CS200: Recurrence Relations and the Master Theorem

Rosen Ch. 8.1 - 8.3

Recurrence Relations: An Overview

- What is a recurrence relation?
  - **A recursively defined sequence**
- Example
  - Arithmetic progression: $a, a+d, a+2d, \ldots, a+nd$
    - $a_0 = a$
    - $a_n = a_{n-1} + d$

Formal Definition

A recurrence relation for the sequence $\{a_n\}$ is an equation that expresses a_n in terms of one of more of the previous terms of the sequence, namely, $a_0, a_1, \ldots a_{n-1}$, for all integers n with $n \geq n_0$ where n_0 is a nonnegative integer.

- Sequence = Recurrence relation + Initial conditions ("*base case*")
- Example: $a_n = 2a_{n-1} + 1, \; a_1 = 1$
- *Pg. 158 in Rosen*

Compound Interest

- You deposit $10,000 in a savings account that yields 10% yearly interest. How much money will you have after 30 years? (b is balance, r is rate)

$$b_n = b_{n-1} + rb_{n-1} = (1+r)^n b_0$$

Modeling with Recurrence

CS200

- Suppose that the number of bacteria in a colony triples every hour
  - Set up a recurrence relation for the number of bacteria after n hours have elapsed.
  - 100 bacteria are used to begin a new colony.

Recursively defined functions and recurrence relations

CS200

- **A recursive function**
  $f(0) = a$ (base case)
  $f(n) = f(n\text{-}1) + d$ for $n > 0$ (recursive step)
- The above recursively defined function generates the sequence
  $a_0 = a$
  $a_n = a_{n-1} + d$
- A recurrence relation **produces a sequence**, an application of a recursive function produces **a value from the sequence**

How to Approach Recursive Relations

CS200

Recursive Functions ⬌ Sequence of Values

$f(0) = 0$ (base case)
$f(n) = f(n\text{-}1) + 2$ for $n > 0$
(recursive part)

$f(0) = 0$
$f(1) = f(0) + 2 = 2$
$f(2) = f(1)+2 = 4$
$f(3) = f(2) + 2 = 6$

Closed Form?(solution, explicit formula)

Find a recursive function

CS200

- Give a recursive definition of a^n, where *a* is a nonzero real number and *n* is a nonnegative integer.

- Rosen Chapter 5 example 3-2 pp. 346

Solving recurrence relations

Solve $a_0 = 2; a_n = 3a_{n-1}, n > 0$

(1) What is the recursive function?

(2) What is the sequence of values?

Hint 1: Solve by substitution

- $a_0 = 2; a_1=3(2)=6; a_2=3(a_1)=3(3(2)); a_3=\ldots$

Use a formula

Hint 2: Use known formula

- $a_n=r^n$ (where r is a constant) is a solution for

  $a_n=c_1a_{n-1}+c_2a_{n-2}+\ldots+c_ka_{n-k}$
- summing geometric series

$$1 + r + r^2 + \ldots + r^n = (r^{n+1} - 1)/(r - 1) \quad \text{if } r \neq 1$$

Linear Recurrence Relations

A linear homogeneous recurrence relation of degree *k* with constant coefficients is a recurrence relation of a form

$a_n = c_1a_{n-1} + c_2a_{n-2}+ \ldots + c_ka_{n-k}$

where, $c_1, c_2, c_3 \ldots, c_k$ are real numbers and c_k is not 0.

Requirements

- Linear: RHS is a sum of previous terms
- Homogeneous: no terms occur that are not multiples of LHS
- Coefficients must be constants

Is this a linear homogeneous recurrence relation?

1. $f_n = f_{n\text{-}1} + f_{n\text{-}2}$
2. $a_n = a_{n\text{-}1} + a^2_{n\text{-}2}$
3. $H_n = 2H_{n\text{-}1} + 1$

Linear homogeneous recurrence relations?

A. 1 B. 2 C. 3 D. 1,2,3

- Modeling of problems
- They can be systematically solved.

Theorem 1

Let c_1 and c_2 be real numbers. Suppose that

$r^2 - c_1 r - c_2 = 0$

has two distinct roots r_1 and r_2. Then the sequence $\{a_n\}$ is a solution of the recurrence relation

$a_n = c_1 a_{n\text{-}1} + c_2 a_{n\text{-}2}$

if and only if

$a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ for $n = 0,1,2,\ldots$

where α_1 and α_2 are constants.

