算法设计与分析

Lecture 4: Recursion

卢杨
厦门大学信息学院计算机科学系
luyang@xmu.edu.cn

Recursion

- Recursion (递归) is one of most powerful methods of solution available to computer scientists.
- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that would be difficult to solve in other ways.
- Recursion splits an problem instance into one or more simpler instances of the same problem.

Homer and Bart

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Image source: https://giphy.com/explore/infinite-recursion

Design a Recursive Algorithm

- Base case: There must be at least one case, for a small value of n, that can be solved directly.
- Recursive case: A problem instance of a given size n can be split into one or more smaller instances of the same problem.
- Steps:
  - Recognize the base case and provide a quick solution to it.
  - Devise a recursion to split the instance into smaller instances of itself, while making progress toward the base case.
  - Combine the solutions of the smaller problems in such a way as to solve the larger problem.

Design a Recursive Algorithm

Questions when using recursive solution:

- How to define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

Why Use Recursion?

- Advantages
  - Interesting conceptual framework (good recursion algorithm is art).
  - Intuitive solutions to difficult problems.
- But, disadvantages...
  - More memory & time.
  - Different way of thinking!

Correctness of Recursive Algorithm

Correctness proof of recursion is similar to induction.

- Base case: Verify that the base case is recognized and solved correctly.
- Induction step: Verify that if all smaller problems are solved correctly, then the original problem is also solved correctly.

Recursion

Example 1

Consider the function $f(n)$ which calculates 2 to the power of n, namely $f(n) = 2^n$.

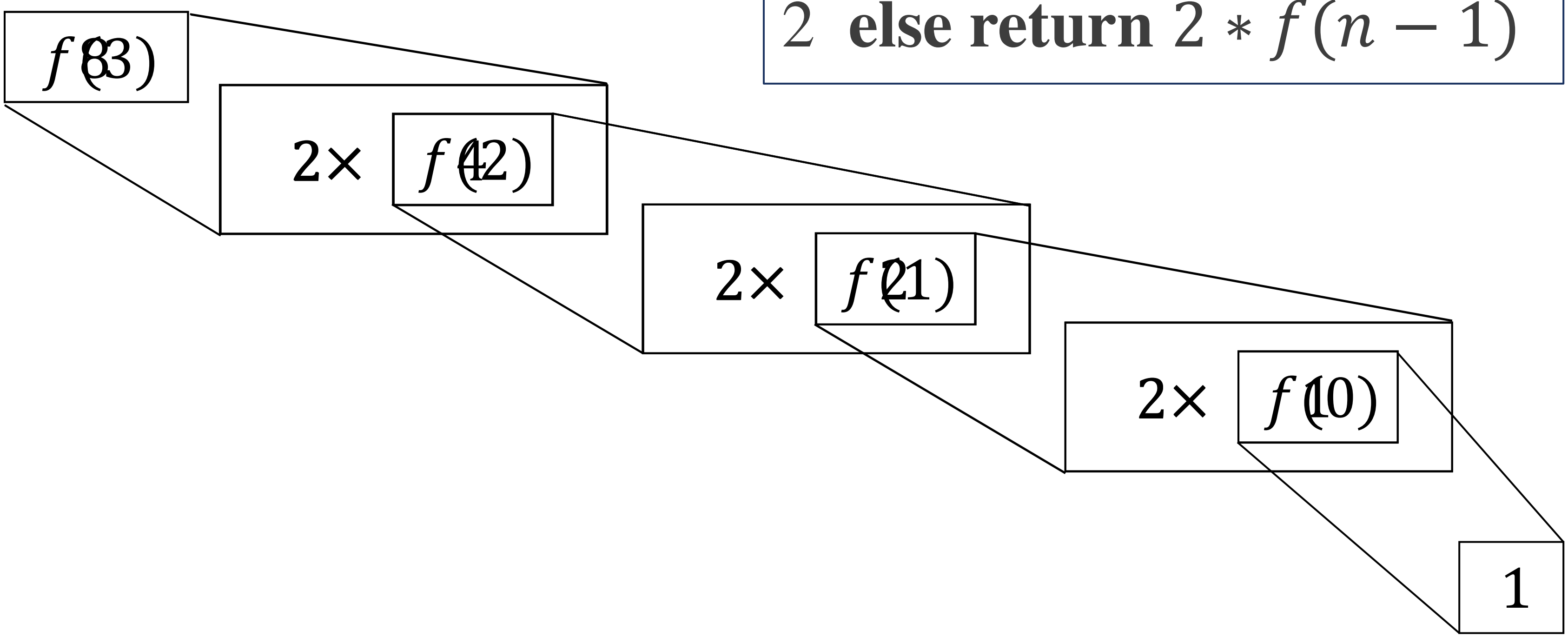
This can be expressed as:

$$f(n) = \begin{cases} 1 & \text{if } n = 0, \\ 2 \times f(n-1) & \text{otherwise.} \end{cases}$$

