COMP 6651
Algorithm Design Techniques

Lecturer: Thomas Fevens

Department of Computer Science and Software Engineering, Concordia U
thomas.fevens@concordia.ca

Table of Contents

Administrivia

See Course Outline (on Moodle course page)

Some notes:

- Textbook: Introduction to Algorithms, 4th Ed., by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein [CLRS]
- Office Hour: Thursday, 1:30pm-2:30pm in ER-947
- Grading Scheme
    1. 16%: Assignments (4). Submitted on Moodle
    2. 19%: Midterm (two stage exam): Oct 22 and 24 (in class). Closed-book
    3. 20%: Project: Details will be posted
    4. 45%: Final Exam: To be scheduled. All material. Closed-book

Introduction

Important Note on Background

Knowledge of material in courses on Discrete Math (e.g., COMP 5361 or COMP 232), Combinatorics (e.g., COMP 339) and, of course, Data Structures and Algorithms (e.g., COMP 5511 or COMP 352) are ”formal” prerequisites to this course.

Specifically, you are expected to have a **good working knowledge** of the material such as:

> *sets, relations, functions, logic, proof techniques (e.g., proof by contradition, proof by induction), graph theory, counting techniques, permutations and combinations, basic data structures (e.g., binary search trees, hashing, stacks and queues), sorting.*

It will be difficult to follow or appreciate this course without a proper background preparation.

Computer code = data structure + algorithm

A **data structure** organizes information.

Examples: stacks, lists, arrays, trees, ...

An **algorithm** manipulates information.

Algorithm: a set of well-defined steps for a solution of a problem.

It must be finite, deterministic,
each step is precisely defined,
the order of steps is precisely defined
it must terminate for any input.

Examples: sorting algorithms, search algorithms, etc.

We assume that a problem deals with some input value(s).

Example: Find whether an integer *i* is a prime number.

E.g., find whether 109027492953 is a prime number.

The latter is not a problem, but is a **specific instance** of a problem.

Notice there are typically more than one algorithm for solving a given problem.

Data structures (and some efficient algorithms) are typically adequately covered in an introductory Data Structures and Algorithms course.

The aim of this course is to

- study additional basic **types** of efficient algorithms.
- study how to analyze algorithms (beyond the basic analysis done in an introductory Data Structures and Algorithms course)
- study how to deal with problems for which we have no efficient algorithms.

Measures of efficiency:
time
space
needed for execution of the algorithm as function of the size of input.

What is the size of the input of a problem?

Examples:

Sort n items:

if all items are approximately of the same fixed size,
n is a measure of the input size.

1.54	9.22	3.17	4.19	2.95	4.46	7.61	2.99	3.04

Find whether an integer i is a prime number:

the input size is the number of digits in i.

109027492953

Preliminary Background

How to determine if an algorithm is efficient?

Empirical Methods:
run experiments and measure the time and space.

Analytical Methods:
analyze the structure of an algorithm and derive, using mathematical methods, the time and space needed.

We will deal with analytical methods.

Analytical Approach

Requires arguing about programs without referencing specific hardware, operating system, programming language, etc.

- Abstract machine model
  Random Access Machine
  (by Church-Turing thesis, doesn't really matter)
- Abstract programming language
  Pseudocode

Random Access Machine (RAM)

Components:

- **Single Processor**
  1. Sequential execution of instructions
  2. Instructions (basic instructions have fixed cost):
     - **Arithmetic** (add, subtract, multiply, divide, remainder, floor, ceiling)
     - **Data movement** (load, store, copy)
     - **Control** (conditional branch, subroutine call and return)
- **Random Access Memory**
  1. Unlimited Memory
  2. Each cell stores fixed length word
  3. Access has fixed cost

This model of computation allows us to derive very general conclusions about the efficiency of algorithms. Results are applicable to most actual machines in most cases as a general guideline.

Pseudocode

Simplified way of writing RAM programs

Resembles many modern languages, e.g., C++, Java, Python

To what level of detail is it specified?

Rule of thumb: a person who doesn't know your algorithm but knows C++, Java, or Python should be able to implement it and run it by using only your pseudocode