Rosen section8.2 theorem 1, pp.515

What is the solution of the recurrence relation:

$a_n = a_{n\text{-}1} + 2a_{n\text{-}2}$ with $a_0 = 2$ and $a_1 = 7$?

From Theorem 1: Given $a_n = c_1 a_{n\text{-}1} + c_2 a_{n\text{-}2}$

$r^2 - c_1 r - c_2 = 0$

Characteristic equation is $r^2 - 1r - 2$; roots are 2 and -1.

Iff $a_n = \alpha_1 2^n + \alpha_2 (-1)^n$

From initial conditions,

$a_0 = 2 = \alpha_1 * 1 + \alpha_2 * 1$ $\quad a_1 = 7 = \alpha_1 * 2 + \alpha_2 * (-1)$

If $\alpha_1 = 3$, then $\alpha_2 = -1$ and $a_n = 3*2^n - (-1)^n$

Rosen Section 8-2 Example 3 pp. 516

Divide-and-Conquer

Basic idea:

Take large problem and **divide** it into **smaller problems** until problem is trivial, then **combine** parts to make solution.

Recurrence relation for the number of steps required:

$$f(n) = a\, f(n / b) + g(n)$$

n/b : the size of the sub-problems solved

a : number of sub-problems

$g(n)$: steps necessary to combine solutions to sub-problems

Example: Binary Search

```
public static int binSearch (int myArray[], int first,
                              int last, int value) {
   // returns the index of value or -1 if not in the array
   int index;
   if (first > last) { index = -1; }
   else {
      int mno = (first + last)/2;
      if (value == myArray[mno]) { index = mno; }
      else if (value < myArray[mno]) {
             index = binSearch(myArray, first, mno-1, value);
      }
         else {
         index = binSearch(myArray, mno+1, last, value);
      }
   }
   return index;
}
```

What are a, b, and g(n)?

$$f(n) = a \cdot f(n/b) + g(n)$$

Estimating big-O (Master Theorem)

Let f be an increasing function that satisfies

$f(n) = a \cdot f(n/b) + c \cdot n^d$

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer > 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

From section 8.3 in Rosen

Binary Search using the Master Theorem

For binary search

$f(n) = a\, f(n / b) + n^d$

$\quad = 1\, f(n / 2) + 3$

$$f(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Therefore, $d = 0$ *(to make* n^d *a constant)*, $b = 2$, $a = 1$.

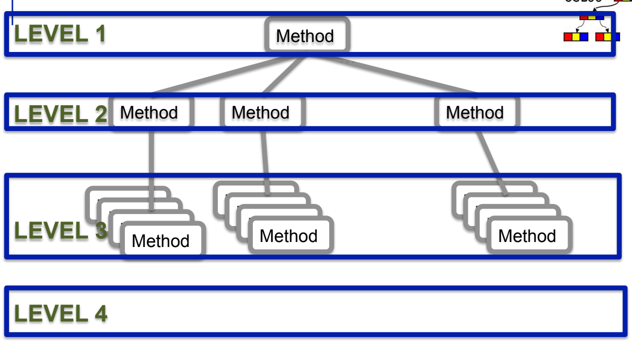
$b^d = 2^0 = 1$

It satisfies the second condition of the Master theorem.

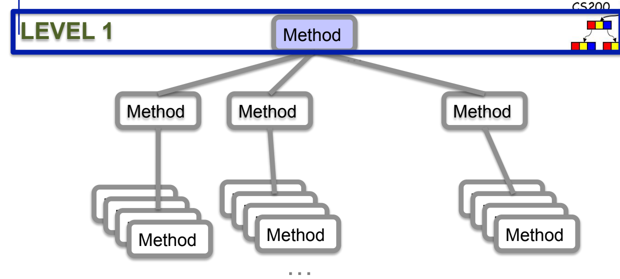
So, $f(n) = O(n^d \log_2 n) = O(n^0 \log_2 n) = \boldsymbol{O(\log_2 n)}$

Step-by-step Master Theorem

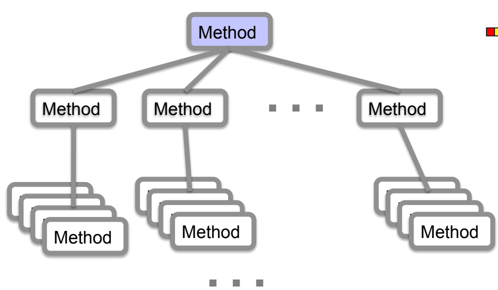
- Modeling divide-and-conquer with a recurrence relation.
- $f(n)$ is a recurrence function representing the number of operations with size of input n.



