Scribe: Juliana Cook (2015), V. Williams **Date:** April 6, 2016

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 a number of 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 simply solved in a straightforward manner. In past lectures we have seen two examples of divide and conquer algorithms, merge sort and the median-of-medians approach to the selection problem.

This naturally gives rise to runtime recurrences, which express the algorithm on runtimes of input size n in terms of the runtimes of smaller inputs.

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 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 MergeSort

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

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

Steps 1 and 3 each take time $O(n)$. Thus we know that $f(n) = n$ given the recurrence formula above. As we are splitting the work up into two evenly sized pieces we know that $k = 2$ and each $n_i = \frac{n}{2}$. Therefore, our final equation is

$$T(n) \leq c \cdot n + 2 \cdot T\left(\frac{n}{2}\right).$$

We note that we can replace c by 1 by, in a sense, changing units. We can analyze $T'(n) = 2T'(n/2) + n$, remembering that each cost 1 operation actually costs c. In the end we can just set $T(n) = cT'(n) = O(T'(n))$. Thus from now on we will remove constants as much as possible.

We will consider two more recurrences. The first one corresponds to a divide and conquer algorithm for finding the minimum number in an (unsorted) array A. Assume that A has size n which is a power of 2. We can make this assumption without loss of generality: Suppose that $2^{k-1} < n = 2^k - r$ for some $r < 2^{k-1}$;

then we can increase the size of A by r by adding r new entries at the end that are bigger than all integers in the array (say ∞). Then, the size of A can at most double, and now it is a power of 2, 2^k.

The algorithm FindMin(A) first checks if the array size n is 1. If so, it just returns the contents of $A[0]$. Otherwise it recursively finds $a = FindMin(A[0 : n/2 - 1])$ and $b = FindMin(A[n/2 : n - 1])$ and returns $\min\{a, b\}$.

The runtime recurrence is $T(n) \leq 2T(n)/2 + 1$ for all $n > 1$ and $T(1) = 1$. (Again, the 1s in this recurrence correspond to some machine dependent constant.)

The final recurrence we consider is $T(n) \leq T(7n/10 + 5) + T(n/5 + 1) + n$ for all $n > 5$ and $T(n) \leq 1$ for $n \leq 5$. This recurrence will come up when we introduce an algorithm in the next lecture.

3 Methods for Solving Recurrences

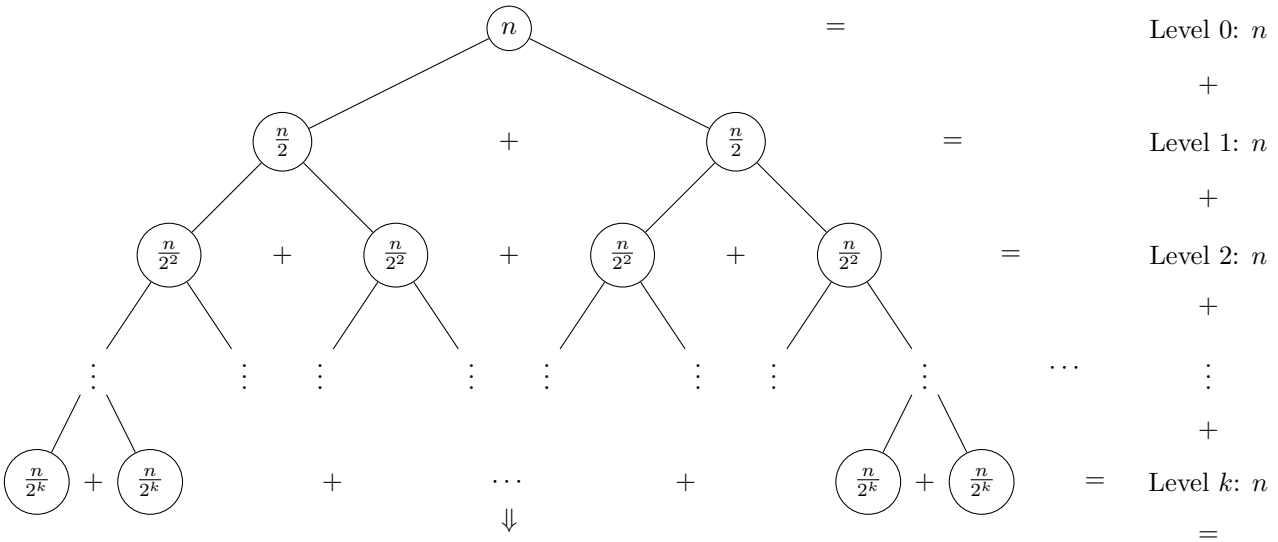
Once we have a recurrence for a function, we want to determine what the asymptotic bounds are (either Θ or O). There are three main methods for solving recurrences.

