splitting the work up into two subproblems of size $\frac{n}{2}$. Therefore, we get the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

In the previous lecture, we saw that the running time of MergeSort is $O(n \log n)$. In this lecture, we will show how to derive this using the master method.

3 The Master Method

We now introduce a general method, called the *master method*, for solving recurrences where all the subproblems are of the same size. We assume that the input to the master method is a recurrence of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

In this recurrence, there are three constants:

- a is the number of subproblems that we create from one problem and must be an integer greater than or equal to 1.
- b is the factor by which the input size shrinks (it must hold that $b > 1$).
- d is the exponent of n in the time it takes to generate the subproblems and combine their solutions.

There is another constant "hidden" in the big-O notation. We will introduce it in the proof and see that it does not affect the result.

In addition, we need to specify the "base case" of the recurrence, that is, the runtime when the input gets small enough. For a sufficiently small n (say, when $n = 1$), the worst-case runtime of the algorithm is constant, namely, $T(n) = O(1)$.

We now state the *master theorem*, which is used to solve the recurrences.

Theorem 1 (Master Theorem)**.** *Let* $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ *be a recurrence where* $a \geq 1, b > 1$*. Then,*

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Remark 2. In some cases, the recurrence may involve subproblems of size $\lceil \frac{n}{b} \rceil$, $\lfloor \frac{n}{b} \rfloor$, or $\frac{n}{b} + 1$. The master theorem holds for these cases as well. However, we do not prove that here.

Before we turn to the proof of the master theorem, we show how it can be used to solve the recurrences we saw earlier.

- Mult1: $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$.

  The parameters are $a = 4, b = 2, d = 1$, so $a > b^d$, hence $T(n) = O(n^{\log_2 4}) = O(n^2)$.

- Karatsuba: $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$.

  The parameters are $a = 3, b = 2, d = 1$, so $a > b^d$, hence $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

- MergeSort: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$.

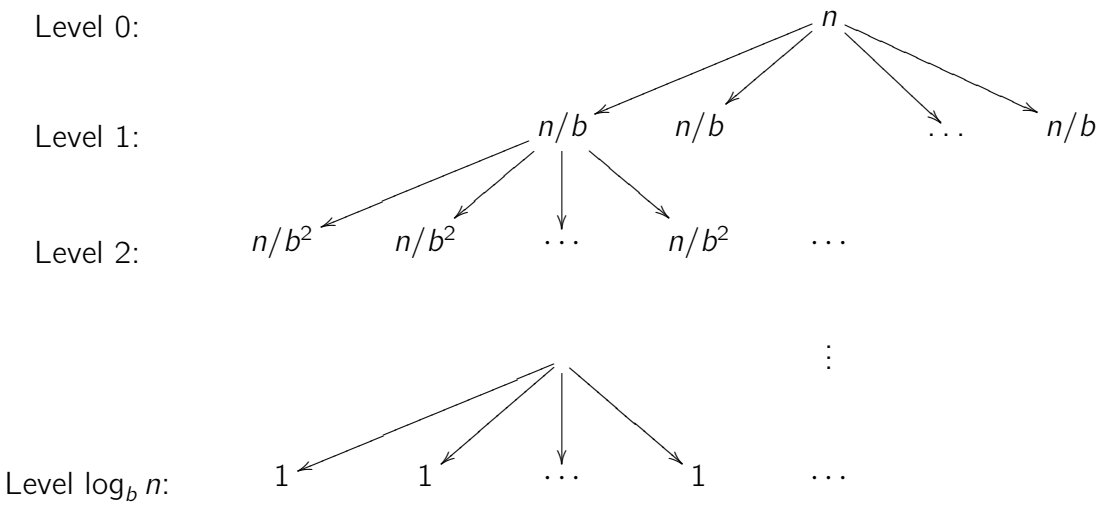
  The parameters are $a = 2, b = 2, d = 1$, so $a = b^d$, hence $T(n) = O(n \log n)$.

- Another example: $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$.

  The parameters are $a = 2, b = 2, d = 2$, so $a < b^d$, hence $T(n) = O(n^2)$.

We see that for integer multiplication, Karatsuba is the clear winner!

Proof of the Master Theorem. Let $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ be the recurrence we solve using the master theorem. For simplicity, we assume that $T(1) = 1$ and that n is a power of b. From the definition of big-O, we know that there is a constant $c > 0$ such that for sufficiently large n, $T(n) \leq a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$. The proof of the master theorem will use the recursion tree in a similar way to our analysis of the running time of MergeSort.