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = Ø
  for each vertex u ∈ G.V
      INSERT(Q, u)
  while Q ≠ Ø
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
          if v.d changed
              DECREASE-KEY(Q, v, v.d)
```

Asymptotic Notation (§3.1-3.2)

We will describe efficiency in terms of ...

Scalability $\approx$ **Asymptotics**

Counting of time is not exact AND we are interested in order of growth

We express the run-time using the asymptotic notation:

$$f = O(g), f = \Omega(g), f = \Theta(g), f = o(g), f = \omega(g), \ldots$$

O, Θ ,Ω, o, ω are used to express the space and time requirements of algorithms in a concise way.

The counting of time is not exact, so we express the run-time using the **asymptotic notation**

- O-notation: $f(n) = O(g(n))$ if there exists positive constants c, n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- Θ-notation: $f(n) = \Theta(g(n))$ if there exists positive constants c_1, c_2, n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
- Ω-notation: $f(n) = \Omega(g(n))$ if there exists positive constants c, n_0 such that $cg(n) \leq f(n)$ for all $n \geq n_0$.
- o-notation: $f(n) = o(g(n))$ for any positive $c > 0$, if there exists a positive constant n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$.
- ω-notation: $f(n) = \omega(g(n))$ for any positive $c > 0$, if there exists a positive constant n_0 such that $f(n) > cg(n)$ for all $n \geq n_0$.

Interpretation:

$f(n) = O(g(n))$: $f(n)$ grows at most as fast as $g(n)$, $g(n)$ is an upper bound on the growth of $f(n)$.

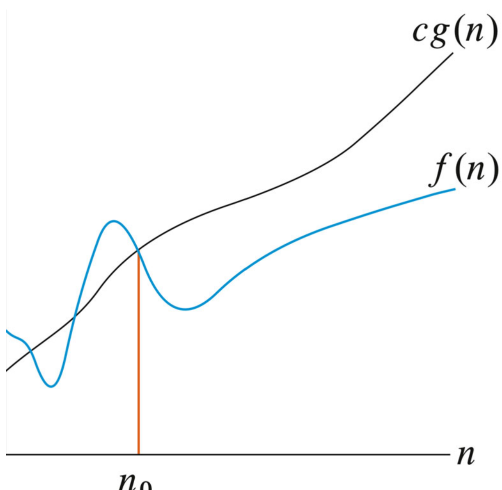
$f(n) = \Omega(g(n))$: $f(n)$ grows at least as fast as $g(n)$, $g(n)$ is a lower bound on the growth of $f(n)$.

$f(n) = \Theta(g(n))$: $f(n)$ grows exactly as fast as $g(n)$.

$f(n) = o(g(n))$: $f(n)$ grows slower than $g(n)$.

$f(n) = \omega(g(n))$: $f(n)$ grows faster than $g(n)$.

Big-O

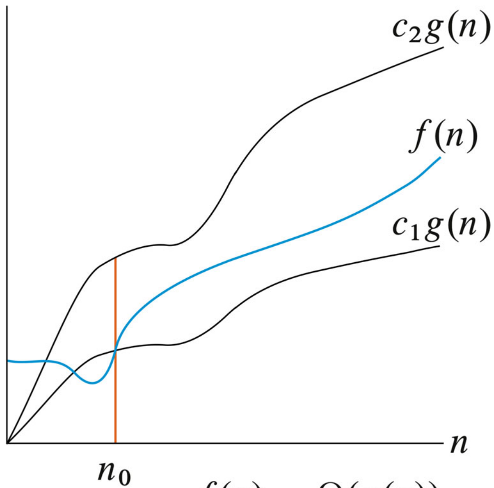


$f(n) = O(g(n))$

$O(g(n)) = \{f(n)$: there exists positive constants c, n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0\}$

"eventually a nontrivial scaled version of $g(n)$ dominates $f(n)$"

Big-Θ



$f(n) = \Theta(g(n))$

$f(n) = \Theta(g(n))$ or alternatively
$f(n) \in \Theta(g(n))$
"f and g have asymptotically similar growth"
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Examples

$2n^2 + 5n - 6 = O(2^n)$
$2n^2 + 5n - 6 = O(n^3)$
$2n^2 + 5n - 6 = O(n^2)$
$2n^2 + 5n - 6 \neq O(n)$

$2n^2 + 5n - 6 = o(2^n)$
$2n^2 + 5n - 6 = o(n^3)$
$2n^2 + 5n - 6 \neq o(n^2)$
$2n^2 + 5n - 6 \neq o(n)$

$2n^2 + 5n - 6 \neq \Omega(2^n)$
$2n^2 + 5n - 6 \neq \Omega(n^3)$
$2n^2 + 5n - 6 = \Omega(n^2)$
$2n^2 + 5n - 6 = \Omega(n)$

$2n^2 + 5n - 6 \neq \omega(2^n)$
$2n^2 + 5n - 6 \neq \omega(n^3)$
$2n^2 + 5n - 6 \neq \omega(n^2)$
$2n^2 + 5n - 6 = \omega(n)$

$2n^2 + 5n - 6 \neq \Theta(2^n)$
$2n^2 + 5n - 6 \neq \Theta(n^3)$
$2n^2 + 5n - 6 = \Theta(n^2)$
$2n^2 + 5n - 6 \neq \Theta(n)$

Simplifications of expressions in asymptotic analysis

c is a constant:

$$O(f(n) + c) = O(f(n))$$

$$O(cf(n)) = O(f(n))$$

If $f_1 = O(f_2)$:

$$O(f_1(n) + f_2(n)) = O(f_2(n))$$

If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then

$$O(f_1(n) * f_2(n)) = O(g_1(n) * g_2(n))$$

If

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

then $g(n) = O(f(n))$ and $f(n) = \Omega(g(n))$.

If

