Recursion and Recurrence

Prepared by Dr. Lee

Repetitive Algorithms

- Two approaches to writing repetitive algorithms
  - Iteration
  - Recursion
- Recursion is a repetitive process in which an algorithm calls itself
  - Usually recursion is utilized in such a way that a subroutine calls itself or a function calls itself

Iteration vs Recursion

- Iteration vs Recursion
  - Iterative algorithms may be reduced to the recursive algorithms
  - This means that often the analysis of repetitive algorithms can be reduced to the analysis of recursive algorithms

Factorial – a case study

- The factorial of a positive number is the product of the integral values from 1 to the number:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{i=1}^{n} i$$

Iterative Factorial Algorithm

- Iterative Factorial Algorithm Definition

$$Factorial(n) = \begin{bmatrix} 1 & if \quad n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & if \quad n > 0 \end{bmatrix}$$

  - A repetitive algorithm is defined **iteratively** whenever the definition involves only the algorithm parameter (parameters) and not the algorithm itself.

Recursive Factorial Algorithm

- Recursive Factorial Algorithm Definition

$$Factorial(n) = \begin{bmatrix} 1 & if \quad n = 0 \\ n \times (Factorial(n-1)) & if \quad n > 0 \end{bmatrix}$$

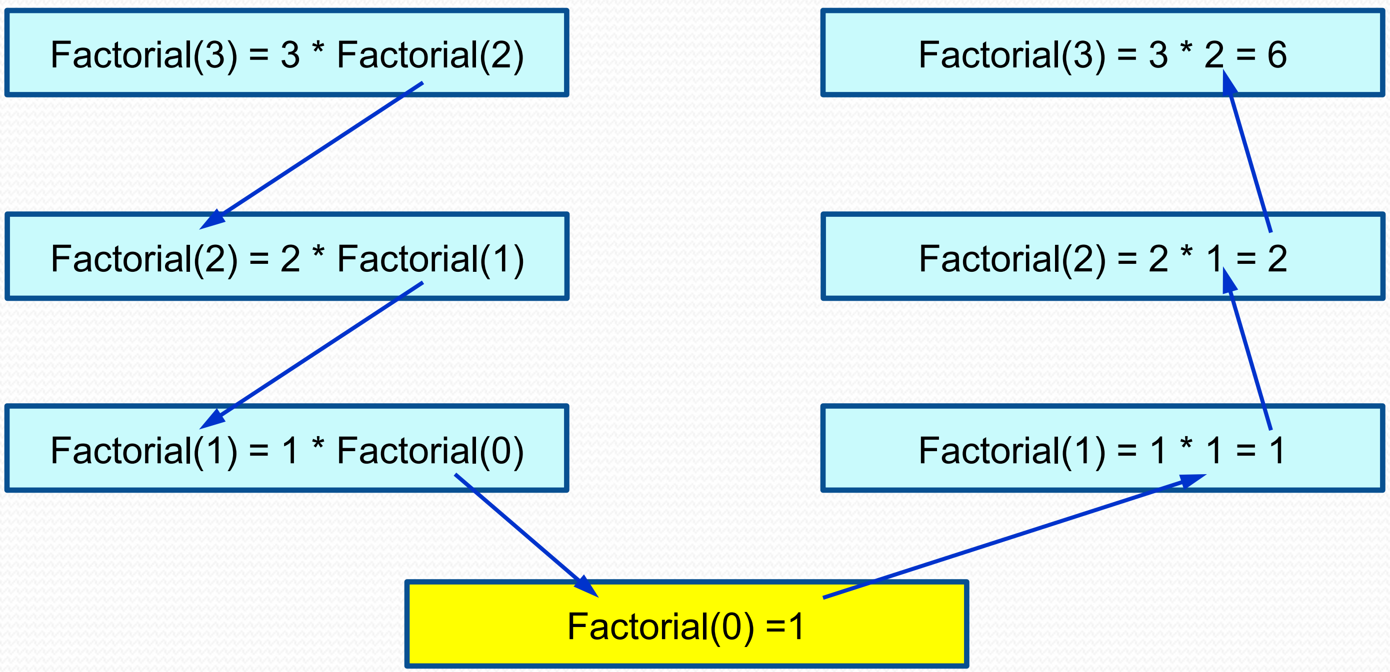
  - A repetitive algorithm uses **recursion** whenever the algorithm appears within the definition itself.

Recursion: basic point

- The recursive solution for a problem involves a two-way journey:
  - First we **decompose** a problem from the top to the bottom
  - Then we **solve** the problem from the bottom to the top.