The recursion tree drawn above has $\log_b n + 1$ level. We analyze the amount of work done at each level and then sum over all levels to get the total running time. Consider level j. At level j, there are a^j subproblems. Each of these subproblems is of size $\frac{n}{b^j}$, and will take time at most $c\left(\frac{n}{b^j}\right)^d$ to solve (this only considers the work done at level j and does not include the time it takes to solve the subsubproblems). We conclude that the total work done at level j is at most $a^j \cdot c\left(\frac{n}{b^j}\right)^d = cn^d\left(\frac{a}{b^d}\right)^j$.

Writing the running time this way shows us where the terms a and b^d come from: a is the branching factor and measures how the number of subproblems grows at each level, and b^d is the shrinkage in the work needed (per subproblem).

Summing over all levels, we get that the total running time is at most $cn^d \sum_{j=0}^{log_b n}\left(\frac{a}{b^d}\right)^j$. We now consider each of the three cases.

1. $a = b^d$. In this case, the amount of work done at each level is the same: cn^d. Since there are $\log_b n+1$ levels, the total running time is at most $(\log_b n+1)cn^d = O(n^d \log n)$.

2. $a < b^d$. In this case, $\frac{a}{b^d} < 1$, hence, $\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty}\left(\frac{a}{b^d}\right)^j = \frac{1}{1-\frac{a}{b^d}} = \frac{b^d}{b^d-a}$. Hence, the total running time is $cn^d \cdot \frac{b^d}{b^d-a} = O(n^d)$.

   Intuitively, in this case, the shrinkage in the work needed per subproblem is more significant, so the work done in the highest level "dominates" the other factors in the running time.

3. $a > b^d$. In this case, $\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j = \frac{\left(\frac{a}{b^d}\right)^{\log_b n+1}-1}{\frac{a}{b^d}-1}$. Since a, b, c, d are constants, we get that the total work done is $O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \cdot \frac{a^{\log_b n}}{b^{d\log_b n}}\right) = O\left(n^d \cdot \frac{n^{\log_b a}}{n^d}\right) = O(n^{\log_b a})$.

   Intuitively, here the branching factor is more significant, so the total work done at each level increases, and the leaves of the tree "dominate".

□

We conclude with a more general version of the master theorem.

Theorem 3 (Master Theorem - more general version)**.** *Let* $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ *be a recurrence where* $a \geq 1$, $b > 1$. *Then,*

- *If* $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ *for some constant* $\epsilon > 0$, $T(n) = \Theta\left(n^{\log_b(a)}\right)$.
- *If* $f(n) = \Theta\left(n^{\log_b(a)}\right)$, $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
- *If* $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$ *for some constant* $\epsilon > 0$ *and if* $af(n/b) \leq cf(n)$ *for some* $c < 1$ *and all sufficiently large* n, *then* $T(n) = \Theta(f(n))$.

4 The Substitution Method

Recurrence trees can get quite messy when attempting to solve complex recurrences. With the substitution method, we can guess what the runtime is, plug it into the recurrence and see if it works out.

Given a recurrence $T(n) \leq f(n) + \sum_{i=1}^{k} T(n_i)$, we can guess that the solution to the recurrence is

$$T(n) \leq \begin{cases} d \cdot g(n_0) \text{ if } n = n_0 \\ d \cdot g(n) \text{ if } n > n_0 \end{cases}$$

for some constants $d > 0$ and $n_0 \geq 1$ and a function $g(n)$. We are essentially guessing that $T(n) \leq O(g(n))$.

For our base case, we must show that you can pick some d such that $T(n_0) \leq d \cdot g(n_0)$. For example, this can follow from our standard assumption that $T(1) = 1$.

Next, we assume that our guess is correct for everything smaller than n, meaning $T(n') \leq d \cdot g(n')$ for all $n' < n$. Using the inductive hypothesis, we prove the guess for n. We must pick some d such that

$$f(n) + \sum_{i=1}^{k} d \cdot g(n_i) \leq d \cdot g(n), \text{whenever } n \geq n_0.$$

Typically the way this works is that you first try to prove the inductive step starting from the inductive hypothesis, and then from this obtain a condition that d needs to obey. Using this condition you try to figure out the base case, i.e., what n_0 should be.