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

then $g(n) = \Omega(f(n))$ and $f(n) = O(g(n))$

If

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$$

where c is a nonzero constant
then $f(n) = \Theta(g(n))$ and $g(n) = \Theta(f(n))$.

Algorithmic Analysis

n - size of input
$T(n)$ - running time on inputs of length n

Worst case analyses:
$T(n) =$ the longest running time (space) for any input of length n.

Average case analyses (expected):
$T(n) =$ the running time (space) averaged over all inputs of length n.

This is the most useful information, but often difficult to get.

It also may involve assumptions on whether all cases are occurring equally often, etc.

Best case analyses:
$T(n) =$ the shortest running time (space) for any input of length n.

Example: Insertion-Sort (§3.1)

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

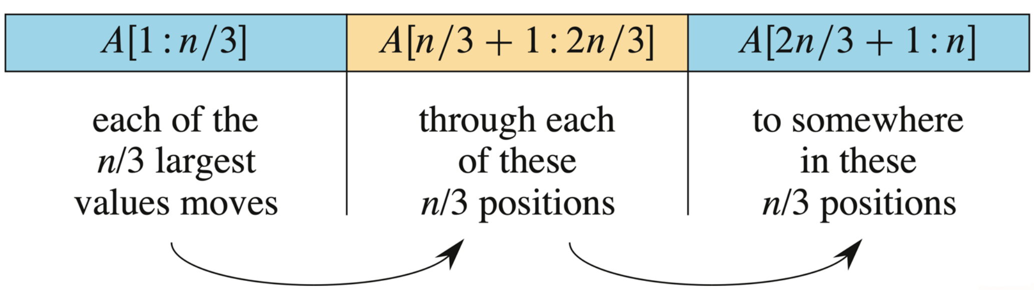
First, show that INSERTION-SORT is runs in $O(n^2)$ time, regardless of the input (of size n):

- The outer **for** loop runs $n-1$ times regardless of the values being sorted.
- The inner **while** loop iterates at most $i-1$ times.
- The exact number of iterations the **while** loop makes depends on the values it iterates over, but it will definitely iterate between 0 and $i-1$ times.
- Since i is at most n, the total number of iterations of the inner loop is at most $(n-1)(n-1)$, which is less than n^2.

Each inner loop iteration takes constant time, for a total of at most cn^2 for some constant c, or $O(n^2)$.

Now show that INSERTION-SORT has a worst-case running time of $\Omega(n^2)$:

- Observe that for a value to end up k positions to the right of where it started, the line $A[j+1] = A[j]$ must have been executed k times.
- Assume that n is a multiple of 3 so that we can divide the array A into groups of $n/3$ positions.



Because at least $n/3$ values must pass through at least $n/3$ positions, the line $A[j+1] = A[j]$ executes at least $(n/3)(n/3) = n^2/9$ times, which is $\Omega(n^2)$. For this input, INSERTION-SORT takes time $\Omega(n^2)$.

Since we have shown that INSERTION-SORT runs in $O(n^2)$ time in all cases and that there is an input that makes it take $\Omega(n^2)$ time, we can conclude that the worst-case running time of INSERTION-SORT is $\Theta(n^2)$.

Expressed using the definitions of the asymptotic notations, for the worst-case running time $T(n)$ for INSERTION-SORT,

there exists positive constants c_1, c_2, n_0 such that $c_1 n^2 \leq T(n) \leq c_2 n^2$ for all $n \geq n_0$

The constant factors for the upper and lower bounds may differ. That does not matter.

Material to Review

Functions (injective, surjective, bijective, partial, total, ...)

Sets and operations on sets

Relations (equivalence relations)

Basic proof techniques: induction, contradiction, pigeonhole principle

Algorithmic Recurrences

A recurrence is used to characterize the running time of a recursive algorithm. Solving the recurrence gives us the asymptotic running time.

A recurrence is a function is defined in terms of one or more base cases, and itself, with smaller arguments.

Interested in recurrences that describe running times of algorithms. A recurrence $T(n)$ is **algorithmic** if for every sufficiently large **threshold** constant $n_0 > 0$:

- For all $n < n_0$, $T(n) = \Theta(1)$. [*Can consider the running time constant for small problem sizes.*]