Recursion

Example 1 (cont'd)

$f(n)$
if $n = 0$ **then return** 1
else return $2 * f(n-1)$



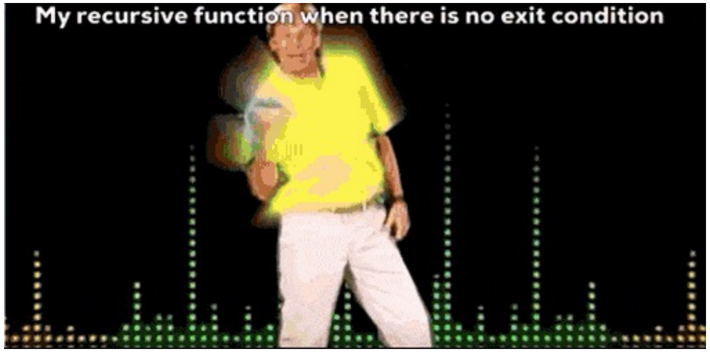
Example 1 (cont'd)

Correctness proof:

- Base case:
  - By definition, $f(0) = 2^0 = 1$, and the recursive algorithm returns 1 when $n = 0$. Therefore, the base case holds.
- Inductive step:
  - Assume that the property is true for $n = k$, i.e. $f(k) = 2^k$. We have to show that the property is true for $n = k + 1$.
  - By recursive algorithm, $f(k+1)$ returns $2 \times f(k) = 2 \times 2^k = 2^{k+1}$. So, inductive proof is complete.

Recursion

- $f(n) = 2 * f(n-1)$ is recursive definition of a function, which is defined in terms of itself.
- Therefore, to stop, there must be a case when it does not call itself (called base case, stopping condition or exit condition (递归出口)).
- Recursion is an alternative to looping. As with looping, recursion can cause your program to loop forever.



Exit condition is very important for recursion...

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY
厦门大学计算机科学系
Computer Science Department of Xiamen University

Image source: https://gfycat.com/soulfulfairarthropods

Rules of Recursion

- Base cases: Always have the base case (stopping condition), which is solved without recursion.
  - Base case is usually the simplest case to solve.
- Making progress: for recursive cases, each new call must always make progress towards base case.
  - Sometimes you have the base case but it can never be reached.
- Design Rule: assume all recursive calls work.

Efficiency of Recursion

- The nature of recursion is iteration. Therefore, any recursive function can be converted to an equivalent iterative (looping) method.
- Although recursion is elegant, it can be inefficient, because there are more calls to methods.
  - Sometimes, there are many recursive calls to the same instance.
- Iterative methods are more efficient and faster.

$f(n)$

$total \leftarrow 1$

for $i \leftarrow 0$ **to** n **do**

  $total \leftarrow total * 2$

return $total$

Iterative way to write $f(n)$

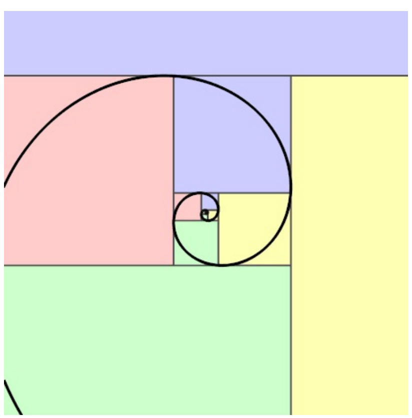
Recursion

Example 2: Fibonacci sequence (斐波那契数列)

- Fibonacci sequence is defined by

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} - f_{n-2}, \qquad \text{for } n \geq 2$$

- 0, 1, 1, 2, 3, 5, 8, 13, 21….



```
Fib(n)
if n ≤ 1 then
    return n
else
    return Fib(n − 1) + Fib(n − 2)
```


Image source: https://en.wikipedia.org/wiki/Golden_spiral

Recursion

Example 2: Fibonacci sequence (cont'd)

- The recursion equation (递归方程) for the number of moves that solve the nth Fibonacci term is:

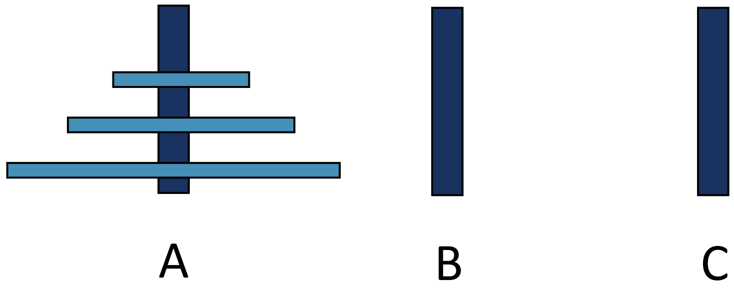
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + 1 & \text{if } n > 1 \end{cases}.$$

- Is it efficient to calculate the nth Fibonacci term by recursion?
  - When calculating $\mathrm{Fib}(5)$, how many times of $\mathrm{Fib}(3)$ and $\mathrm{Fib}(2)$ is calculated?

Recursion

Example 3: Towers of Hanoi (汉诺塔)

- Objective: Transfer disks from pole A to pole C.
- Rules: Only move one disk at a time, and can’t put a bigger disk on a smaller one.



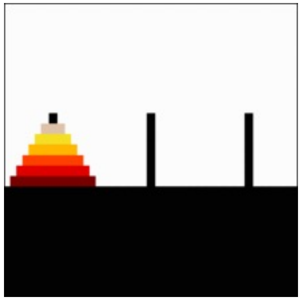


Image source: https://en.wikipedia.org/wiki/Tower_of_Hanoi

Example 3: Towers of Hanoi (cont'd)

- The recursive function $\text{Hanoi}(n, A, B, C)$ means moving n disks from pole A to pole C using B as auxiliary.
- Steps:
  - Move $n-1$ disks from A to B, using C as auxiliary.
  - Move the disk left on A directly to C.
  - Move the $n-1$ disks from B to C, using A as auxiliary.

```
Hanoi(n, A, B, C)
    if n = 1 then move(A, C)
    else
        Hanoi(n − 1, A, C, B)
        move(A, C)
        Hanoi(n − 1, B, A, C)
```