Factorial (3): Decomposition and solution

- Factorial(3) Recursively



Iterative Factorial Algorithm

ITERFACTORIAL(N)

1. Nfact ← 1
2. for i ← 1 to N do
3. Nfact ← Nfact × i
4. Return (Nfact)

Computational complexity?

Recursive Factorial Algorithm

```
RECURSIVEFACTORIAL(N)
If (N=0)
then
        Nfact ← 1
else
        Nfact ← N × RECURSIVEFACTORIAL (N–1)
Return (Nfact)
```

Computational complexity?

Designing recursive algorithms

- Each step (or each call) of a recursive algorithm solves one **part** of the problem and reduces the **size** of the problem.
- The general part of the solution is the recursive call. At each recursive call, the size of the problem is reduced.
- Every recursive algorithm must have a ***base case*** that "**solves**" the problem.
- The rest of the algorithm is known as the ***general case***. The general case contains the logic needed to reduce the size of the problem.

Designing recursive algorithms

- Once the ***base case*** has been reached, the decomposition is complete and the solution begins.
- We now know one part of the answer and can return that part to the next, more general statement.
- This allows us to solve the next ***general case***.
- As we solve each ***general case*** in turn, we are able to solve the next-higher ***general case*** until we finally solve the ***most general case***, which solves the original problem.

Designing recursive algorithms

- The rules for designing a recursive algorithm:
  1. First, determine the **base case**.
  2. Then determine the **general case**.
  3. **Combine** the base case and the general cases into an algorithm

Designing recursive algorithms

- Each recursive call must reduce the size of the problem and move it toward the **base case**.
- The **base case**, when reached, must terminate without a call to the recursive algorithm; that is, it must execute a return.

Divide-and-Conquer Approach

Prepared by Dr. Lee

Divide-and-Conquer Approach

- Many useful algorithms are **recursive in structure**.
- To solve a given problem, they call themselves recursively to deal with closely related subproblems.
- **Divide-and-Conquer** (**DaC**) approach means that an algorithm breaks the problem into several subproblems that are similar to the original problem but smaller in size.
- Then these smaller subproblems should be solved.
- Then their solutions are combined into the original problem solution.

Divide-and-Conquer Approach

- DaC paradigm involves three steps at each level of the recursion:
  - Divide the problem into a number of subproblems.
  - Conquer the subproblems by solving them.
  - Combine the solutions to the subproblems into the solution for the original problem.

Divide-and-Conquer Approach

- Efficiency
  - The efficiency function of an algorithm designed using the DaC approach is a sum of the subproblems efficiency functions and the “combine-merger” efficiency function.

Example – Problem 1

- Assume that we are reading data from the keyboard until the “end of input” sign is entered and need to print the data in reverse.
- The easiest formal way to print the list in reverse is to write a recursive algorithm using the **divide-and-conquer** approach.

Solution

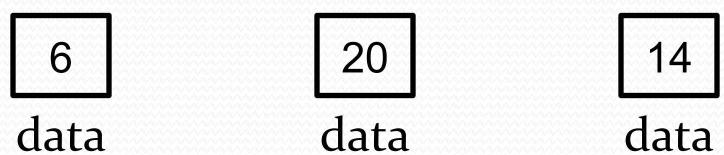
- It should be obvious that to print the list in reverse, we must first read all of the data. If we print before we read all of the data, we print the list in sequence. If we print after we read the last piece of data – that is, if we print it as we back out of the recursion – we print it in reverse sequence.
- The **base case**, therefore, is that we have read the last piece of data.
- Similarly, the **general case** is to read the next piece of data.