- For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations. [*The recursive algorithm terminates.*]

Conventions

Will often state recurrences without base cases. When analyzing algorithms, assume that if no base case is given, the recurrence is algorithmic.

Some recurrences are inequalities rather than equations.

Example:

$$T(n) \leq 2T(n/2) + \Theta(n)$$

gives only an upper bound on $T(n)$, so state the solution using O-notation rather than Θ-notation.

Conventions, cont.

Ceilings and floors in divide-and-conquer recurrences don't change the asymptotic solution $\Rightarrow$ often state algorithmic recurrences without floors and ceilings.

Example:

The algorithm merge sort breaks a problem of size n into two subproblems of size $n/2$, taking $\Theta(n)$ time to divide and then combine/merge the sorted subproblems.

Typically, the recurrence for merge sort is written:

$$T(n) = 2T(n/2) + \Theta(n)$$

But, the recurrence for merge sort is really

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

Examples of Recurrences

- An algorithm that breaks a problem of size n into one subproblem of size $n/3$ and another of size $2n/3$, taking $\Theta(n)$ time to divide and combine: $T(n) = T(n/3) + T(2n/3) + \Theta(n)$. Solution: $T(n) = \Theta(n \lg n)$.
- An algorithm that breaks a problem of size n into one problem of size $n/5$ and another of size $7n/10$, taking $\Theta(n)$ time to divide and combine: $T(n) = T(n/5) + T(7n/10) + \Theta(n)$. Solution: $T(n) = \Theta(n)$.
- Subproblems do not always have to be a constant fraction of the original problem size. Example: recursive linear search creates one subproblem and it has one element less than the original problem. Time to divide and combine in $\Theta(1)$, giving $T(n) = T(n-1) + \Theta(1)$. Solution: $T(n) = \Theta(n)$.

Methods for Solving Recurrences

The textbook contains four methods for solving recurrences. Each gives asymptotic bounds.

- Substitution method (§4.3): Guess the solution, then use induction to prove that it's correct.
- Recursion-tree method (§4.4): Draw out a recursion tree, determine the costs at each level, and sum them up. Useful for coming up with a guess for the substitution method.
- Master method (§4.5): A cookbook method for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a > 0$ and $b > 1$ are constants, subject to certain conditions. Requires memorizing three cases, but applies to many divide-and-conquer algorithms.
- Akra-Bazzi method (§4.7): A general method for solving divide-and-conquer recurrences. Requires calculus, but applies to recurrences beyond those solved by the master method.

Substitution Method (§4.3)

1. Guess the solution
2. Use induction to find the constants and show that the solution works

Example:

Determine an asymptotic upper bound on $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$.
Floor function ensures that $T(n)$ is defined over integers.

Guess: $T(n) = O(n \lg n)$

Inductive step: Assume that $T(n) \le cn \lg n$ for all numbers $\ge n_0$ and $< n$.
If $n/2 \ge n_0$, holds for $\lfloor n/2 \rfloor \Rightarrow T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$.

Substitute into the recurrence:

$$\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + \Theta(n) \\
&\le 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + \Theta(n) \\
