Problem 2.1-2: Show that for any real constants a *and* b*,* $b > 0$,

$$(n+a)^b = \Theta(n^b)$$

To show $f(n) = \Theta(g(n))$, we must show O and Ω. *Go back to the definition!*

- *Big O* – Must show that $(n+a)^b \leq c_1 \cdot n^b$ for all $n > n_0$. When is this true? If $c_1 = 2^b$, this is true for all $n > |a|$ since $n+a < 2n$, and raise both sides to the b.

- *Big* Ω – Must show that $(n+a)^b \geq c_2 \cdot n^b$ for all $n > n_0$. When is this true? If $c_2 = (1/2)^b$, this is true for all $n > 3|a|/2$ since $n+a > n/2$, and raise both sides to the b.

Note the need for absolute values.

Problem 2.1-4:

(a) Is $2^{n+1} = O(2^n)$*?*

(b) Is $2^{2n} = O(2^n)$*?*

(a) Is $2^{n+1} = O(2^n)$?

Is $2^{n+1} \le c * 2^n$?

Yes, if $c \ge 2$ for all n

(b) Is $2^{2n} = O(2^n)$?

Is $2^{2n} \le c * 2n$?

note $2^{2n} = 2^n * 2^n$

Is $2^n * 2^n \le c * 2^n$?

Is $2^n \le c$?

No! Certainly for any constant c we can find an n such that this is not true.

Recurrence Relations

Many algorithms, particularly divide and conquer algorithms, have time complexities which are naturally modeled by recurrence relations.

A recurrence relation is an equation which is defined in terms of itself.

Why are recurrences good things?

1. Many natural functions are easily expressed as recurrences:

$$a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n \quad (polynomial)$$

$$a_n = 2a_{n-1}, a_1 = 1 \longrightarrow a_n = 2^{n-1} \quad (exponential)$$

$$a_n = na_{n-1}, a_1 = 1 \longrightarrow a_n = n! \quad (weird\ function)$$

2. It is often easy to find a recurrence as the solution of a counting problem. *Solving* the recurrence can be done for many special cases as we will see, although it is somewhat of an art.

Recursion *is* Mathematical Induction!

In both, we have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces.

The *initial* or boundary condition terminate the recursion.

As we will see, induction provides a useful tool to solve recurrences – guess a solution and prove it by induction.

$$T_n = 2T_{n-1} + 1, T_0 = 0$$

n	0	1	2	3	4	5	6	7
T_n	0	1	3	7	15	31	63	127

Guess what the solution is?

Prove $T_n = 2^n - 1$ by induction:

1. Show that the basis is true: $T_0 = 2^0 - 1 = 0$.
2. Now assume true for T_{n-1}.
3. Using this assumption show:

$$T_n = 2T_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$$

∎

Solving Recurrences

No general procedure for solving recurrence relations is known, which is why it is an art. My approach is:

Realize that linear, finite history, constant coefficient recurrences always can be solved

Check out any combinatorics or differential equations book for a procedure.

Consider $a_n = 2a_{n-1} + 2a_{n-2} + 1$, $a_1 = 1$, $a_2 = 1$

It has history = 2, degree = 1, and coefficients of 2 and 1. Thus it can be solved mechanically! Proceed:

- Find the characteristic equation, eg.

$$\alpha^2 - 2\alpha = 2 = 0$$

- Solve to get roots, which appear in the exponents.
- Take care of repeated roots and inhomogeneous parts.
- Find the constants to finish the job.

$$a_n = -1/3 + (1-\sqrt{3})^n(1+\sqrt{3})/3 + (1+\sqrt{3})^n(-1+\sqrt{3})/3$$

Systems like Mathematica and Maple have packages for doing this.

Guess a solution and prove by induction

To guess the solution, play around with small values for insight.

Note that you can do inductive proofs with the big-O's notations - just be sure you use it right.

Example: $T(n) \leq 2c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n$.

Show that $T(n) \leq c \cdot n \lg n$ for large enough c and n. Assume that it is true for $n/2$, then

$$\begin{aligned}
T(n) &\leq 2c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \\
&\leq c \cdot n \lg(\lfloor n/2 \rfloor) + n \quad \text{dropping floors makes it bigger} \\
&= c \cdot n(\lg n - (\lg 2 = 1)) + n \quad \text{log of division} \\
&= c \cdot n \lg n - cn + n \\
&\leq c \cdot n \lg n \quad \text{whenever } c > 1
\end{aligned}$$

Starting with basis cases $T(2) = 4$, $T(3) = 5$, lets us complete the proof for $c \geq 2$.