Implementation

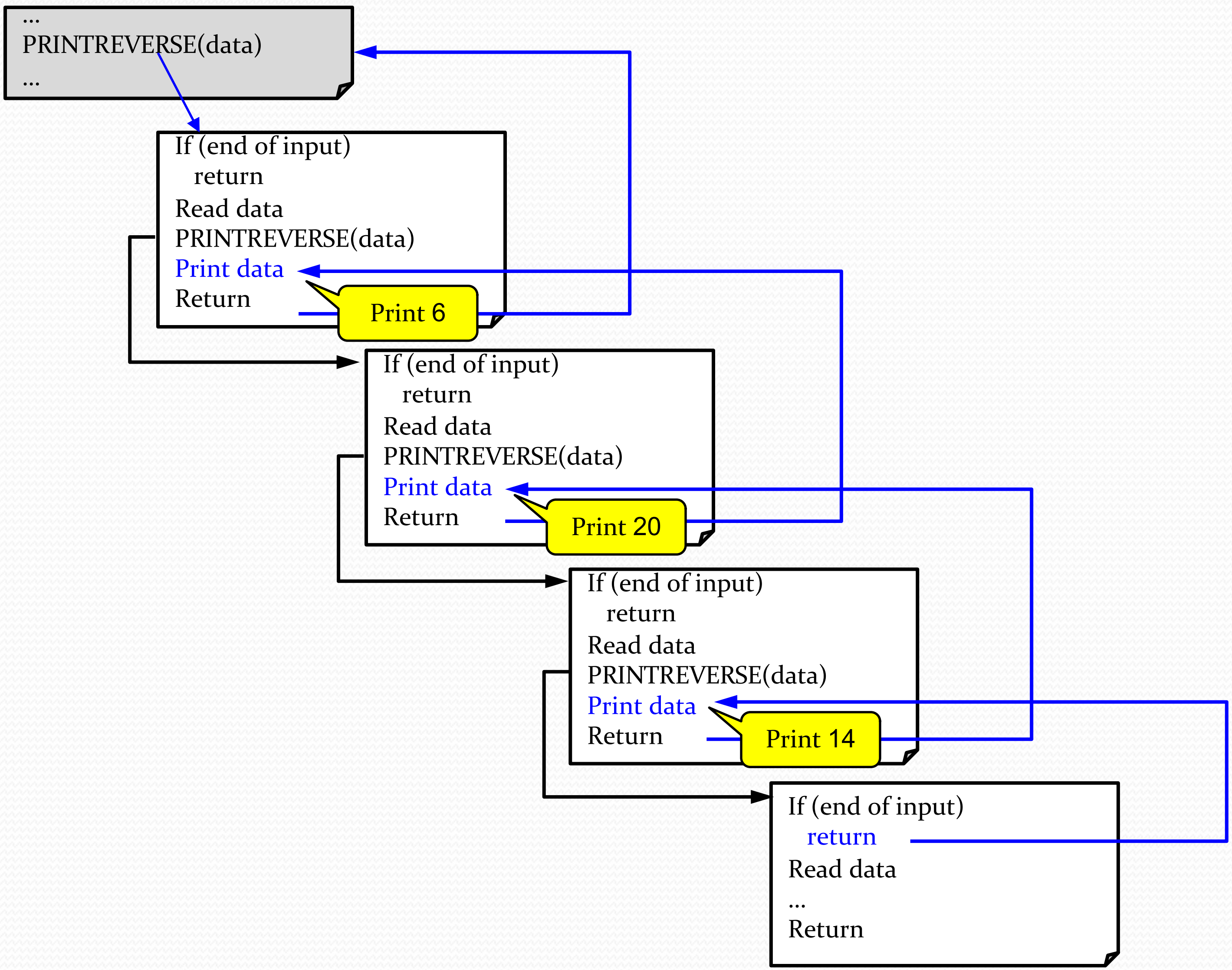
```
PRINTREVERSE(data)
If (end of input)
then
        Return    // This is the base case
Read(data)
PRINTREVERSE(data)
// This statement will be executed only after the last symbol will be read
PRINT(data)
Return
```

Print Keyboard Input in Reverse

- Recursive calls (reads)



- Recursive returns (prints)


...
PRINTREVERSE(data)
...
If (end of input)
return
Read data
PRINTREVERSE(data)
Print data
Return
Print 6
If (end of input)
return
Read data
PRINTREVERSE(data)
Print data
Return
Print 20
If (end of input)
return
Read data
PRINTREVERSE(data)
Print data
Return
Print 14
If (end of input)
return
Read data
...
Return

Analysis

- The first subproblem is reading data from the keyboard
- The second subproblem is printing the list
- The running time for both subproblems is n.
- Hence the running time for the entire problem is $n + n = 2n$, its efficiency is $\Theta(n)$

Example – Problem 2

- Determine the greatest common divisor (*GCD*) for two numbers.
- Euclidean algorithm: $GCD(a, b)$ can be recursively found from the formula

$$GCD(a,b)=\begin{cases} a & \text{if} \quad b=0 \\ b & \text{if} \quad a=0 \\ GCD(b, a \bmod b) & \text{otherwise} \end{cases}$$