Illustration of recursion calls for $n = 3$

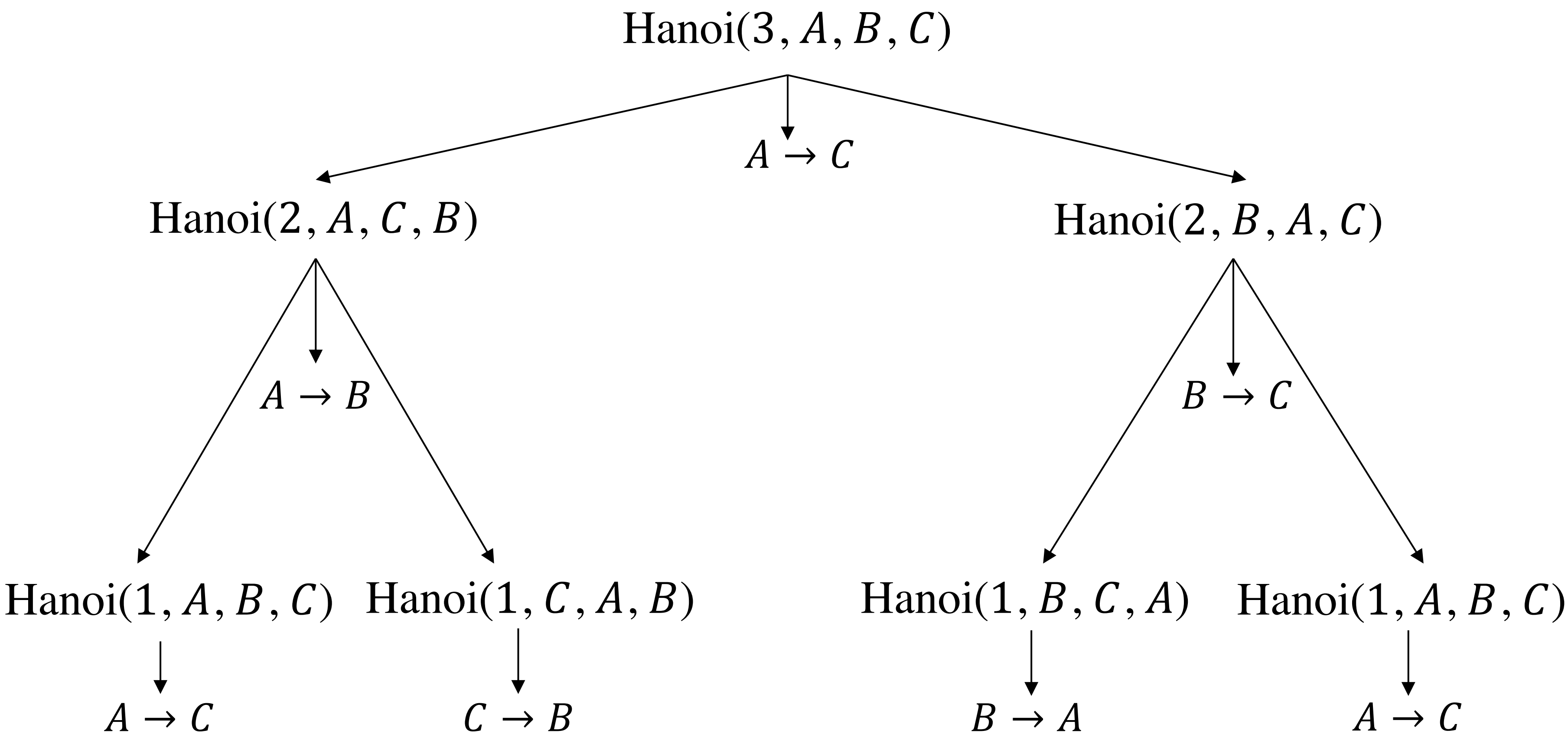


Illustration of recursion instances for $n = 4$

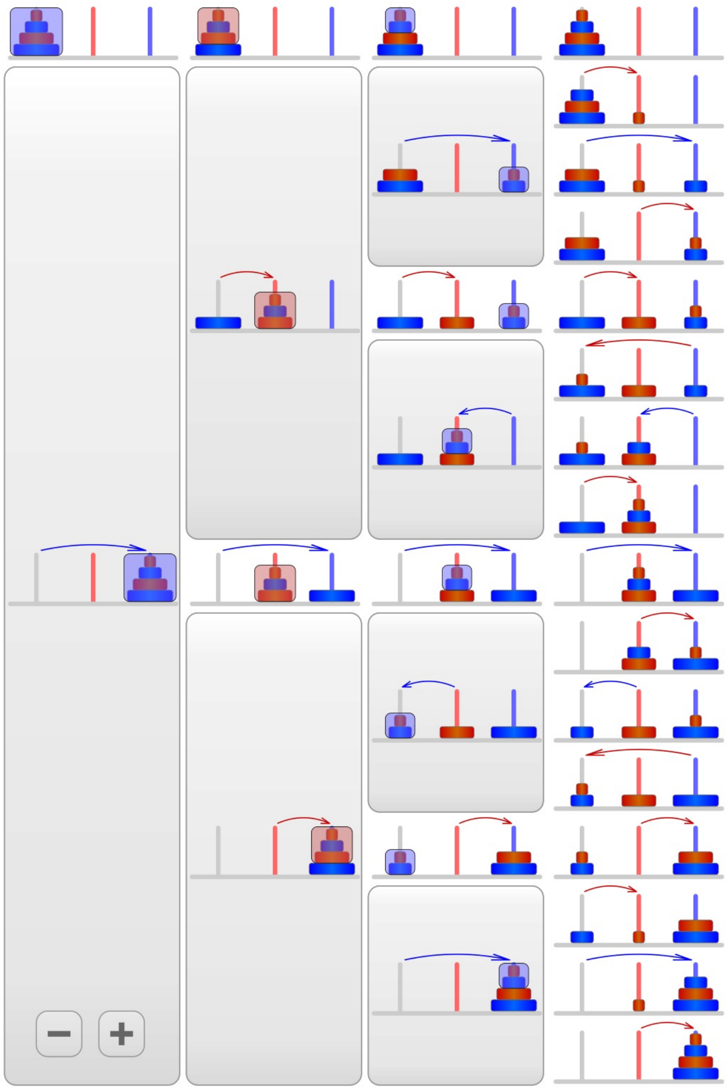


Image source: https://en.wikipedia.org/wiki/Tower_of_Hanoi

Recursion

Example 3: Towers of Hanoi (cont'd)

- The recursion equation for the number of moves that solve Towers of Hanoi is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n-1) + 1 & \text{if } n > 1 \end{cases}.$$

- However, it is a recursion equation, rather than a function of n. How to convert it as a function of n?
  - Recall what we have learned in discrete mathematics: characteristic equation (特征方程) with characteristic root (特征根).

