Asymptotically Comparing Functions

In many situations, it may be patently obvious that one function is asymptotically larger than another, such as when comparing the growth of n^2 to that of n^3. Still, in other scenarios, the difference may not be immediately obvious. The following techniques offer a general way to approach this problem.

1. Take the limit of the ratio of the two functions being compared. If it tends to infinity, the numerator is larger. If it tends to 0, the denominator is larger. For example, suppose we want to compare 2^n to 3^n. Then we see that

$$\lim_{n\to\infty} \frac{2^n}{3^n} = (2/3)^n \to 0 \tag{1}$$

   And so 3^n is asymptotically larger.

2. Another useful technique is to transform the two functions by a strictly increasing function, like $\log n$ or $\sqrt{n}$, and compare the behavior of the functions then. For instance, it may be tricky to see how $2^{(\log_2 n)^2}$ compares to $n^{\log_2 n}$ but if we take $\log_2$ of these, we see that they are asymptotically the same. **Be careful to make sure you understand how constants transform under these functions. For instance, if you apply $\log_2$ to n^2 and n^3, the results are within a constant factor of each other. Still, after applying $\log$, functions are asymptotically the same only if they are within a constant additive factor of each other.**

3. Finally, you can use l'hopital's rule to determine asymptotic behavior. The rule states that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \tag{2}$$

   Note that we can repeatedly take the derivative until a clear trend is found. For an example, suppose we are trying to compare $\log n$ to $n^{0.01}$. If we take the ratio of their derivatives, we obtain

$$\lim_{n\to\infty} \frac{1/n}{0.01/n^{0.99}} = 100/n^{0.01} \to 0 \tag{3}$$

   Implying that $n^{0.01}$ is asymptotically greater.

Solving recurrences

Supplementary Reading: CLRS, Chapter 4

Recall back to peak finding where we solved recurrences by showing them in the form of "Runtime of original problem" = "Runtime of reduced problem" + "Time taken to reduce problem", and

then solved them using by expanding out the subproblems. We are going to formalize this a little more.

The form of the recurrences that we'll be dealing with look like:

$$T(n) = aT(n/b) + f(n) \tag{4}$$

Where a is the number of subproblems that the original problem is divided into, n/b is the size of each subproblem, and $f(n)$ is how long it takes to divide into subproblems and combine the results of the subproblems.

Example: Binary Search Binary searching a sorted list involves splitting the list in two and recursively searching into one half of the list. $a = 1$ since whenever we split the list in two, we just call a binary search on one half. $b = 2$ since the new subproblem has half the elements of the original problem. $f(n) = O(1)$ since finding the middle of a list and deciding which half to recurse into is a constant time operation.

Recursion trees

One way to solve recurrences is to draw a recursion tree where each node in the tree represents a subproblem and the value at each node represents the amount of work spent at each subproblem. The root node represents the original problem. In a recursion tree, every node that is not a leaf has a children, representing the number of subproblems it is splitting into. To figure out how much work is being spent at each subproblem, first find the size of the subproblem with the help of b, then substitute the size of the subproblem in the recurrence formula $T(n)$, then take the value of $f(n)$ as the amount of work spent at that subproblem. Long story short, a node with a problem size of x, the node will have a children each contributing $f(x/b)$ amount of work.

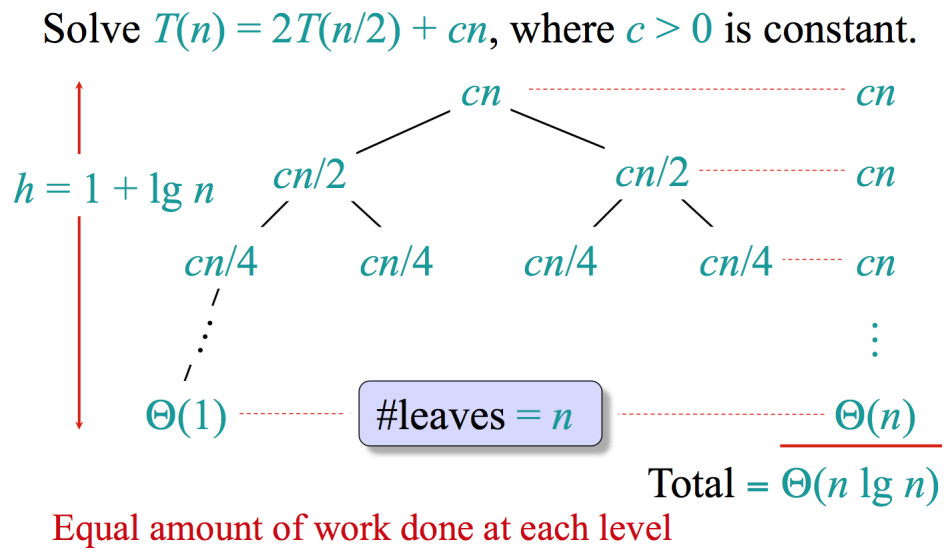
The work at the leaves is $T(1)$, since at that point we have divided the original problem up until it can no longer be further divided. Note that this means that the work contributed by the leaves are $O(1)$.