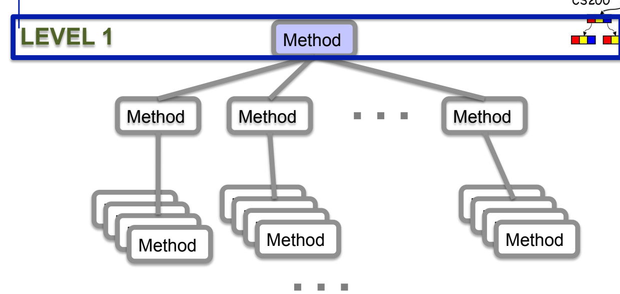
Goal 1: What is the complexity per level?
Goal 2: What is the total complexity?



"**Method**" is called with input size n.
"**Method**" calls "**Method**" recursively a times
"**Method**" calls "**Method**" recursively with input size n/b

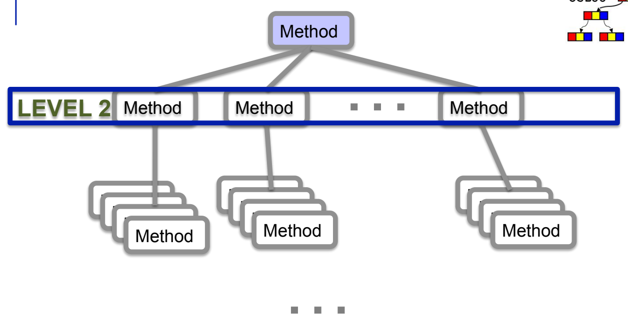


"**Method**" includes software fragment on top of calling methods recursively, and it takes $g(n)$ operations.



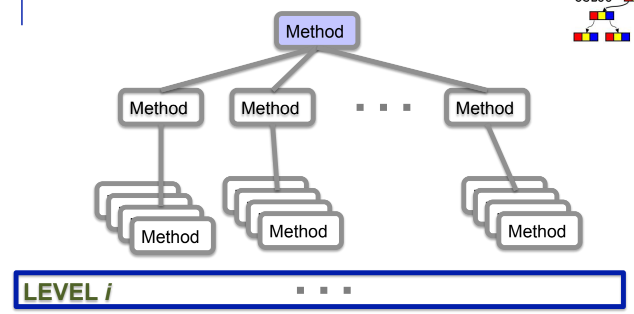
Therefore, in LEVEL 1, "**Method**" requires,

$$f(n) = a\, f(n / b) + g(n)$$



This is also true for LEVEL 2

$f(n) = a\, f(n / b) + g(n)$



This is also true for LEVEL i

$f(n) = a\, f(n / b) + g(n)$

More generally,

Let f be an increasing function that satisfies

$f(n) = af(n/b) + g(n) = af(n/b) + cn^d$

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer > 1, and c and d are real numbers with c positive and d nonnegative.

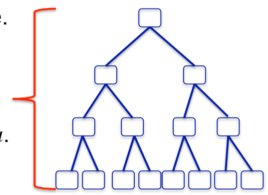
$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Base case

- We will set $d = 1$
  - The bottom level of the tree is equally well computed.
  - Base case
  - It is straightforward to extend the proof for the case when $d \neq 1$.

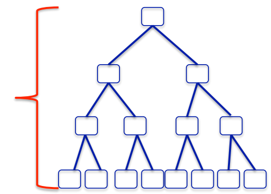
Goal 1. complexity per level

- Let's think about the recursion tree.
- There will be $log_b n$ levels.
- At each level, the number of subproblems will be multiplied by a.
- Therefore, the number of subproblems at level i will be a^i.
- Each subproblem at level i is a problem of size (n/b^i).
- A subproblem of size (n/b^i) requires $(n/b^i)^d$ additional work.



