

# Recitation 1: Review, the Master Theorem, Divide and Conquer

## 1 Introduction

Today's recitation mostly aims to review some of the basic material that is necessary for the analysis of algorithms. Most students will be familiar with what's covered today, but this is not what you should expect from future recitations - those will cover new 6.046 material.

## 2 Asymptotic Notation

### 2.1 Definitions

First let us review the definitions. The quantifier  $\forall$  means for all, and  $\exists$  means there exists.

**Definition 1** “Big  $O$ ” is represented as  $O(\dots)$ . If  $f(n) = O(g(n))$  then  $\exists C > 0$  such that  $\exists N$  where  $\forall n > N$   $f(n) < Cg(n)$ .

*In English:*  $f(n) = O(g(n))$  if for very large  $n$ ,  $f(n)$  is smaller than  $Cg(n)$  for some sufficiently large constant  $C$ .

**Definition 2** “Little  $o$ ” is represented as  $o(\dots)$ . If  $f(n) = o(g(n))$  then  $\forall \epsilon > 0$   $\exists N$  where  $\forall n > N$   $f(n) < \epsilon g(n)$ .

*In English:*  $f(n) = o(g(n))$  if for very large  $n$ ,  $f(n)$  becomes arbitrarily smaller than  $g(n)$ .

**Definition 3** “Omega” is represented as  $\Omega(\dots)$ . If  $f(n) = \Omega(g(n))$  then  $\exists C > 0$  such that  $\exists N$  where  $\forall n > N$   $Cf(n) > g(n)$ .

*In English:*  $f(n) = \Omega(g(n))$  if for very large  $n$ ,  $Cf(n)$  is larger than  $g(n)$  for some sufficiently large constant  $C$ .

**Definition 4** “Theta” is represented as  $\Theta(\dots)$ . If  $f(n) = \Theta(g(n))$  then  $\exists a, b > 0$  such that  $\exists N$  where  $\forall n > N$   $af(n) > g(n) > bf(n)$ .

*In English:*  $f(n) = \Theta(g(n))$  if for very large  $n$ ,  $f(n)$  and  $g(n)$  grow at the same speed up to a constant multiple.

Intuitively, we are ignoring constant factors and low-order terms to focus only on the asymptotic order of growth of the functions. A useful mnemonic device may be to think of  $O$  as meaning smaller than or equal to,  $o$  as strictly less than,  $\Omega$  as greater than or equal to, and  $\Theta$  as equal to.

## 2.2 Examples

1.  $5 = O(n^2)$  and  $5 = o(n^2)$ . 5 is NOT  $\Theta(n^2)$  or  $\Omega(n^2)$ .
2.  $3n^3 + n^2 = O(n^3)$ ,  $3n^3 + n^2 = \Omega(n^3)$ , and  $3n^3 + n^2 = \Theta(n^3)$ .  $3n^3 + n^2$  is NOT  $o(n^3)$ .
3.  $n^{0.000001} = \Omega(\lg(n))$ .  $n^{0.000001}$  is NOT Theta, big O, or little-o of  $\lg n$ .

## 3 Writing proofs

6.046 is a mathematically heavy class, and requires proofs on the pssets and on the exams. When writing a proof, it is important to ensure that your argument is clear, concise, and rigorous.

### 3.1 Clear

One important step in writing a clear proof is having a good proof structure. This means that your proof is broken down into separate chunks, each of which is more easily understood on its own, which together imply the overall statement. Intermediate results are called *lemmas* and typically improve clarity.

You should also aim to introduce useful variables to make your writing clearer. Saying “the second number referred to previously” is much harder to understand than “ $x_2$ ”, especially if it is used repeatedly.

Structuring your proofs well will make them more understandable to you, as well as to other readers, making them easier to write as well.

### 3.2 Concise

When solving a problem initially, you may try various approaches that are not successful, reach something in an unintuitive way, etc. This is fine, but your solution and your proof should not reflect how you came to the solution. Instead, when you reach a solution, you should think carefully about why the solution is correct. Start your proof from the information you are given about the problem and previous results you know, and proceed from that point to the solution, without introducing any unnecessary results.

### 3.3 Rigorous

A rigorous proof is one where each statement is precise and each implication step is correct.