&\le 2(cn/2 \lg(n/2)) + \Theta(n) \\
&= cn \lg(n/2) + \Theta(n) \\
&= cn \lg n - cn \lg 2 + \Theta(n) \\
&= cn \lg n - cn + \Theta(n) \\
&\le cn \lg n \qquad \text{if } cn \ge \Theta(n)
\end{aligned}$$

Base cases: Need to show that $T(n) \le cn \lg n$ when $n_0 \le n < 2n_0$. Add new constraint: $n_0 > 1 \Rightarrow \lg n > 0 \Rightarrow n \lg n > 0$. Pick $n_0 = 2$. Because no base case is given in the recurrence, it's algorithmic $\Rightarrow T(2), T(3)$ are constant. Choose $c = \max\{T(2), T(3)\} \Rightarrow T(2) \le c < (2 \lg 2)c$ and $T(3) \le c < (3 \lg 3)c \Rightarrow$ inductive hypothesis established for the base cases.

Wrap up: Have $T(n) \le cn \lg n$ for all $n \ge 2 \Rightarrow T(n) = O(n \lg n)$.

In practice

Don't usually write out substitution proofs this detailed, especially regarding base cases. For most algorithmic recurrences, the base cases are handled the same way.

No general way to make a good guess. Experience helps. Approaches such as recursion trees help guide finding possible solutions.

When the additive term uses asympotic notation (e.g., $\Theta(n)$)

- Name the constant in the additive term (e.g., $\Theta(n) \to cn$)
- Show the upper (O) and lower (Ω) bounds separately. Might need to use different constants for each.

Example:
$T(n) = 2T(n/2) + \Theta(n)$. If we want to show an upper bound of $T(n) = 2T(n/2) + O(n)$, we write $T(n) \leq 2T(n/2) + cn$ for some possible constant c.

Important: We get to name the constant hidden in the asymptotic notation (c in this case), but we do **not** get to choose it, other than assume that it's enough to handle the base case of the recursion.

Upper bound:

Guess: $T(n) \leq dn \lg n$ for some positive constant d. This is the inductive hypothesis.

Important

We get to both name and choose the constant in the inductive hypothesis (d in this case). It OK for the constant in the inductive hypothesis (d) to depend on the constant hidden in the asymptotic notation (c).

Substitution:

$$
\begin{aligned}
T(n) &\leq 2T(n/2) + cn \\
&\leq 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\
&= dn\lg\frac{n}{2} + cn \\
&= dn\lg n - dn + cn \\
&\leq dn\lg n \\
&\quad \text{if } -dn + cn \leq 0, \\
&\quad \Rightarrow d \geq c
\end{aligned}
$$

Therefore, $T(n) = O(n \lg n)$.

Lower bound:

Write: $T(n) \geq 2T(n/2) + cn$ for some positive constant c.

Guess: $T(n) \geq dn \lg n$ for some positive constant d.

Substitution:

$$\begin{aligned} T(n) &\geq 2T(n/2) + cn \\ &\geq 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\geq dn\lg n \\ &\quad \text{if } -dn + cn \geq 0 \Rightarrow d \leq c \end{aligned}$$

Therefore, $T(n) = \Omega(n \lg n)$.

Therefore, $T(n) = \Theta(n \lg n)$. [For this particular recurrence, we can use $d = c$ for both the upper-bound and lower-bound proofs. That won't always be the case.]

Subtracting a low-order term

Might guess the right asymptotic bound, but the math doesn't go through in the proof. Resolve by subtracting a lower-order term.

Example:

$T(n) = 2T(n/2) + \Theta(1)$. Guess that $T(n) = O(n)$, and try to show $T(n) \leq cn$ for $n \geq n_0$, where we choose c, n_0:

$$\begin{aligned} T(n) &\leq 2(c(n/2)) + \Theta(1) \\ &= cn + \Theta(1) \end{aligned}$$

But this doesn't say that $T(n) \leq cn$ for any choice of c.

Subtracting a low-order term, cont.