1. Recursion Tree
2. Substitution Method - guess runtime and check using induction
3. Master Theorem

3.1 Recursion Tree

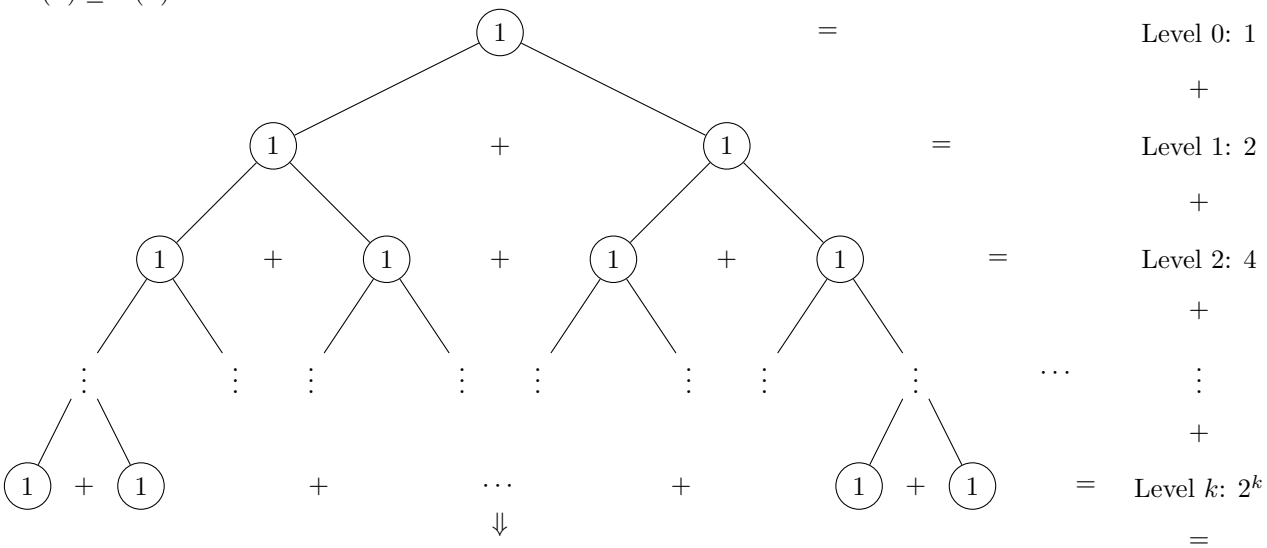
Recursion trees are a visual way to devise a good guess for the solution to a recurrence, which can then be formally proved using the substitution method. In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. Summing the costs within each level of the tree gives us the cost of each level of recursion. To determine the total cost we then sum the costs of all levels of the recursion.

Below is an example of a recursion tree for the MergeSort algorithm. At the root node we have n, representing the size of the original instance. Since we have two recursive calls at each step in which we divide the current size into two equally sized pieces, each node as two children that are each half the size of their parent.



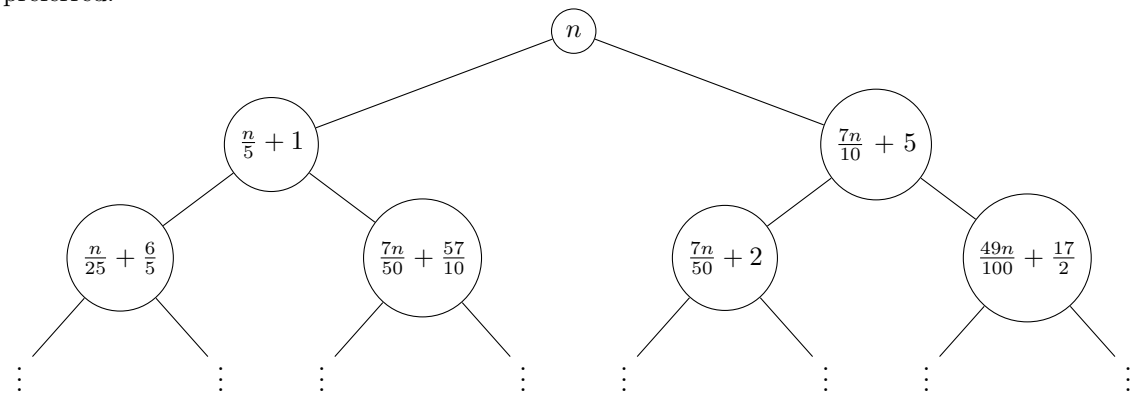
Recursion tree for Merge Sort. LaTEX credit Manuel Kirsch