A precise statement is one which is clearly either correct or incorrect. For instance “The graph  $G$  has at least  $100n$  vertices” is a precise statement, while “The graph  $G$  has lots of vertices” is not precise. When a proof contains imprecise statements, it is often impossible to decide whether the proof is valid at all, so such statements should be avoided.

Proofs are typically structured as a chain of implications. This implies that, and that and the other imply the third, and so on. For the proof to be rigorous, each implication must be correct. Note that a proof may arrive at a true result while still having incorrect steps. For instance, suppose you are trying to prove the statement “If  $x > 0$ ,  $x + 1 < x^2 + 2x + 1$ ”. Here is one proof attempt:

*Proof.* For all  $a$ ,  $a < a^2$ . Substitute  $x + 1$  for  $a$ . Thus,  $x + 1 < x^2 + 2x + 1$ .

This proof is not correct, because the first step, the assertion that for all  $a$ ,  $a < a^2$ , is false. Specifically, if  $0 \leq a \leq 1$ ,  $a \geq a^2$ . However, the proof can be adjusted to be rigorous, because we can prove that  $a > 1$ .

*Proof.* For all  $a > 1 \in \mathbb{R}$ ,  $a < a^2$ . Since  $x > 0$ ,  $x + 1 > 1$ , and so  $x + 1 > 1$ . Substitute  $x + 1$  for  $a$ . Thus,  $x + 1 < x^2 + 2x + 1$ .

### 3.4 Example - Robotic Coin Collection

Recall that in each round of the “peel algorithm” for the robotic coin collection problem, we choose the bottommost coin possible in each column as we progress from the left to the right of the grid. We want to show that the number of rounds of this algorithm is equal to the minimum number of robots required to pick up all of the coins. Since we know that the required number of robots is lower bounded by the maximum size of any disjoint set of locations on the grid, it suffices to show that there is a disjoint set which is as large as the number of rounds of our algorithm. In class, we presented the intuition behind this claim as well as a sort of “proof by picture.” Let us now formally prove this statement in the manner we expect to see in problem sets and exams (i.e., clear, concise, and rigorous).

Let us define some useful notation. Suppose our grid is specified by an  $m \times n$  matrix. We will associate each coin on the grid with its location in the matrix (the origin is in the top left). For a given coin  $c$ , let  $c_x$  be its  $x$ -coordinate and  $c_y$  be its  $y$ -coordinate. We say that  $c' \triangleleft c$  if  $c'$  and  $c$  are disjoint such that the  $c'_x < c_x$  and  $c'_y > c_y$ .

Let us first prove the transitivity of disjointness.

**Lemma 1 (Transitivity of Disjointness)** *Let  $a$ ,  $b$ , and  $c$  be coins. If  $a \triangleleft b$  and  $b \triangleleft c$ , then  $a \triangleleft c$ .*

*Proof.*  $a \triangleleft b$  implies  $a_x < b_x$  and  $a_y > b_y$ .  $b \triangleleft c$  implies  $b_x < c_x$  and  $b_y > c_y$ . Therefore, we have

$$a_x < b_x < c_x \text{ and } a_y > b_y > c_y,$$

so  $a \triangleleft c$ . □

Let  $P_i$  be the path taken by the peel algorithm in the  $i$ th round. We now state the key lemma.

**Lemma 2** Consider any two paths  $P_i$  and  $P_j$  for  $i < j$ . For all coins  $c \in P_j$  there exists a coin  $c' \in P_i$  such that  $c' \triangleleft c$ .

*Proof.* Fix some coin  $c \in P_j$ . First notice that during the  $i$ th round, the path  $P_i$  contains the entire column of coins with the minimum  $x$ -coordinate. Therefore, there must exist coins in  $P_i$  whose  $x$ -coordinate is less than  $c_x$ . Let  $c' \in P_i$  be the lowest coin on the column nearest to  $c$ 's left. Therefore, we know that  $c'$  is to the left of  $c$ . To prove that  $c' \triangleleft c$ , it suffices to show that  $c'$  is below  $c$  (that is,  $c'_y > c_y$ ). Suppose by contradiction that  $c'_y \leq c_y$ . Because we chose  $c'$  to be the coin closest to the left of  $c$ , we know that there are no other coins on path  $P_i$  until column  $x$ . Therefore, the robot on path  $P_i$  must pass the coin  $c$ , contradicting the fact that  $c$  was in path  $P_j$ .  $\square$

We are now ready to prove the theorem.