Could try a larger guess, such as $T(n) = O(n^2)$, but not necessary. We are off by $\Theta(1)$, a lower-order term. Try subtracting a lower-order term in the guess: $T(n) \leq cn - d$, where $d \geq 0$ is a constant:

$$\begin{aligned} T(n) &\leq 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &= cn - d - (d - \Theta(1)) \\ &\leq cn - d \end{aligned}$$

as long as d is larger than the constant in $\Theta(1)$.

Subtracting a low-order term, cont.

Why subtract off a lower-order term, rather than add it? Notice that it's subtracted twice. Adding a lower-order term twice would take us further away from the inductive hypothesis. Subtracting it twice gives us $T(n) \leq cn - d - (d - \Theta(1))$, and it's easy to choose d to make that inequality hold.

Important: Once again, we get to name and choose the constant c in the inductive hypothesis. And we also get to name and choose the constant d that we subtract off.

Subtracting a low-order term, cont.

Be careful when using asymptotic notation

A false proof for the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$, that $T(n) = O(n)$:

$$\begin{aligned} T(n) &\le 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n)\ . \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

This “proof” changes the constant in the Θ-notation. Can see this by using an explicit constant. Assume $T(n) \le cn$ for all $n \ge n_0$:

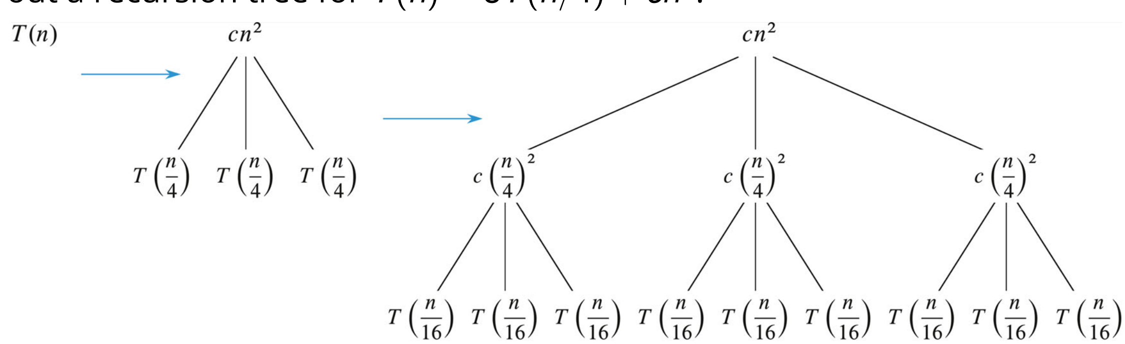
$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ &= cn + \Theta(n), \end{aligned}$$

but this is not $\le cn$ since $cn + \Theta(n) > cn$.

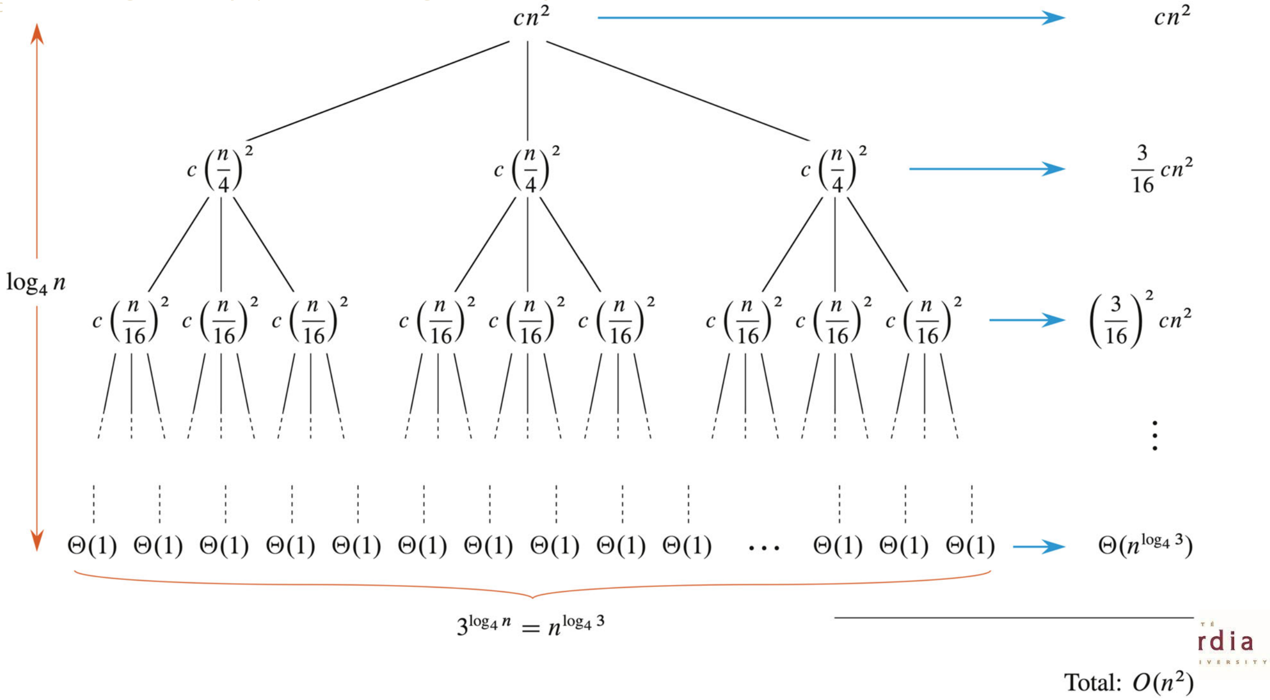
Recursion Trees (§4.4)

Used to generate a guess. Then verify by substitution method.

Example: $T(n) = 3T(n/4) + \Theta(n^2)$
Draw out a recursion tree for $T(n) = 3T(n/4) + cn^2$.



For simplicity: Assume that n is a power of 4
and the base case is $T(1) = \Theta(1)$.



Subproblem size for nodes at depth i is $n/4^i$. Get to base case when

$$n/4^i = 1 \Rightarrow n = 4^i \Rightarrow i = \log_4 n.$$

Each level has 3 times as many nodes as the level above, so that depth i has 3^i nodes. Each internal node at depth i has cost $c(n/4^i)^2$

$$\Rightarrow \text{total cost at depth } i \text{ (except for leaves) is } 3^i c(n/4^i)^2 = (3/16)^i cn^2.$$

Bottom level has depth $\log_4 n \Rightarrow$ number of leaves is $3^{\log_4 n} = n^{\log_4 3}$. Since each leaf contributes $\Theta(1)$, total cost of leaves is $\Theta(n^{\log_4 3})$.

Add up costs over all levels to determine cost for the entire tree:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2).
\end{aligned}$$

Idea: Coefficients of cn^2 form a decreasing geometric series. Bound it by an infinite series, and get a bound of $16/13$ on the coefficients.

Use substitution method to verify $O(n^2)$ upper bound. Show that $T(n) \leq dn^2$ for constant $d > 0$:

$$\begin{aligned}
T(n) &\leq 3T(n/4) + cn^2 \\
&\leq 3d(n/4)^2 + cn^2 \\
&= \frac{3}{16} dn^2 + cn^2 \\
&\leq dn^2,
\end{aligned}$$

by choosing $d \geq (16/13)c$. [*Again, we get to name but not choose c, and we get to name and choose d.*]

That gives an upper bound of $O(n^2)$. The lower bound of $\Omega(n^2)$ is obvious since the recurrence relation contains a $\Theta(n^2)$ term. Hence, $T(n) = \Theta(n^2)$.

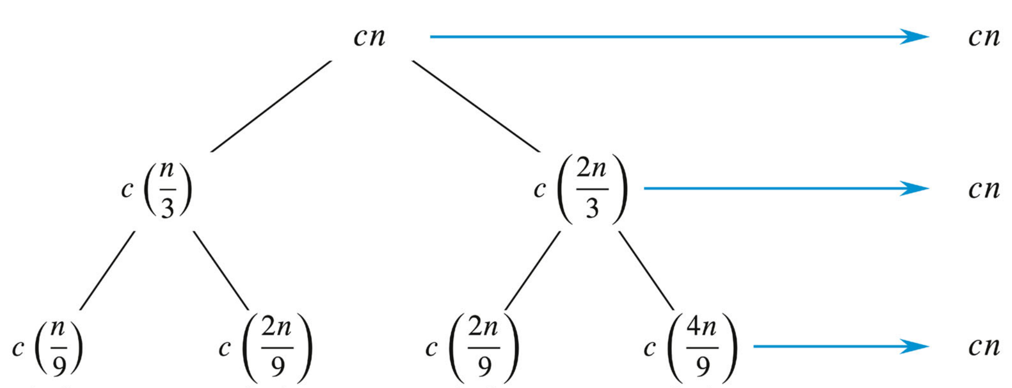
Irregular Example:

$T(n) = T(n/3) + T(2n/3) + \Theta(n)$.