Try backsubstituting until you know what is going on

Also known as the iteration method. Plug the recurrence back into itself until you see a pattern.

Example: $T(n) = 3T(\lfloor n/4 \rfloor) + n$.

Try backsubstituting:

$$\begin{aligned} T(n) &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor) \\ &= n + 3\lfloor n/4 \rfloor + 9(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \end{aligned}$$

The $(3/4)^n$ term should now be obvious.

Although there are only $\log_4 n$ terms before we get to $T(1)$, it doesn't hurt to sum them all since this is a fast growing geometric series:

$$T(n) \leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3} \times T(1))$$

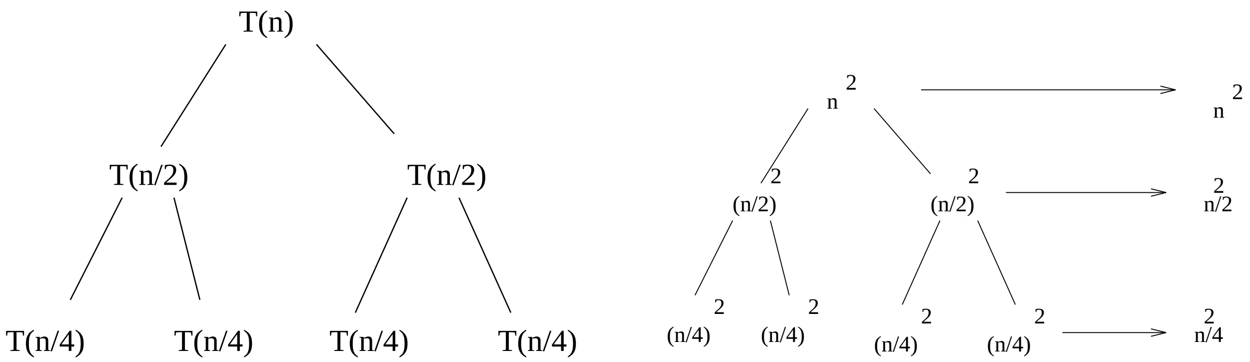
$$T(n) = 4n + o(n) = O(n)$$

Recursion Trees

Drawing a picture of the backsubstitution process gives you a idea of what is going on.

We must keep track of two things – (1) the size of the remaining argument to the recurrence, and (2) the additive stuff to be accumulated during this call.

Example: $T(n) = 2T(n/2) + n^2$



The remaining arguments are on the left, the additive terms on the right.

Although this tree has height $\lg n$, the total sum at each level decreases geometrically, so:

$$T(n) = \sum_{i=0}^{\infty} n^2/2^i = n^2 \sum_{i=0}^{\infty} 1/2^i = \Theta(n^2)$$

The recursion tree framework made this much easier to see than with algebraic backsubstitution.

See if you can use the Master theorem to provide an instant asymptotic solution

The Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

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

Examples of the Master Theorem

Which case of the Master Theorem applies?

- $T(n) = 4T(n/2) + n$

  Reading from the equation, $a = 4$, $b = 2$, and $f(n) = n$.

  Is $n = O(n^{\log_2 4 - \epsilon}) = O(n^{2-\epsilon})$?

  Yes, so case 1 applies and $T(n) = \Theta(n^2)$.

- $T(n) = 4T(n/2) + n^2$

  Reading from the equation, $a = 4$, $b = 2$, and $f(n) = n^2$.

  Is $n^2 = O(n^{\log_2 4 - \epsilon}) = O(n^{2-\epsilon})$?

  No, if $\epsilon > 0$, but it is true if $\epsilon = 0$, so case 2 applies and $T(n) = \Theta(n^2 \log n)$.

- $T(n) = 4T(n/2) + n^3$

  Reading from the equation, $a = 4$, $b = 2$, and $f(n) = n^3$.

  Is $n^3 = \Omega(n^{\log_2 4 + \epsilon}) = \Omega(n^{2+\epsilon})$?

  Yes, for $0 < \epsilon < 1$, so case 3 *might* apply.

  Is $4(n/2)^3 \leq c \cdot n^3$?

  Yes, for $c \geq 1/2$, so there exists a $c < 1$ to satisfy the regularity condition, so case 3 applies and $T(n) = \Theta(n^3)$.

Why should the Master Theorem be true?