**Theorem 3** Suppose peel algorithm takes  $r$  rounds. Then, there is a disjoint set of coins of size  $r$ .

*Proof.* We will specify a set of  $r$  disjoint coins  $c_1, \dots, c_r$ . First, find coin  $c_r \in P_r$ . In decreasing order, find coin  $c_i$  disjoint from  $c_{i+1}$  using Lemma 2. We have  $c_1 \triangleleft c_2, c_2 \triangleleft c_3, \dots, c_{r-1} \triangleleft c_r$ . Therefore, by the transitivity of disjointness, we have that  $c_1, \dots, c_r$  are disjoint.  $\square$

### 3.5 Example - Mergesort

In this section, we will analyze/describe the Mergesort algorithm to give an example of both a rigorous proof, and the divide-and-conquer technique. The problem we are attempting to solve is: given an array of elements  $a$ , and a comparison function  $<$ , we wish to sort  $a$  according to  $<$ .

The algorithm is:

1. If  $a$  is of length 1, return it unchanged.
2. Otherwise, split the array in half, into subarrays  $l$  and  $r$ , where the odd element, if it exists, is placed in  $l$ .
3. Using this mergesort algorithm, sort  $l$  and  $r$ .
4. Merge  $l$  and  $r$  by comparing the first elements of each, putting the smaller into an output list  $o$ , and removing that element from  $l$  or  $r$ .
5. Repeat 4 until one of  $l$  or  $r$  is empty, and move all of its elements to the end of  $o$ .

For every algorithm you create in this class, you will have to give both a proof that the algorithm is correct (i.e., does what you claim it should do), and a proof that its running time is bounded by what you claim (unless otherwise stated). First, we show that the mergesort algorithm correctly sorts the list.

*Proof.* We will use strong induction, a general mathematical principle. In particular, we will prove that if mergesort correctly sorts every array of length up to  $n - 1$ , it will also correctly sort

every array of length  $n$ , and we will prove that mergesort correctly sorts every array of length 1. Together, these imply that mergesort correctly sorts every list of positive length.

**Lemma 4** *Mergesort correctly sorts every array of length 1.*

Every array of length 1 is sorted, and mergesort returns arrays of length 1 unchanged. This proves the lemma.

**Lemma 5** *If mergesort correctly sorts every array of length up to  $n - 1$ , it will also correctly sort every array of length  $n$ , if  $n > 1$ .*

If  $n > 1$ , the left and right subarrays will both be shorter than the original array, since neither subarray is empty. By assumption, after the third step, each of the subarrays will be correctly sorted. Now, we must show that the merge step results in a sorted array. To do this, we will show that the following invariant is maintained: The first element of each of  $l$  ( $l_1$ ) and  $r$  ( $r_1$ ) is larger than the last element placed in  $o$  ( $o_{-1}$ ). Since  $o_{-1}$  came from one of  $l$  or  $r$ , and each of those subarrays is sorted, it will be smaller than the new first element of that list. Without loss of generality, assume it came from  $l$ . Then, since  $l$  was sorted,  $o_{-1} < l_1$ . Moreover, since the smaller of the two first elements was placed in  $o$ ,  $o_{-1} < r_1$  as well. Therefore, whenever an element is moved into  $o$ , it is larger than the previous last element, and so  $o$  is maintained in sorted order throughout the step. This is maintained when the final elements are moved to  $o$  (in step 5), since those elements are larger than  $o_{-1}$  by the same argument as above and sorted to start with. Thus, the lemma is proved.

With both lemmas, we may apply the principle of strong induction to conclude that mergesort is always correct.

Before we move on to proving the runtime of mergesort, we first discuss common methods for analyzing the runtimes of divide-and-conquer algorithms.

## 4 Divide and Conquer

The Mergesort algorithm is an example of the divide-and-conquer approach. We can summarize the algorithm in three phases.

1. **Divide** the array in two subarrays.
2. **Conquer** the two subarrays separately by recursively using mergesort on them.
3. **Combine** the sorted subarrays, and output the sorted result.

This structure is emblematic of the divide and conquer paradigm, which we can more generally describe as follows.

