# Part 5. Computational Complexity (3)

CS 200 Algorithms and Data Structures



## Outline

- **Recurrence Relations**
- Divide and Conquer
- Understanding the Complexity of Algorithms



## Recurrence Relations: An Overview

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



## Recurrence Relations: 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: $a_n = 2a_{n-1} + 1,\ a_1 = 1$



## 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)

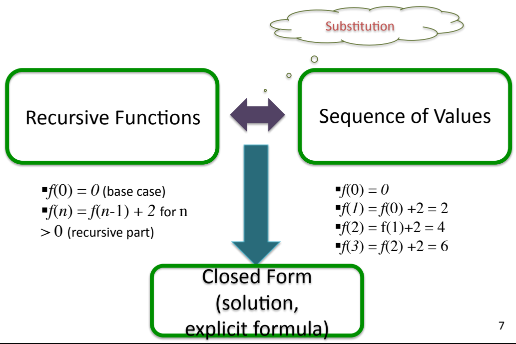
$$b_n = b_{n-1} + rb_{n-1} = (1+r)^n b_0$$



## Recursively defined functions and recurrence relations

- **A recursive function**
  - $f(0) = a$ (base case)
  - $f(n) = f(n-1) + d$ for $n > 0$ (recursive part)
- 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




7

## Solving recurrence relations (1)

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

(1) What is the recursive function?

(2) What is the sequence of values?

**Hint 1**: Solve by substitution

8

## Solving recurrence relations (2)

**Solve** $a_0 = 2; \quad a_n = 3a_{n\text{-}1} + 2$ **for** $n > 0$

(1) What is the recursive function?

(2) What is the sequence of values?

**Hint 1**: Solve by substitution

**Hint 2:** Use formula for summing geometric series

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

9

## Solving recurrence relations (3)

**Solve** $a_0 = 1; \quad a_n = a_{n\text{-}1} + \mathrm{n}$ **for** $n > 0$

**Hint 1**: Solve by substitution

10

## Modeling with Recurrence

- 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.

11

## Linear Recurrence Relations

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

$a_0 = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k}$

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

12

## Is this linear homogeneous recurrence relation?

- $f_n = f_{n-1} + f_{n-2}$
- $a_n = a_{n-1} + a^2_{n-2}$

- Modeling of problems
- They can be systematically solved.

13

## Theorem 1

Let $c_1$ and $c_2$ a. 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 $a_n = c_1 a_{n-1} + c_2 a_{n-2}$ if and only if an $= \alpha_1 r_1^n + \alpha_2 r_2^n$ for $n = 0,1,2,\ldots$ where $\alpha_1$ and $\alpha_2$ are constants. (Rosen 6th edition, section7.2 theorem 1)

14

## What is the solution of the recurrence relation

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

15

## Outline

- Recurrence Relations
- **Divide and Conquer**
- Understanding the Complexity of Algorithms

16

## 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

17

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

18

## 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 (for 6th Edition 7.3) in Rosen*

19

## Example: Binary Search using the Master Theorem

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

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

for the binary search algorithm

$f(n) = a\, f(n / b) + n^d$
$\quad = 1\, f(n / 2) + 3$

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, $f(n) = O(n^d \log_2 n)$
$\quad = O(n^0 \log_2 n)$
$\quad = \mathbf{O(\log_2 n)}$

20

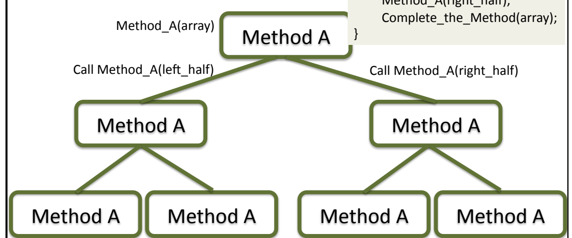
## Step-by-step Master Theorem

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

21

## Step-by-step Maste

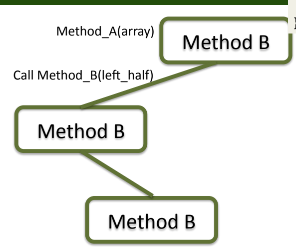
```
Method_A(array){
    Method_A(left_half);
    Method_A(right_half);
    Complete_the_Method(array);
}
```