  - $a \bmod b$ is determined as follows:

$$a \bmod b = \text{remainder of } a \,/\, b$$

Implementation

```
GCD(a, b)
if (b = 0)
then
        result ← a                    // This is the base case 1
else
        if (a = 0)
        then
                result ← b            // This is the base case 2
        else
         GCD(b, a mod b)              // This is the general case
return
```

Implementation – Example

Find *GCD*(60, 36) **12**

$a = 60$; $b = 36$

GCD(36, 60 mod 36) = GCD(36, 24) → general case

GCD(24, 36 mod 24) = GCD(24, 12) → general case

GCD(12, 24 mod 12) = **GCD(12, 0)** → **base case**

GCD(12, 0) = **12**

Analysis

- The efficiency of the algorithm is logarithmic: the number of steps is about ~lg b for $b<a$
- Once the base case is reached (either a or b is 0), the problem is solved
- The general case is determined by $GCD(b, a \bmod b)$

Example – Problem 3

- Generation of the Fibonacci numbers series.
- Each next number is equal to the sum of the previous two numbers.
- A classical Fibonacci series is 0, 1, 1, 2, 3, 5, 8, 13, …
- The series of *n* numbers can be generated using a recursive formula

$$Fibonacci(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ Fibonacci(n-1) + Fibonacci(n-2) & otherwise \end{cases}$$

Implementation

```
FIBONACCI(n)
if (n = 0)
then
            result ← 0          // This is the base case 1
else
            if (n = 1)
            then
              result ← 1        // This is the base case 2
            else
              result = FIBONACCI(n – 1) + FIBONACCI(n – 2)
Return
```

Implementation – Example

Find Fibonaci(5) **5**

Fibonacci(5)=Fibonacci(4)+Fibonacci(3) 5

Fibonacci(4)=Fibonacci(3)+Fibonacci(2) → general case 3

Fibonacci(3)=Fibonacci(2)+Fibonacci(1) → general case 2

Fibonacci(2)=Fibonacci(1)+Fibonacci(0) → general case 1

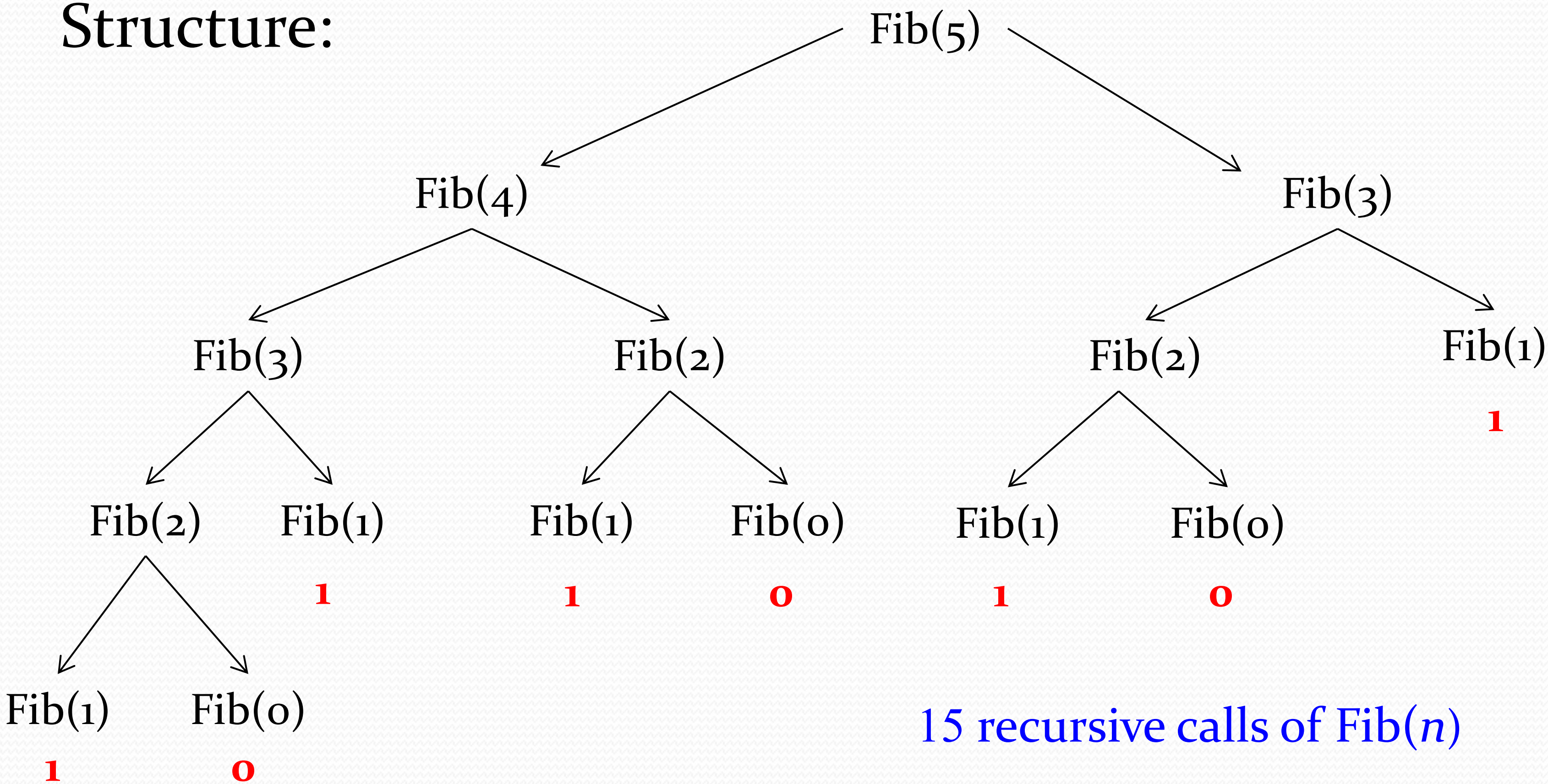
Fibonacci(1)=1; Fibonacci(0)=0 → **base case**