If we analyzed the FindMin recurrence $T(n) \leq 2T(n/2) + 1$ with the recursion tree method we would get that $T(n) \leq O(n)$:



Total over levels $i \in \{0, \ldots, k = \log_2 n\}$: $\sum_{i=0}^{\log n} 2^i \leq 2^{\log_2(n)+1} - 1 \quad = \quad 2n - 1 \quad \Leftrightarrow \quad O(n)$

While recursion trees work quite well for understanding simpler recurrences, more complex recurrences start to get quite tricky. Below is a partial recurrence tree for the recurrence $T(n) \leq T(7n/10+5)+T(n/5+1)+n$. Note that at each stage the node is divided into uneven groups, which will result in some leaf nodes being at different levels than others. To get the sums in each level one needs to carefully calculate the different node sizes, and it can get quite messy. Thus for such recurrences a different solution method may be preferred.



Recursion tree for selection problem from next lecture.

We can nevertheless use the recursion tree method to generate a *guess* for what $T(n) \leq T(7n/10 + 5) + T(n/5 + 1) + n$ solves to. Suppose that we drop the $+5$ and $+1$ terms and analyze $T'(n) \leq T'(7n/10) + T'(n/5) + n$ instead. We might not get the right answer but we can guess that it will be something close to correct. With the recursion tree method we see that:

- In level 0, the algorithm performs n work.
- In level 1, the sum of runtimes is $7n/10 + n/5 = 9n/10$

- In level 2, the sum under the node with $7n/10$ is $9/10(7n/10)$ and under the node with $n/5$ it's $9/10(n/5)$. Altogether it's $(9/10)^2 n$.
- Inductively we can show that in level k the algorithm performs $(9/10)^k n$ work.
- Thus the total amount of work is $< n \sum_{i=0}^{\infty} (9/10)^i = 10n \leq O(n)$.

Thus we can guess that the answer is $O(n)$ for $T(n) \leq T(7n/10 + 5) + T(n/5 + 1) + n$. With the next method we will prove it by induction.

3.2 Substitution Method

As we saw above, recurrences trees can get quite messy when attempting to solve complex recurrences. With the substitution method we can guess what the runtime is, plug it in to the recurrence and see if it works out. Given a recurrence $T(n) \leq cf(n) + \sum_{i=1}^{k} T(n_i)$, 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 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 $T(n_0) \leq d \cdot g(n_0)$, assuming $T(1) = 1$ for instance (or whatever the base case of the recurrence is).

Next we assume that our guess is correct for everything smaller than n, meaning $T(n') \leq d \cdot g(n')$ for all $n' < n$. For our 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 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.

3.2.1 Solving $T(n) \leq T(7n/10 + 5) + T(n/5 + 1) + n$ using Substitution Method

Recall the recurrence $T(n) \leq cn + T\left(\frac{n}{5} + 1\right) + T\left(\frac{7n}{10} + 5\right)$ for $n > 5$ and $T(n) \leq 1$ for $n \leq 5$. From our recursion tree we make the guess:

$$T(n) \leq \begin{cases} dn_0 \text{ if } n = n_0 \\ d \cdot n \text{ if } n > n_0 \end{cases}$$

For our inductive hypothesis, we assume that our guess is correct for anything smaller than n, and we prove our guess for n. That is, pick some d such that for all $n \geq n_0$,

$$n + \sum_{i=1}^{k} dg(n_i) \leq d \cdot g(n).$$

$$n + d \cdot \left(\frac{n}{5} + 1\right) + d \cdot \left(\frac{7n}{10} + 5\right) \leq d \cdot n.$$

$$n \cdot \left[1 + \frac{9d}{10} + \frac{6d}{n}\right] \leq d \cdot n.$$

$$\left[1 + \frac{9d}{10} + \frac{6d}{n}\right] \leq d.$$

Replacing the n in the denominator with n_0 since we only care about $n \geq n_0$, we get that we are looking for d, n_0 that satisfy the base case $T(n_0) \leq dn_0$ and such that

$$\left(1 + \frac{9d}{10} + \frac{6d}{n_0}\right) \leq d$$

$$1 \leq d \cdot \left(\frac{1}{10} - \frac{6}{n_0}\right)$$

The above is true whenever $\frac{1}{10} - \frac{6}{n_0} > 0$ and

$$d \geq \frac{10}{1 - \frac{60}{n_0}}.$$

If we pick $n_0 = 61$ for instance we get that we can select any $d \geq 610$. Thus we consider the base case and figure out what $T(61)$ is by brute-forcing the recurrence up to 61. Then we select $d = \max\{610, T(61)/61\}$. The induction is completed.