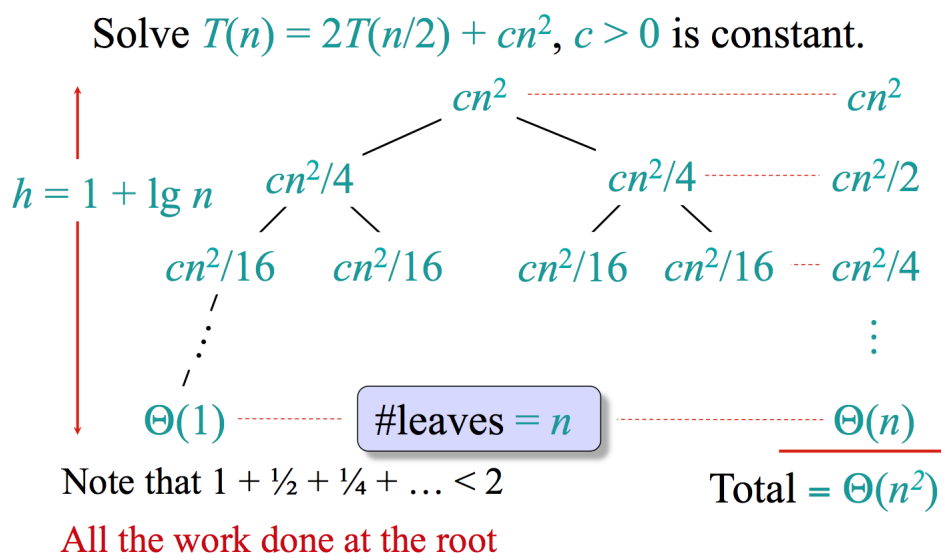
Once we have our tree, the total runtime can be calculated by summing up the work contributed by all of the nodes. We can do this by summing up the work at each level of the tree, then summing up the levels of the tree.

Example 1: (Merge sort) Merge sorting a list involves splitting the list in two and recursively merge sorting each half of the list. $a = 2$ since we call merge sort twice at every recursion (once on each half). $b = 2$ since each of the new subproblem has half the elements in the original list. $f(n) = O(n)$ since combining the results of the subproblems, the merge operation, is $O(n)$.

The merge sort recursion tree is considered somewhat balanced. The work done at each level stays consistent (in this case, it is $O(n)$ at each level) so the total work done can be calculated by multiplying the work at each level by the number of levels, hence $O(n \log n)$ running time for merge sort. However, there are two other cases that may happen.

Example 2: If the work at each level geometrically decreases as we go down the tree, then the work done at the root node (i.e. $f(n)$) will dominate the runtime.