1. **Divide** the problem into subproblems, where the subproblems are smaller instances of the same type of problem. Smaller arrays to sort are an example of smaller instances of the sorting problem.

2. **Conquer** each subproblem separately. This is done by recursively using the algorithm on them.
3. **Combine** the results to the subproblems. This yields a solution to the full problem.

We proved that divide and conquer gives an answer to the sorting problem. The more important question to ask, however, is why divide and conquer is a good strategy for approaching sorting, and more generally what problems are suitable for a divide and conquer strategy. An array can be split into subproblems very naturally: it suffices to break the array into subarrays. Typically, the presence of natural subproblems is a mark of the fact that divide and conquer is a suitable strategy. Besides arrays, other commonly found examples include matrices, which can be split into submatrices, and geometrical problems.

## 5 Run time Recurrences

With many types of algorithms we want to solve the larger problem by splitting it into smaller parts, solving each of those and then merging the results. Solving algorithms like this often results in fast run times.

It can help from a design perspective because you only need to verify the the correctness of the smallest problem and the correctness of the merging of two solved problems. It can help from an analysis perspective because this method is so common time analysis techniques and machinery have already been produced.

So what do these recurrences look like? Let  $T(n)$  be the running time for the algorithm on a problem of size  $n$ . Standard divide-and-conquer algorithms often result in a recurrence of the form

$$T(n) = aT(n/b) + f(n).$$

### 5.1 Master Theorem

The master theorem is a general method to analyze recurrence relations of the above form. There are three cases:

1. If  $f(n)$  polynomially less than  $n^{\log_b(a)}$  (i.e.  $f(n) = O(n^{\log_b(a) - \epsilon})$  for some constant  $\epsilon > 0$ ), then  $T(n) = \Theta(n^{\log_b(a)})$ .
2. If  $f(n) = \Theta(n^{\log_b(a)} \log^k(n))$  for some constant  $k \geq 0$ , then  $T(n) = \Theta(f(n) \log(n)) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$ .
3. If  $f(n)$  is polynomially greater than  $n^{\log_b(a)}$  (i.e.  $f(n) = \Omega(n^{\log_b(a) + \epsilon})$  for some constant  $\epsilon > 0$ ), and  $af(n/b) \leq cf(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) = \Theta(f(n))$ .

It's worth noting that not all divide-and-conquer algorithms will necessarily give rise to a recurrence which is in the form of the Master theorem. For instance, if  $n^{\log_b(a)}$  is greater, but not polynomially greater, than  $f(n)$ , the Master Theorem cannot be used to determine a precise bound. An example of this is the recurrence  $T(n) = 2T(n/2) + \Theta(n/\log(n))$ . (The solution to this recurrence is  $T(n) = O(n \log \log n)$ , which can be arrived at by unrolling the recurrence and using the fact that the harmonic series is  $O(\log n)$ .)

In general, you also can't directly apply the Master theorem when the subproblems aren't of size  $n/b$ . For instance, you might encounter recurrences which look like  $T(n) = 2T(\sqrt{n}) + 1$ . The solution to this recurrence is  $O(\log n)$  and can be obtained by either unrolling the recurrence (see the appendix on recursion trees) or through substitution, which we explore in the next section.

## 5.2 Substitution

Substitution of variables is often helpful in solving recurrences. For instance, suppose we wish to solve the recurrence  $T(n) = 2T(\sqrt{n}) + 1$ , which we've already seen does not fall into the scope of the Master theorem. Let us create a new variable  $m$  such that  $n = 2^m$ . We now have

$$T(2^m) = 2T(2^{m/2}) + 1.$$

Furthermore, we can create a new function  $S(m) = T(2^m)$ , so that

$$S(m) = 2S(m/2) + 1.$$

Notice that  $S(m)$  is in a form where we can apply the Master theorem! Therefore, we have  $S(m) = O(m)$ , and  $T(n) = O(m) = O(\log n)$ .