3.2.2 Issues when using the Substitution Method

Now we will try out an example where our guess is incorrect. Consider MergeSort, which has the recurrence $T(n) \leq n + 2T\left(\frac{n}{2}\right)$. We will guess that the algorithm is linear.

$$T(n) \leq \begin{cases} dn_0 \text{ if } n = n_0 \\ d \cdot n \text{ if } n > n_0 \end{cases}$$

We try the inductive step. We try to pick some d such that for all $n \geq n_0$,

$$n + \sum_{i=1}^{k} dg(n_i) \leq d \cdot g(n)$$

$$n + 2 \cdot d \cdot \frac{n}{2} \leq dn$$

$$n(1 + d) \leq dn$$

$$n + dn \leq dn$$

$$n < 0,$$

However, the above can **never** be true, and there is no choice of d that works! Thus our guess was incorrect.

This time the guess was incorrect since MergeSort takes superlinear time. Sometimes however the guess can be asymptotically correct but the induction might not work out. Consider for instance $T(n) \leq 2T(n/2) + 1$, the runtime for FindMin.

We know that the runtime is $O(n)$ so let's try to prove it with the substitution method. Let's guess that $T(n) \leq cn$ for all $n \geq n_0$.

First we do the induction step: We assume that $T(n/2) \leq cn/2$ and consider $T(n)$. We want that

$$2 \cdot cn/2 + 1 \leq cn.$$

However this is impossible since the LHS is $cn + 1$ and the RHS is cn.

This doesn't mean that $T(n)$ is not $O(n)$, but that we chose the wrong linear function. We could guess instead that $T(n) \leq cn - 1$. Now for the induction we get $2 \cdot (cn/2 - 1) + 1 = cn - 1$ which is true for all c. We can then even leave the base case $T(1) = 1$.

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 $z_1 \cdot 10^n + z_4 \cdot 10^{\frac{n}{2}} + z_2$

3.3 Master Method

The Master Method uses the **Master Theorem** to solve recurrences of a special form.

Theorem 3.1 (Master Theorem)**.** *A time recurrence $T(n) = f(n) + a \cdot T(n/b)$, where $a \geq 1$, $b > 1$ are constants solves to:*

Case 1: $T(n) = \Theta\left(n^{\log_b a}\right)$ *if* $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ *for some constant* $\epsilon > 0$. *(intuitively: the cost is dominated by the leaves of the recursion tree).*

Case 2: $T(n) = \Theta\left(f(n) \log n\right)$ *if* $f(n) = \Theta\left(n^{\log_b a}\right)$, *or*

Case 3: $T(n) = \Theta(f(n))$ *if* $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ *for some constant* $\epsilon > 0$ *and for* $c < 1$, $af(n/b) \leq cf(n)$. *(intuitively: cost is dominated by root).*

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 digits. In lecture 1 we saw two approaches to solving this problem.

Let $Add(n)$ be the runtime to add two n digit integers. Then, the runtime of Mult1 can be written as the recurrence $T_1(n) \leq c \cdot Add(n) + 4T_1\left(\frac{n}{2}\right)$, and Karatsuba's runtime can be written as the recurrence $T(n) \leq c \cdot Add(n) + 3T\left(\frac{n}{2}\right)$.

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 $i-1$st digits. We can then rewrite our Mult1 recurrence as

$$T_1(n) \leq c \cdot n + 4T_1\left(\frac{n}{2}\right),$$

and Karatsuba as

$$T(n) \leq c \cdot n + 3T\left(\frac{n}{2}\right).$$

For Mult1, $f(n) = cn$, $a = 4$ and $b = 2$, so $n^{\log_b a} = n^2$. We can see that $f(n) = cn \leq O(n^{2-\epsilon})$ for any $\epsilon \leq 1$, which means case 1 of the Master Theorem applies and

$$T_1(n) \leq \Theta(n^2).$$

For Karatsuba, $f(n) = cn$, $a = 3$ and $b = 2$, so $n^{\log_b a} = n^{\log_2 3} \leq n^{1.59}$. We see that $f(n) = cn \leq O(n^{1.59-\epsilon})$ (for instance for $\epsilon = 0.5$), which means case 1 of the Master Theorem applies and

$$T(n) \leq O(n^{1.59}).$$

Therefore, Karatsuba's algorithm is $T(n) \leq \Theta(n^{1.59})$. For integer multiplication, Karatsuba is the clear winner!