Recursion Tree

- To represent a Divide-and-Conquer approach, a recursion tree should be used.
- A root of the recursion tree represents the most general case (solution).
- The lowest level leaves of the recursion tree represent the base case.
- Other nodes represent general cases.

Analysis

- Recursion Tree Shows the Divide-and-Conquer Structure:



15 recursive calls of Fib(n)

Analysis

- The efficiency of the Fibonacci recursive algorithm is **exponential !!!**

Fib(n)	Calls	Fib(n)	Calls
1	1	11	287
2	3	12	465
3	5	13	753
4	9	14	1219
5	15	15	1973
6	25	20	21,891
7	41	25	242,785
8	67	30	2,692,573
9	109	35	29,860,703
10	177	40	331,160,281

Solving Recurrences

Prepared by Dr. Lee

Analysis of Recurrences

- How we can evaluate the running time/efficiency of the recursive algorithms?

Recurrence

- When an algorithm contains a recursive call to itself or if it is represented using a Divide-and-Conquer approach, its running time can often be described by a **recurrence equation** or **recurrence**
- It describes the overall running time on a problem of size *n* in terms of running time on smaller inputs

Solving Recurrences

- Solving recurrences means the asymptotic evaluation of their efficiency
- The recurrence can be solved using some mathematical tools and then bounds (big-O, big-Ω, and big-Θ) on the performance of the algorithm should be found according to the corresponding criteria

Composing Recurrences

- A recurrence for the running time of a divide-and-conquer algorithm is based on the three steps:
  1) Let $T(n)$ be the running time of a problem of size n. If the problem size is small enough ($n \le c$) for some constant c, the straightforward solution takes constant time, i.e. $\Theta(1)$
  2) Suppose that our division of the problem yields k subproblems, each of which is $1/m$ size of the original.
  3) If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems to the original problem, we got the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ kT(n/m) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Solving Recurrences

- Hence, solving recurrences means finding the asymtotic bounds (big-O, big-Ω, and big-Θ) for the function $T(n)$

Solving Recurrences

- **Substitution method** – we guess a bound and then use mathematical induction to prove our guess
- **Recursion-tree method** converts recursion into a tree whose nodes represent the "subproblems" and their costs. It is used to estimate a good guess
- **Master Theorem method** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n); \quad a \geq 1, \quad b > 1$$

$f(n)$ is a given function

Solving Recurrences

- **Master Theorem method**
  - Provides the immediate solution for recurrences of the form

$$T(n) = aT(n/b) + f(n); \quad a \geq 1, \quad b > 1$$

  - $f(n)$ is a given function, which satisfies some pre-determined conditions

Solving Recurrences

- **Recursion-tree method**
  - Converts recursion into a tree whose nodes represent the "subproblems" and their costs
  - Then the sum of these costs can be used as a "good guess" for the substitution method or the master theorem method

Solving Recurrences

- **Substitution method**
  - Known as a “good guess method”
  - The first step is: to guess a solution (a bound)
  - The second step is: to prove the correctness of the guess substituting the guess into the recurrence and using induction.