22

## Step-by-step Maste

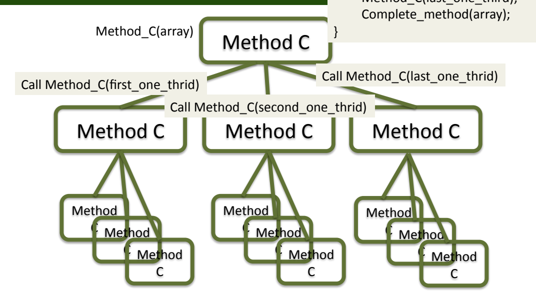
```
Method_B(array){
    Method_A(possible_half);
    Do_nothing_Method(array);
}
```



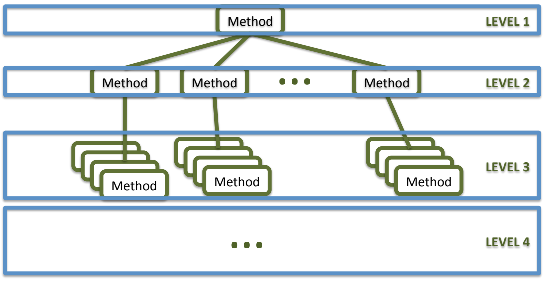

23

## Step-by-step Maste

```
Method_C(array){
    Method_C(first_one_third);
    Method_C(second_one_third);
    Method_C(last_one_third);
    Complete_method(array);
}
```

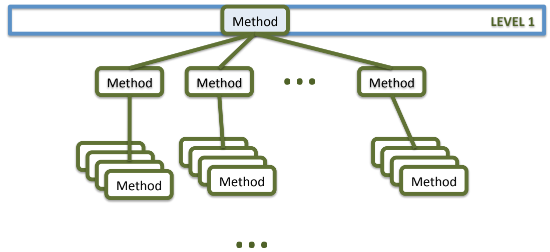



24



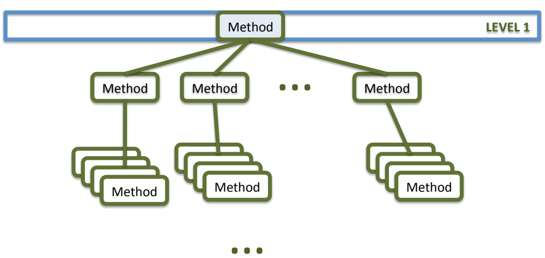

Goal 1: What is the complexity per level?
Goal 2: What is the total complexity?

25



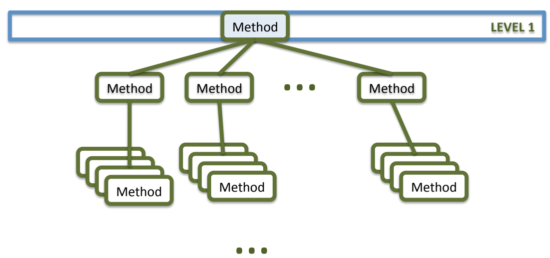

"Method" is called with input size $n$.
"Method" calls "Method" recursively $a$ times
"Method" calls "Method" recursively with input size $n/b$

26




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

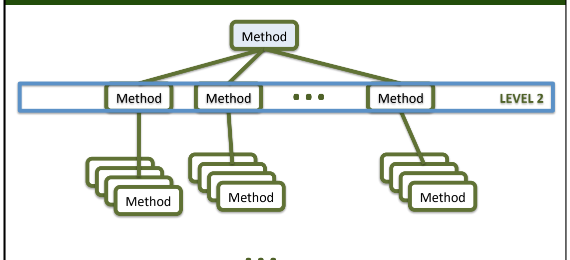
27




Therefore, in LEVEL 1, "Method" requires,

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

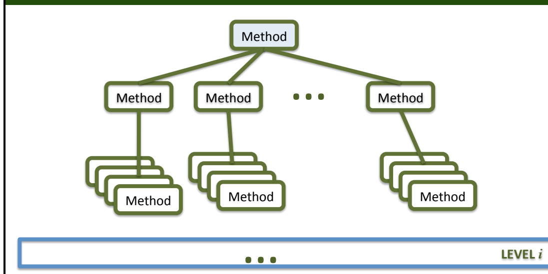
28




This is also true for LEVEL 2

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

29




This is also true for LEVEL $i$

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

30

## More generally,

Let $f$ be an increasing function that satisfies

$f(n) = af(n/b) + g(n) = a \cdot f(n/b) + n^c$ ,if n > 1

$f(n) = d$ ,if n = 1

whenever $n = a^b$, where a 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.