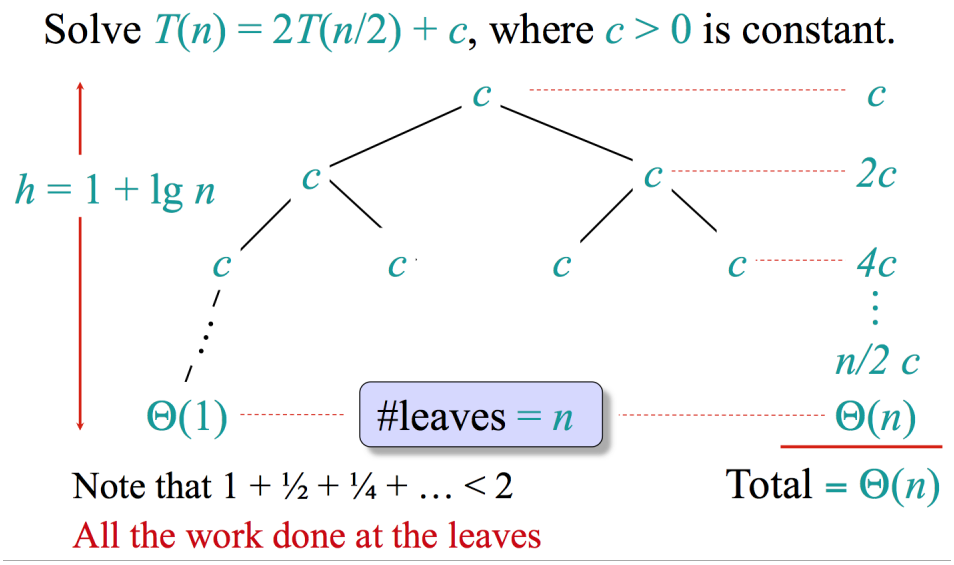


Example 3: If instead the work at each level geometrically increases as we go down the tree, then the work done at the bottom level (i.e. number of leaves $\times f(1)$ or O(number of leaves) or $O(n^{\log_b a})$) will dominate the runtime.

Master Theorem

Whether or not the work per level stays relatively consistent, geometrically increases, or geometrically decreases can be determined by looking at a, b, and $f(n)$ from the recurrence formula. Using this information gives us the master theorem, which allows us to solve recurrences of the form $T(n) = aT(n/b) + f(n)$.

1. **Case 1:** If $f(n) = O(n^{\log_b a - \epsilon})$, then the amount of work per level geometrically increases



as we go down the tree. The work at the leaf level dominates and $T(n) = \Theta(n^{\log_b a})$.

2. **Case 2:** If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then the amount of work per level have about the same cost (i.e. work per level does not *polynomially* increase or decrease, though work per level still may increase or decrease at some slower rate). We have to take the work of all levels into account and $T(n) = \Theta(n^{\log_b a} \log^k n \log n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

3. **Case 3:** If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then the amount of work per level geometrically decreases as we go down the tree. The work at the root level dominates and $T(n) = \Theta(f(n))$.

Example 1: $T(n) = 2T(n/2) + 1$
Example 2: $T(n) = 2T(n/2) + n$ **(merge sort)**
Example 3: $T(n) = 3T(n/2) + O(n)$ **(Karatsuba multiplication)**
Example 4: $T(n) = 4T(n/2) + n^3$ **((naive) integer multiplication)**

Change of Variables

For some particularly tricky recurrences, it may be necessary to combine the existing methods we've looked at with a method known as **change of variables**. The idea behind change of variables is to rewrite a tricky recurrence in terms of a new variable that is somehow related to the old variable, solve the new recurrence, then change back to the original variable.

Consider, for example, the following recurrence containing a square root:

$$T(n) = 2T(\sqrt{n}) + \Theta(\log n)$$

None of our existing methods provide an easy way to solve this recurrence. To solve this we can perform a change of variables by defining a new variable m as $n = 2^m$. Substituting this in

gives:

$$\begin{aligned} T(2^m) &= 2T(\sqrt{2^m}) + \Theta(\log 2^m) \\ &= 2T(2^{(m/2)}) + \Theta(m) \end{aligned}$$

This recursion still isn't in the standard form that we expect for Master Theorem, so we define a new recurrence, S as $S(m) = T(2^m) = T(n)$. This gives us:

$$\begin{aligned} S(m) = T(2^m) &= 2T(2^{(m/2)}) + \Theta(m) \\ &= 2S(m/2) + \Theta(m) \end{aligned}$$

Suddenly, we have a form we can work with using any of the previous methods we've learned. Using Master Theorem, for example, gives us $S(m) = m \log m$. Now we simply have to substitute back to make this meaningful in the context of T and n:

$$\begin{aligned} S(m) &= \Theta(m \log m) \\ T(n) &= \Theta(\log n \log \log n) \end{aligned}$$

Exercise 3 – Suppose you are given $T(\log_2 n) = 2T(\log_4 n) + \Theta(\log \log n))$. Compute a closed form for $T(x)$ using change of variables.