Inductive hypothesis: $T(n) \leq cn \lg n$ for all $n \geq n_0$. Will choose constants $c, n_0 > 0$ later, once we know their constraints.

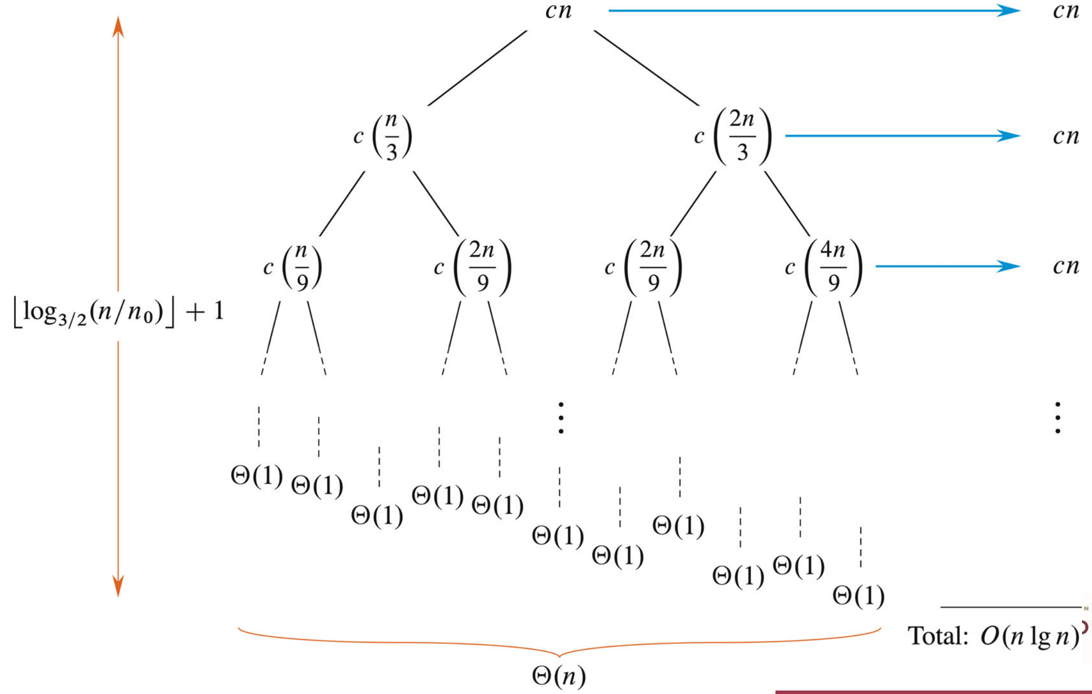
For upper bound, rewrite as $T(n) \leq T(n/3) + T(2n/3) + cn$; for lower bound, as $T(n) \geq T(n/3) + T(2n/3) + cn$.

By summing across each level, the recursion tree shows the cost at each level of recursion (minus the costs of recursive calls, which appear in subtrees):



The leftmost branch reaches $n = 1$ after $\log_3 n$ levels whereas the rightmost branch reaches $n = 1$ after $\log_{3/2} n$ levels.

So, the full recurrence tree looks like:



- There are $\log_3 n$ full levels (going down the left side), and after $\log_{3/2} n$ levels, the problem size is down to 1 (going down the right side).
- Each level contributes $\leq cn$.
- Lower bound guess: $\geq dn \log_3 n = \Omega(n \lg n)$ for some positive constant d.
- Upper bound guess: $\leq dn \log_{3/2} n = O(n \lg n)$ for some positive constant d.
- Then *prove* by substitution.

Expansion Method [not in CLRS]

Another approach to guessing the functional form of the running time is the Expansion Method (also called Iterative/Repeated Substitution, Unfolding methods, or Plug-n-Chug).

Step 1: Substitute for $T(\cdot)$ a few times in the right side of the recurrence relation to determine a pattern

Step 2: Guess the recurrence formula after k substitutions (in terms of k and n)

For each *base case*:

Step 3: Solve for k

Step 4: Plug the solution for k back into the formula from Step 2 to find a potential (perhaps wrong) closed form

Step 5: Prove the potential closed form is correct by substitution.

For example, $T(n) = 2T(n/2) + n/2$, $T(1) = 1$

Step 1: Substitute for $T(\cdot)$ a few times

$k = 1$:

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

$k = 2$:

$$\begin{aligned} T(n) &= 2(2T(n/4) + n/4) + n/2 \\ &= 2^2 T(n/4) + n/2 + n/2 \\ &= 2^2 T(n/4) + n \end{aligned}$$

$k = 3$:

$$\begin{aligned} T(n) &= 2(2T(n/8) + n/8) + n \\ &= 2^3 T(n/8) + n/2 + n \\ &= 2^3 T(n/8) + 3n/2 \end{aligned}$$

Step 2: Guess the recurrence formula

kth substitution:

$$T(n) = 2^k T(n/2^k) + kn/2$$

Step 3: Set k so that we get the base case

Let $n/2^k = 1$ such that $T(n/2^k) = T(1) = 1$. Then $2^k = n$ or

$$k = \log_2 n$$

Step 4: Plug k back into the formula

$$\begin{aligned} T(n) &= 2^{\log_2 n} T(n/2^{\log_2 n}) + (\log_2 n)n/2 \\ &= n + (\log_2 n)n/2 \end{aligned}$$

Step 5: Prove potential closed form using substitution

Master Theorem (§4.5)

Theorem 4.1 (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 n/b can be $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bound 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 $f(n) = \Theta(n^{log_b a} \lg^k n)$, where $k \geq 0$ is a constant, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$
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 $T(n) = \Theta(f(n))$

Master Theorem: Examples

Use the master theorem to solve

- $T(n) = 7\ T(n/2) + n^2$ $\boxed{f(n) < n^{\log_b a}}$

  $n^{\log_2 7}$ vs. n^2

  $\log_2 7 = 2.81 \Rightarrow 2.81 - \epsilon = 2$ for some $\epsilon > 0$

  So, case 1 applies and $T(n) = \Theta(n^{\log_2 7})$

- $T(n) = 27\ T(n/3) + n^3 \lg n$ $\boxed{f(n) \text{ is a polylog of } n^{\log_b a}}$

  $n^{\log_3 27} = n^3$ vs. $n^3 \lg n$

  So, case 2 applies with $k = 1$ and $T(n) = \Theta(n^3 \lg^2 n)$