## 5.3 Guess and Check

You can often determine the solution for a recurrence by inspection. In that case, you can just guess the solution to the recurrence and then prove that it is correct by induction. Some examples include:

1.  $T(n) = 2T(n/4) + n$  ( $T(n) = O(n)$ )
2.  $T(n) = 2T(n/2) + 1$  ( $T(n) = O(n)$ )
3.  $T(n) = 4T(n/2) + n$  ( $T(n) = O(n^2)$ )
4.  $T(n) = 2T(n/2) + n$  ( $T(n) = O(n \log n)$ , e.g. mergesort). Here it's crucial that guess-and-check is done with  $T(n) \leq n \log_2 n$ , and using the fact that  $\log_2 \frac{n}{2} = \log_2 n - 1$ .

**A common mistake:** When you want to guess, you have to come up with the closed-form expression and not just asymptotic notation. For example, consider  $T(n) = 2T(n/4) + 1$ . Suppose

you guess that  $T(n) = O(1)$ . It seems it is correct, since  $T(n) = 2O(1) + 1 = O(1)$ . However, this is not true. If we guessed a closed-form expression of a function, namely  $c$  for a constant  $c$ , we would not be able to show that  $T(n) = O(1)$ . Let's try to prove  $T(n) \leq c$ . By induction hypothesis, assume  $T(m) \leq c$  for any  $m < n$ , then we can only show that  $T(n) \leq 2c + 1$ . This does not imply that  $T(n)$  is also at most  $c$ . Thus, the induction fails.

More subtly, note that a function failing to satisfy a recurrence does not on its own indicate whether the actual solution to the recurrence is large or smaller. As a contrived example, consider the recurrence  $T(n) = 2T(n/2) + 1$ . The guess  $T(n) = cn + 1$  fails, as  $2T(n/2) + 1 = 2(cn/2 + 1) + 1 = cn + 3 \not\leq cn + 1$ .

## 5.4 The Running Time of Mergesort

As an example, we will go back to Mergesort and analyze its running time by defining a recurrence.

As we saw in the previous sections, mergesort divides the problem in two halves, solves the two halves recursively and it merges them by iterating through all the elements in the two sublists.

Let  $T(n)$  be defined as the number of operations required to sort a list of  $n$  elements using mergesort. Then,  $T(n)$  is equal to the time to solve the two subproblems plus the time to merge the two sublists. Since the two lists are of length  $n/2$ , and since merging iterates through all the  $n$  elements, we know that:

$$T(n) = 2T(n/2) + \Theta(n) \tag{1}$$

We solve this recurrence by using the Master Theorem. Using the notation from the previous part, we have  $a = 2$ ,  $b = 2$ ,  $f(n) = \Theta(n)$ . Since  $f(n)$  is  $\Theta(n)$ , we are in the second case of the Master Theorem. Consequently, we can apply the theorem to conclude that  $T(n) = \Theta(n \log n)$ .

## A Recursion Trees

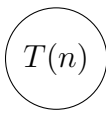
A recursion tree can help to visualize where the algorithm is doing the most work—at the root of the tree, at the leaves, or spread evenly throughout (or perhaps something more complicated).

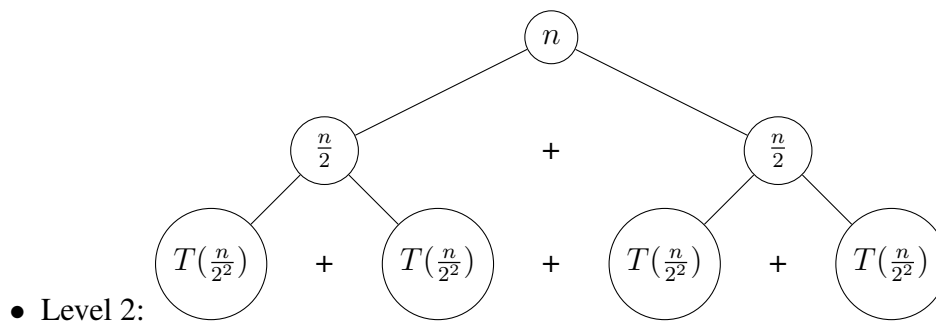
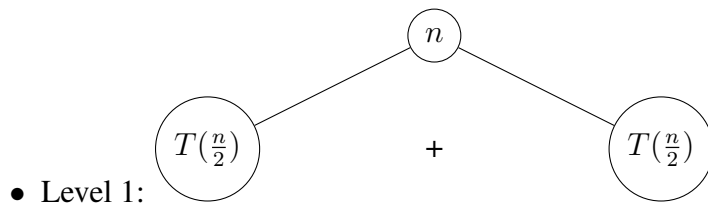
Basic idea: draw a node for each instance of the problem, and draw edges to represent the parent-child relationships among the problems. In each node, write the cost incurred directly at that subproblem ( $f(n)$  for the particular  $n$  of the subproblem). The total cost of the algorithm may be computed by summing up the costs of all the nodes.