Substitution Method: Example

$$T(n) = \begin{cases} 1 & if \quad n = 1 \\ 2T\left(\dfrac{n}{2}\right) + n & if \quad n > 1 \end{cases}$$

- Guess for the exact solution: $g(n) = n\lg n + n$

Substitution Method (the exact solution)

- Induction: Guess: $T(n) = n \lg n + n$
- **Basis**: $n = 1 \Rightarrow T(n) = 1$; $T(n) = n\lg n + n = 1\cdot\lg 1 + 1 = 1$
  $\rightarrow n_0 = 1$
- **Inductive step**: Inductive Hypothesis is

$$T(k) = k \lg k + k, \qquad \forall k \geq n_0$$

- Let us use this hypothesis:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(\underbrace{\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}}_{substitution\ \ T(n/2)}\right) + n = n\lg\frac{n}{2} + n + n =$$

$$= n(\lg n - \lg 2) + n + n = n\lg n - n + n + n = n\lg n + n \qquad \square$$

Substitution Method

- Generally, we use asymptotic notation
  - We would write $T(n) = 2T(n/2) + \Theta(n)$
  - We assume $T(n) = O(1)$ for sufficiently small n
  - We express the solution by asymptotic notation:

    $$T(n) = \Theta(n \lg n)$$
- For the substitution method
  - Name the constant in the additive term
  - Show the upper(O) and lower (Ω) bounds separately. Might need to use different constants for each.

Substitution Method (with asymptotic notation)

- $T(n) = 2T(n/2) + \Theta(n)$
- If we want to show an upper bound of $T(n) = 2T(n/2) + O(n)$, we write $T(n) \le 2T(n/2) + cn$ for some positive constant c

Substitution Method (with asymptotic notation)

$$T(n)=\begin{cases}1 & \text{if} \quad n=1\\ 2T\left(\frac{n}{2}\right)+n & \text{if} \quad n>1\end{cases}$$

- Upper bound:
  - Guess: $T(n) \le dn \lg n$ for some positive constant d.
  - Substitution:

$$T(n) \le 2T(n/2) + cn = 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn = dn\lg\frac{n}{2} + cn =$$

$$= dn\lg n - dn + cn \le dn\lg n$$

if $-dn + cn \le 0$, $d \ge c$

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

What about n_0?

$T(1) = 1 \le d1 \lg 1 = 0$ (no)

$T(2) = 4 \le d2 \lg 2 = 2d$ (yes)

$\Rightarrow d \ge 2,\ n_0 = 2$

Substitution Method (with asymptotic notation)

$$T(n) = \begin{cases} 1 & \text{if} \quad n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{if} \quad n > 1 \end{cases}$$

- 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:

$$T(n) \geq 2T(n/2) + cn = 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn = dn\lg\frac{n}{2} + cn =$$

$$= dn\lg n - dn + cn \geq dn\lg n$$

if $-dn + cn \geq 0$, $d \leq c$

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

What about n_0?

$T(1) = 1 \geq d1 \lg 1 = 0$ (yes)

$T(2) = 4 \geq d2 \lg 2 = 2d$ (yes)

$\Rightarrow d \leq 2,\ n_0 = 2$

  - Therefore, $T(n) = \Theta(n \lg n)$ □

Solving Recurrences

- The substitution method
  - Examples:
    - $T(n) = 2T(n/2) + O(n)$ $\rightarrow$ $T(n) = O(n\lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n$ $\rightarrow$???

Solving Recurrences

- The substitution method
  - Examples:
    - $T(n) = 2T(n/2) + \mathrm{O}(n) \quad \rightarrow \quad T(n) = \mathrm{O}(n\lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \quad \rightarrow \quad T(n) = \mathrm{O}(n\lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \rightarrow$???

Solving Recurrences

- The substitution method
  - Examples:
    - $T(n) = 2T(n/2) + O(n) \quad \rightarrow \quad T(n) = O(n\lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \quad \rightarrow \quad T(n) = O(n\lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \rightarrow \quad T(n) = O(n\lg n)$

Recursion Tree

- A recursion tree is used to present a problem as a composition of subproblems. It is very suitable to present any divide-and-conquer algorithm
- Each node represents the cost of a single subproblem
- Usually each level of the tree corresponds to one step of the recursion