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

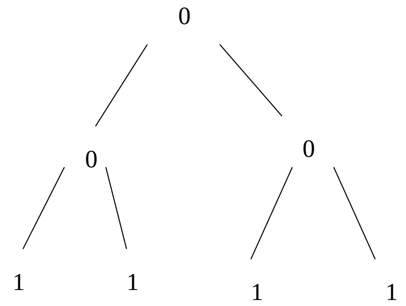
Suppose $f(n)$ is small enough

Say $f(n) = 0$, ie. $T(n) = aT(n/b)$.

Then we have a recursion tree where the only contribution is at the leaves.

There will be $\log_b n$ levels, with a^l leaves at level l.

$$T(n) = a^{\log_b n} = n^{\log_b a} \quad \text{Theorem 2.9 in CLR}$$



so long as $f(n)$ is small enough that it is dwarfed by this, we have case 1 of the Master Theorem!

Suppose f(n) is large enough

If we draw the recursion tree for $T(n) = aT(n/b) + f(n)$.

If $f(n)$ is a big enough function, the one top call can be bigger than the sum of all the little calls.

Example: $f(n) = n^3 > (n/3)^3 + (n/3)^3 + (n/3)^3$. In fact this holds unless $a \geq 27$!

In case 3 of the Master Theorem, the additive term dominates.

In case 2, both parts contribute equally, which is why the log pops up. It is (usually) what we want to have happen in a divide and conquer algorithm.

Famous Algorithms and their Recurrence

Matrix Multiplication

The standard matrix multiplication algorithm for two $n \times n$ matrices is $O(n^3)$.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \qquad \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

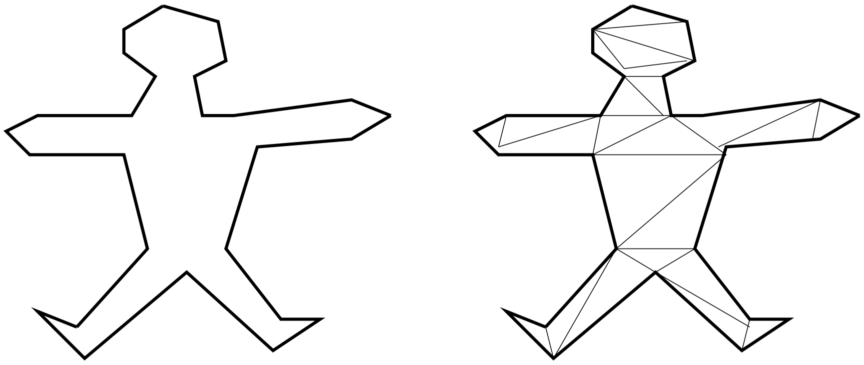
Strassen discovered a divide-and-conquer algorithm which takes $T(n) = 7T(n/2) + O(n^2)$ time.

Since $O(n^{\lg 7})$ dwarfs $O(n^2)$, case 1 of the master theorem applies and $T(n) = O(n^{2.81})$.

This has been "improved" by more and more complicated recurrences until the current best in $O(n^{2.38})$.

Polygon Triangulation

Given a polygon in the plane, add diagonals so that each face is a triangle None of the diagonals are allowed to cross.



Triangulation is an important first step in many geometric algorithms.

The simplest algorithm might be to try each pair of points and check if they see each other. If so, add the diagonal and recur on both halves, for a total of $O(n^3)$.

However, Chazelle gave an algorithm which runs in $T(n) = 2T(n/2)+O(\sqrt{(n)})$ time. Since $n^{1/2} = O(n^{1-\epsilon})$, by case 1 of the Master Theorem, Chazelle's algorithm is linear, ie. $T(n) = O(n)$.

Sorting

The classic divide and conquer recurrence is Mergesort's $T(n) = 2T(n/2) + O(n)$, which divides the data into equal-sized halves and spends linear time merging the halves after they are sorted.

Since $n = O(n^{\log_2 2}) = O(n)$ but not $n = O(n^{1-\epsilon})$, Case 2 of the Master Theorem applies and $T(n) = O(n \log n)$.

In case 2, the divide and merge steps balance out perfectly, as we usually hope for from a divide-and-conquer algorithm.

Mergesort Animations

Approaches to Algorithms Design

Incremental

Job is partly done - do a little more, repeat until done.

A good example of this approach is insertion sort

Divide-and-Conquer

A recursive technique

- Divide problem into sub-problems of the same kind.
- For subproblems that are really small (trivial), solve them directly. Else solve them recursively. (conquer)
- Combine subproblem solutions to solve the whole thing (combine)

A good example of this approach is Mergesort.