The branching factor for the tree is  $a$ . The height (number of levels) is  $\log_b(n)$ . At a given level  $h$  levels below the root, each node incurs  $f(\frac{n}{b^h})$ . At a given level  $h$  levels below the root, there are  $a^h$  nodes. Summing across the tree, the cost of a given level  $h$  levels below the root will be  $f(\frac{n}{b^h})a^h$ .

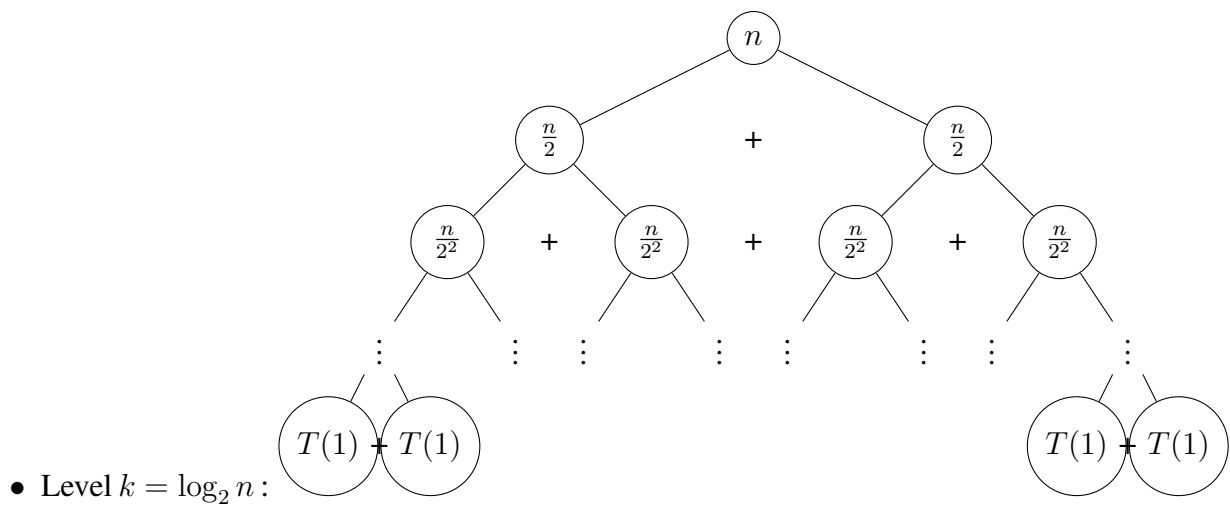


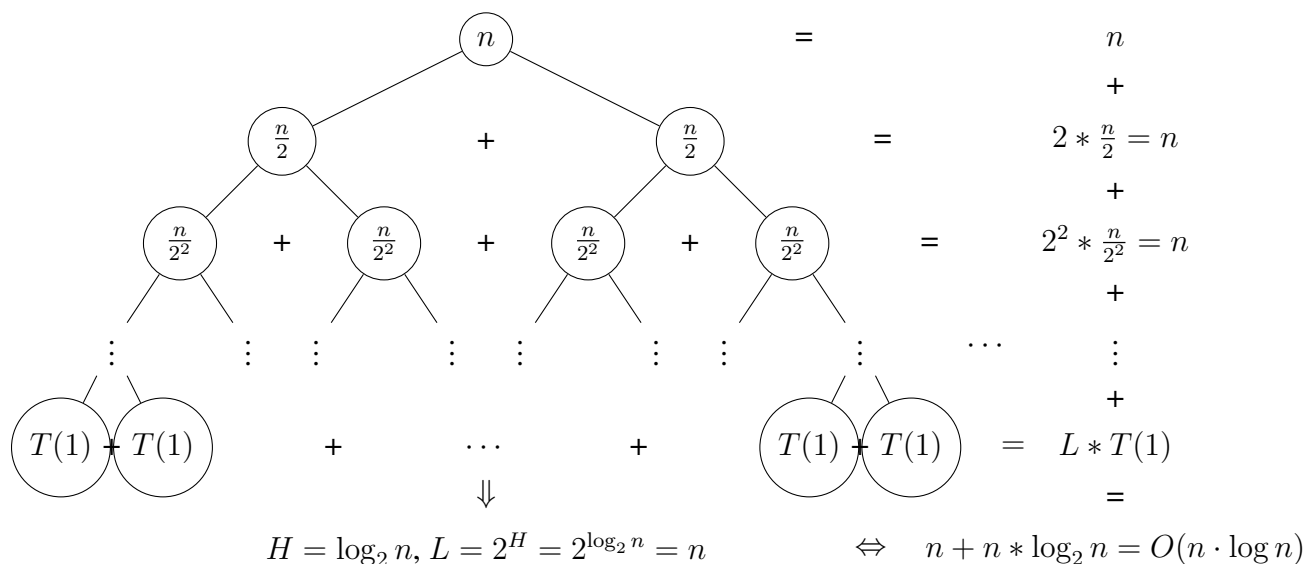
**A.0.1 Building a recursion tree for  $T(n) = 2T(n/2) + n$** 