Recursion Tree

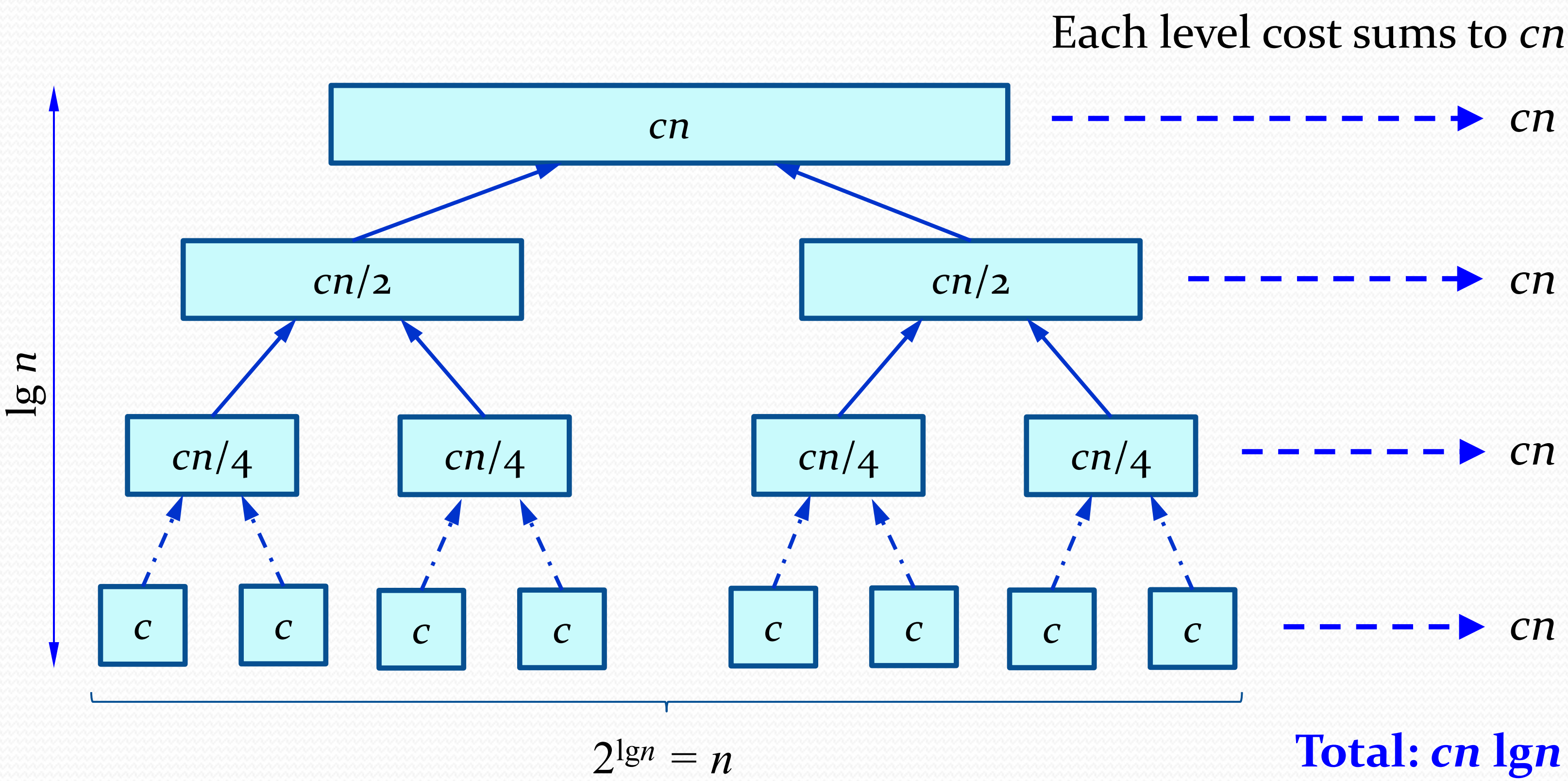
- We sum the costs within each level of the tree to obtain a set of per-level costs
- Then we sum all the per-level costs to determine the total cost of all levels of the recursion
- As a result, we generate a guess that can be then proven by the substitution method

Recursion Tree: Determination of a "Good" Asymptotic Bound

- Draw the tree based on the recurrence
- From the tree determine:
  - # of levels in the tree
  - cost per level
  - # of nodes in the last level
  - cost of the last level (which is based on the number of nodes in the last level)
- Write down the summation using $\sum$ notation – this summation sums up the cost of all the levels in the recursion tree
- Simplify the summation expression coming up with your "guess" in terms of Big-O, or Big-Ω depending on which type of asymptotic bound is being sought).
- Then use Substitution Method to prove that the "guess" is correct.