31

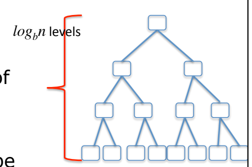
## Base case

- 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 $d \neq 1$.

32

## 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 $(n/b^i)^c$ additional work.

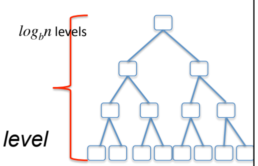



33

## The total number of units of work on level $i$

- $a^i(n/b^i)^c \;=\; n^c\,(a^i/b^{ci})$
  $\quad\quad\quad\;\; = n^c\,(a/b^c)^i$

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




34

## Observation

- Now, observe that,

  $(a/b^c) = 1$

  $a = b^c$

- Therefore, the relations,

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

are the conditions deciding the types of the growth function

35

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

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

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

$(1)\ a < b^c$

$(2)\ a = b^c$

$(3)\ a > b^c$

36

## Case 1. $a < b^c$

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

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

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

37

## Case 2. $a = b^c$

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

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

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

38

## Case 3. $a > b^c$

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

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

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

39

## Complexity of MergeSort with Master Theorem (1/2)

- 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 $M(n)$ satisfying the divide-and-conquer recurrence relation
- $M(n) = 2M(n/2) + n$

40

## Complexity of MergeSort with Master Theorem (2/2)

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

for the mergesort algorithm

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

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

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

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

41

## Quicksort?

42

## The Closest-Pair Problem

- Consider the problem of determining the **closest pair of points** in a set of $n$ points $(x_1, y_1)\ldots(x_n, y_n)$ in the plain, where the distance between two points $(x_i, y_i)$ and $(x_j, y_j)$ is the usual Euclidean distance,

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

43

## Solution (1/5)

- Calculate the distance between every pair of points and then find the smallest of these distances
  - It will take $O(n^2)$

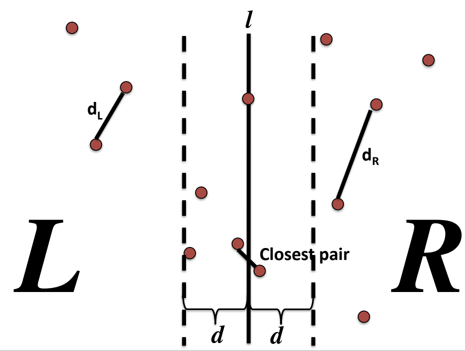
44

## Solution (2/5)

- Assume that we have $n$ points such that $n = 2^k$.
- When $n = 2$, the distance between two points is the minimum distance.
- We sort the points in order of increasing $x$ coordinates, and also sort the points in order of increasing $y$ coordinates using mergesort.
  - It will take $O(n \log n)$
- Using sorted list by $x$ coordinate, divide the group of points into two (***R***ight and ***L***eft)

45

## Solution (3/5)




46

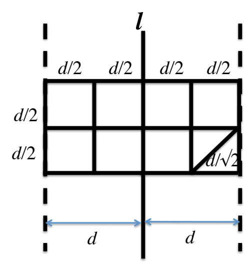
## Solution (4/5)

- Find the closest pair within ***R*** and ***L***.
- Merge the region ***R*** and ***L*** by examining the points on the boarder area.
  - Find closest pair between pairs from R and L. ($d = min(d_r, d_l)$)
  - Check the points within the strip of the maximum distance of $d$ from the border.
  - Sort the points within the strip with associated $y$ coordinate.
  - Beginning with a point in the strip with the smallest $y$ coordinate.

47

## Solution (5/5)

- We need at most 7 comparisons for each of the points in the strip.




48

## Analysis

- We find the function $f(n)$ satisfying the recurrence relations

$$f(n) = 2f(n/2) + 7n$$

where $f(2) = 1$

By the master theorem, $f(n) = O(n \log n)$



## Outline

- Recurrence Relations
- Divide and Conquer
- **Understanding the Complexity of Algorithms**



## Understanding the Complexity of Algorithm

- Commonly used terminology for the complexity of algorithms
  - ✓ $O(1)$ : Constant complexity
  - ✓ $O(\log n)$ : Logarithmic complexity
  - ✓ $O(n)$ : Linear complexity
  - ✓ $O(n \log n)$ : Linearithmic complexity
  - ✓ $O(n^b)$ : Polynomial complexity
  - ✓ $O(b^n)$, where $b > 1$ : Exponential complexity
  - ✓ $O(n!)$ : Factorial complexity



## 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

55

## Next Reading

- Section 11.1, 11.2 in Prichard

56