- Level 0 : 



⋮

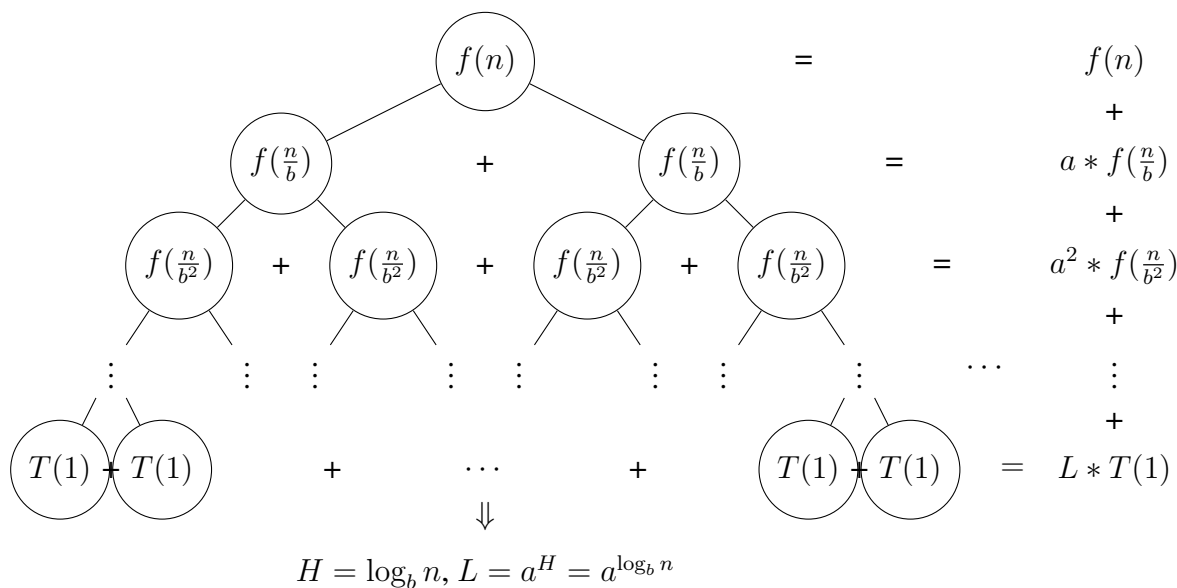




Why does  $n^{\log_b(a)}$  shows up a lot in recurrences?

It shows up because this is a count of the number of leaves of the recursion tree. The number of leaves is  $a^H$  where  $H$  is the height of the tree,  $\log_b n$ . We then have  $a^H = a^{\log_b(n)} = n^{\log_b(a)}$ .

**Building a recursion tree for  $T(n) = aT(n/b) + f(n)$**



The analysis of the overall running time of the general case depends on the comparison of  $f(n)$  and  $n^{\log_b a}$ . The proof of the exact analysis of the running time to the general case is what is known as the “Master Theorem”. It is a proof by cases that can be found in Chapter 4.6 of CLRS.