Recursion Tree: Example – Merge Sort

- Total number of elements per level is always n



Total: $cn \lg n + cn$

Recursion Tree: Example – Merge Sort

- Close form solution as "guess"

$$T(n) = cn \lg n + cn = cn \lg n + O(n) = O(cn \lg n) + O(n) = O(n \lg n)$$

- Substitution method
  - Assume n is a power of 2 to avoid floor and cell complica.

$$T(n) = \begin{cases} c & \text{if} \quad n = 1 \\ 2T(n/2) + cn & \text{if} \quad n > 1 \end{cases}$$

  - Inductive Hypothesis (IH):
    - Assume: $T(k/2) \leq d\, k/2 \lg k/2$
    - Show: $T(k) = 2\, T(k/2) + ck \leq d\, k \lg k$

Recursion Tree: Example – Merge Sort

- $T(k) = 2T(k/2) + ck$ Recurrence
  $\le 2(dk/2 \lg k/2) + ck$ Substitute IH
  $= dk \lg k/2 + ck$
  $= dk \lg k - dk + ck \le dk \lg k$

- Find d that satisfies the last line

$$
\begin{aligned}
dk \lg k - dk + ck &\le dk \lg k \\
-dk + ck &\le 0 \\
ck &\le dk \\
c &\le d
\end{aligned}
$$

Satisfied by $d \ge c$

Recursion Tree: Example – Merge Sort

- Basis:
  $T(1) = 2T(1/2) + c{\cdot}1 = c \le d \cdot 1 \lg 1 = 0$
  since need $n \ge n_0$ for n a power of 2, choose $n_0 = 2$

- Use as basis:
  $T(2) = d2 \lg 2 = 2d$

- By the recurrence, where c is the constant divide and combine time:
  $$\begin{aligned} T(2) &= 2T(2/2) + 2c \\ &= T(1) + T(1) + 2c \\ &= c + c + 2c = 4c \end{aligned}$$

Recursion Tree: Example – Merge Sort

Need $T(2) = 4c \le d2 \lg 2 = 2d$

$4c \le 2d$

so let $d = 2c$

Satisfied $d = 2c \ge c$

- $O(n \lg n)$: $0 \le T(n) \le dn \lg n$ for $d > 0$, for $\forall n \ge n_0$ satisfied by $d \ge 2c > 0$, for $\forall n \ge n_0 = 2$

APPENDIX

Substitution Method (with asymptotic notation)

- Induction: Guess: $T(n) = O(n \lg n)$
- Basis: $n = 1 \rightarrow T(1) = 1 > c \cdot g(1) = c \cdot 1 \cdot \lg 1 = 0$
  $n = 2 \rightarrow T(2) = 2 \cdot T(1) + 2 = 4 \leq c \cdot g(2) = c(2 \cdot \lg 2) = 2c \rightarrow 2 \leq c$
- Inductive Hypothesis:
  $T(n) = O(n \lg n), \quad \forall n \geq n_0 \qquad \exists c > 0,\ n_0 = 2:\ T(n) \leq cn \lg n$
- Inductive step

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq 2\left(c\frac{n}{2}\lg\frac{n}{2}\right) + n = cn\lg\frac{n}{2} + n = cn\left(\lg n - \lg 2\right) + n =$$

$$= cn\lg n - cn\lg 2 + n = cn\lg n - cn + n = cn\lg n - n(c-1) \leq cn\lg n$$

$$n(c-1) \geq 0; n > 0, c > 0 \Rightarrow c - 1 \geq 0 \Rightarrow c \geq 1$$

Substitution Method (with asymptotic notation)

- Analysis: Guess: $T(n) = O(n \lg n)$
- We have to find such $c \geq 1$ and n_0 that

$$\forall n \geq n_0 : T(n) \leq cn\lg n$$

$$n_0 = 1;\ T(1) = 1;\ g(n) = 1 \cdot \lg 1 = 0;$$

$$cg(n) = c \cdot 1 \cdot \lg 1 = c \cdot 0 = 0;\ T(1) = 1 > 0 \quad \rightarrow \quad n_0 > 1$$

$$n_0 = 2;\ T(2) = 2 \cdot T(1) + 2 = 2 \cdot 1 + 2 = 4;\ g(2) = 2 \cdot \lg 2;$$

$$c \cdot 2 \cdot \lg 2 = 2c; \qquad 4 \leq 2c \quad \forall c \geq 2 \qquad \rightarrow \quad n_0 = 2;\ c \geq 2$$