Total amount of work on level *i*

$$a^i(n/b^i)^d = n^d\,(a^i/b^{di})$$
$$= n^d\,(a/b^d)^i$$



For the level i, the work per level is decreasing, constant, or increasing exactly when $(a/b^d)^i$ is decreasing, constant, or increasing.

Observation

- Now, observe that,

  $(a/b^d) = 1$

  $a = b^d$

- Therefore, the relations,

(1) $a < b^d$ (2) $a = b^d$, (3) $a > b^d$

are the conditions deciding the types of the growth function

Goal 2: Bounding $f(n)$ in the different cases.

In general, we have that the total work done is,

$$\sum_{i=0}^{\log_b n} n^d (a/b^d)^i = n^d \sum_{i=0}^{\log_b n} (a/b^d)^i$$

1. $a < b^d$
2. $a = b^d$
3. $a > b^d$

Case 1. $a < b^d$

$$\sum_{i=0}^{\log_b n} n^d (a/b^d)^i = n^d \sum_{i=0}^{\log_b n} (a/b^d)^i$$

- n^d times a geometric series with a ratio of less than 1.
- First item is the biggest one.

$$n^d \sum_{i=0}^{\log_b n} (a/b^d)^i = O(n^d)$$

Case 2. $a = b^d$

$$\sum_{i=0}^{\log_b n} n^d (a/b^d)^i = n^d \sum_{i=0}^{\log_b n} (a/b^d)^i$$

- $(a/b^d) = 1$
- $n^d(1+1+1+\ldots+1) = n^d(\log_b n)$

$$n^d \sum_{i=0}^{\log_b n} (a/b^d)^i = O(n^d \log_b n)$$

Case 3. $a > b^d$

$$\sum_{i=0}^{\log_b n} n^d (a/b^d)^i = n^d \sum_{i=0}^{\log_b n} (a/b^d)^i$$

$(a/b^d) > 1$. Therefore the largest term is the last one.

$$n^d (a/b^d)^{\log_b n} = n^d (a^{\log_b n} / (b^d)^{\log_b n})$$

$$= n^d (n^{\log_b a} / n^{\log_b b^d})$$

$$= n^d (n^{\log_b a} / n^d)$$

$$= n^{\log_b a}$$

$$n^d \sum_{i=0}^{\log_b n} (a/b^d)^i = O(n^{\log_b a})$$

Complexity of MergeSort with Master Theorem

- Mergesort splits a list to be sorted twice per level.
- Uses fewer than n comparisons to merge the two sorted lists of $n/2$ items each into one sorted list.
- Function $M(n)$ satisfies the divide-and-conquer recurrence relation
- $M(n) = 2M(n/2) + n$

Complexity of MergeSort with Master Theorem (2/2)

$M(n) = 2M(n/2) + n$
for the mergesort algorithm

$f(n) = a\,f(n/b) + n^d$
$\quad = 2\,f(n/2) + n^1$

Therefore, $d = 1,\ b = 2,\ a = 2.$
$b^d = 2^1 = 2$

$$f(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

It satisfies the second condition of the Master theorem.
So, $f(n) = O(n^d \log_2 n)$
$\quad = O(n^1 \log_2 n)$
$\quad = \mathbf{O(n \log_2 n)}$

quickSort: Recurrence Analysis

$$f(n) = a \cdot f(n/b) + cn^d$$

- a=
- b=
- c=
- d=
- O(?)

Tractability

- A problem that is solvable using an algorithm with **polynomial worst-case complexity** is called **tractable.**
- If estimation has high degree or if the coefficients are extremely large, the algorithm may take an extremely long time to solve the problem.

Intractable problems and Unsolvable problems

- If the problem *cannot be solved* using an algorithm with worst-case polynomial time complexity, such problems are called **intractable**.
- If it can be shown that no algorithm exists for solving them, such problems are called **unsolvable**.

Class NP and NP-Complete

- Problems for which a solution can be ***checked*** in **polynomial time** are said to belong to the **class NP**.
  - Tractable problems belong to class **P**.

- **NP-complete** problems are an important class of problems
  - If any of these problems can be solved by a polynomial worst-case time algorithm then …
    - All problems in the class NP can be solved by polynomial worst-case time algorithms.

NP-Hard

- **NP-Hard** problems are *at least as hard* as the hardest problems in NP