Master Theorem: Examples, cont.

Use the master theorem to solve

- $T(n) = 7\ T(n/3) + n^2$ $\boxed{f(n) > n^{\log_b a}}$

  $n^{\log_3 7}$ vs. n^2

  $\log_3 7 = 1.8 \Rightarrow 1.8 + \epsilon = 2$ for some $\epsilon > 0$

  So, case 3 apparently applies.

  But need to check **regularity condition**:

  $af(n/b) = 7f(n/3) = 7n^2/9 \leq cf(n) = cn^2$ for some constant $c \geq 7/9$

  So, case 3 applies and $T(n) = \Theta(n^2)$

Akra-Bazzi Method (§4.7)

Akra-Bazzi recurrences take the form

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(n/b_i)$$

where k is a positive integer, all the constants $a_1, a_2, \ldots, a_k \in \mathcal{R}$ are strictly positive, all the constants $b_1, b_2, \ldots, b_k \in \mathcal{R}$ are strictly greater than 1, and the driving function $f(n)$ is defined on sufficiently large nonnegative reals and is itself nonnegative.

Whereas for the master theorem deals with equal-sized subproblems, Akra-Bazzi recurrences allow for different-sized subproblems.

We are going to ignore the issue of floors and ceilings in Akra-Bazzi recurrences. Most driving functions behave nicely and this is not an issue.

The Akra-Bazzi method was developed to solve Akra-Bazzi recurrences.

The method involves first determining the unique real number p such that

$$\sum_{i=1}^{k} a_i \left(\frac{1}{b_i}\right)^p = 1$$

Such a p always exists, because when $p \to -\infty$, the sum goes to ∞, it decreases as p increases; and when $p \to \infty$, it goes to 0.

The Akra-Bazzi method then gives the solution to an Akra-Bazzi recurrence as

$$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{f(x)}{x^{p+1}} dx\right)\right)$$

Examples

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + \Theta(n)$$

From

$$\left(\frac{1}{5}\right)^p + \left(\frac{4}{5}\right)^p = 1$$

we get $p = 1$. Then

$$\begin{aligned} T(n) &= \Theta\left(n^p\left(1 + \int_1^n \frac{f(x)}{x^{p+1}}dx\right)\right) \\ &= \Theta\left(n\left(1 + \int_1^n \frac{1}{x}dx\right)\right) \\ &= \Theta(n \ln n) \end{aligned}$$

Introduction Background Asymptotic Notation Algorithmic Analysis Recurrence Relations Master Theorem Akra-Bazzi Method

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$$

From $\left(\frac{1}{5}\right)^p + \left(\frac{7}{10}\right)^p = 1$ we get $p < 1$. Then

$$\begin{aligned} T(n) &= \Theta\left(n^p\left(1 + \int_1^n \frac{f(x)}{x^{p+1}}dx\right)\right) \\ &= \Theta\left(n^p\left(1 + \int_1^n \frac{1}{x^p}dx\right)\right) \\ &= \Theta\left(n^p\left(1 + \left[\frac{1}{(1-p)x^{p-1}}\right]_1^n\right)\right) \\ &= \Theta\left(n^p\left(1 + \left(\frac{1}{(1-p)n^{p-1}} - \frac{1}{1-p}\right)\right)\right) \\ &= \Theta\left(\left(\frac{1}{1-p}\right)n - \left(\frac{p}{1-p}\right)n^p\right) \\ &= \Theta(n) \end{aligned}$$

Note on using the master theorem and Akra-Bazzi method

If you use the Expansion Method or Recursion trees to determine a (best guess) running time for a recurrence relation, you still have to prove it is correct using the substitution method.

But if you determine the running time for a recurrence relation using the master theorem or Akra-Bazzi method, you **do not** prove it is correct (e.g., by substitution)! That proof of correctness was already done in the development of the master theorem and Akra-Bazzi method. Learn how to use these solution methods correctly – that is all you need.