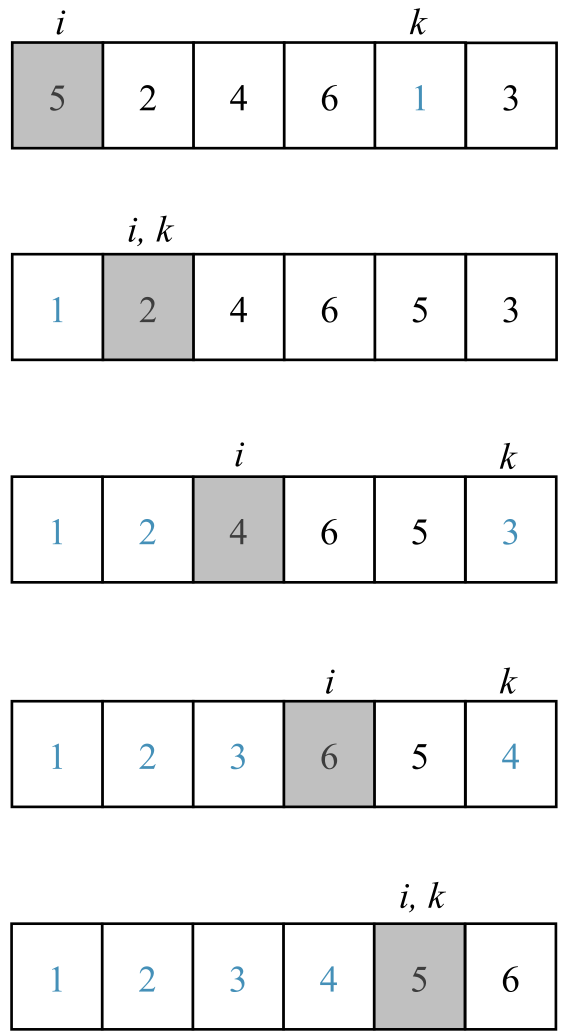
Recursion

Example 4: Selection sort (选择排序)

Similar to insertion sort, selecion sort is a very straightforward sorting algrotihm.

- Start with an empty left hand and the cards face down on the table.
- Then remove the smallest card at a time from the table, and insert it into the rightmost in the left hand.
- At all times, the cards held in the left hand are sorted.

```
SelectionSort(A)
for i ← 1 to n − 1 do
    k ← i
    for j ← i + 1 to n do
        if A[j] < A[k] then
            k ← j
    if k ≠ i then A[i] ↔ A[k]
```


i k
5 2 4 6 1 3
i, k
1 2 4 6 5 3
i k
1 2 4 6 5 3
i k
1 2 3 6 5 4
i, k
1 2 3 4 5 6

Recursion

Example 4: Selection sort (cont'd)

- The recursive version of selection sort is very easy to convert.
- Replace the outer loop by a recursive call.
  - Because we are actually doing the same thing for each subsequence $A[i \dots n]$.
- Although it works, it is not elegant at all as a recursive algorithm.

Usually, we only write the changing variables as the arguments of a recursive function in pseudocode.

RecursiveSelectionSort(i)

if $i \geq n$ **then return** 0
else
    $k \leftarrow i$
    for $j \leftarrow i + 1$ **to** n **do**
        if $A[j] < A[k]$ **then**
            $k \leftarrow j$
    if $k \neq i$ **then** $A[i] \leftrightarrow A[k]$
    RecursiveSelectionSort($i + 1$)

Recursion

Example 4: Selection sort (cont'd)

- Selecting the minimal one among n elements needs $n-1$ comparisons.
- Therefore, the recursion equation is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + (n-1) & \text{if } n > 1 \end{cases}.$$

Recursion

Example 5: Generating permutations

Goal: Generate all $n!$ permutations of sequence $\langle 1,2,\ldots,n\rangle$.

- What is a proper small instance of this problem?
  - Get all permutation of a sequence with $n-1$ elements.
- Given the solution of a small instance, how to solve the original problem?
  - Get all permutation of the sequence with n elements by the ones with $n-1$ elements.

Recursion

Example 5: Generating permutations

Idea 1: Put different elements on fixed position.

- Suppose we can generate all permutations for $n - 1$ numbers.
- Generate all the permutations of the numbers $2, 3, \ldots, n$ and add the number 1 to the beginning of each permutation (the ones starting with 1).
- Next, generate all permutations of the numbers $1, 3, \ldots, n$ and add the number 2 to the beginning of each permutation (the ones starting with 2).
- Repeat this procedure until finally the permutations of $1, 2, 3, \ldots, n - 1$ are generated and the number n is added at the beginning of each permutation.

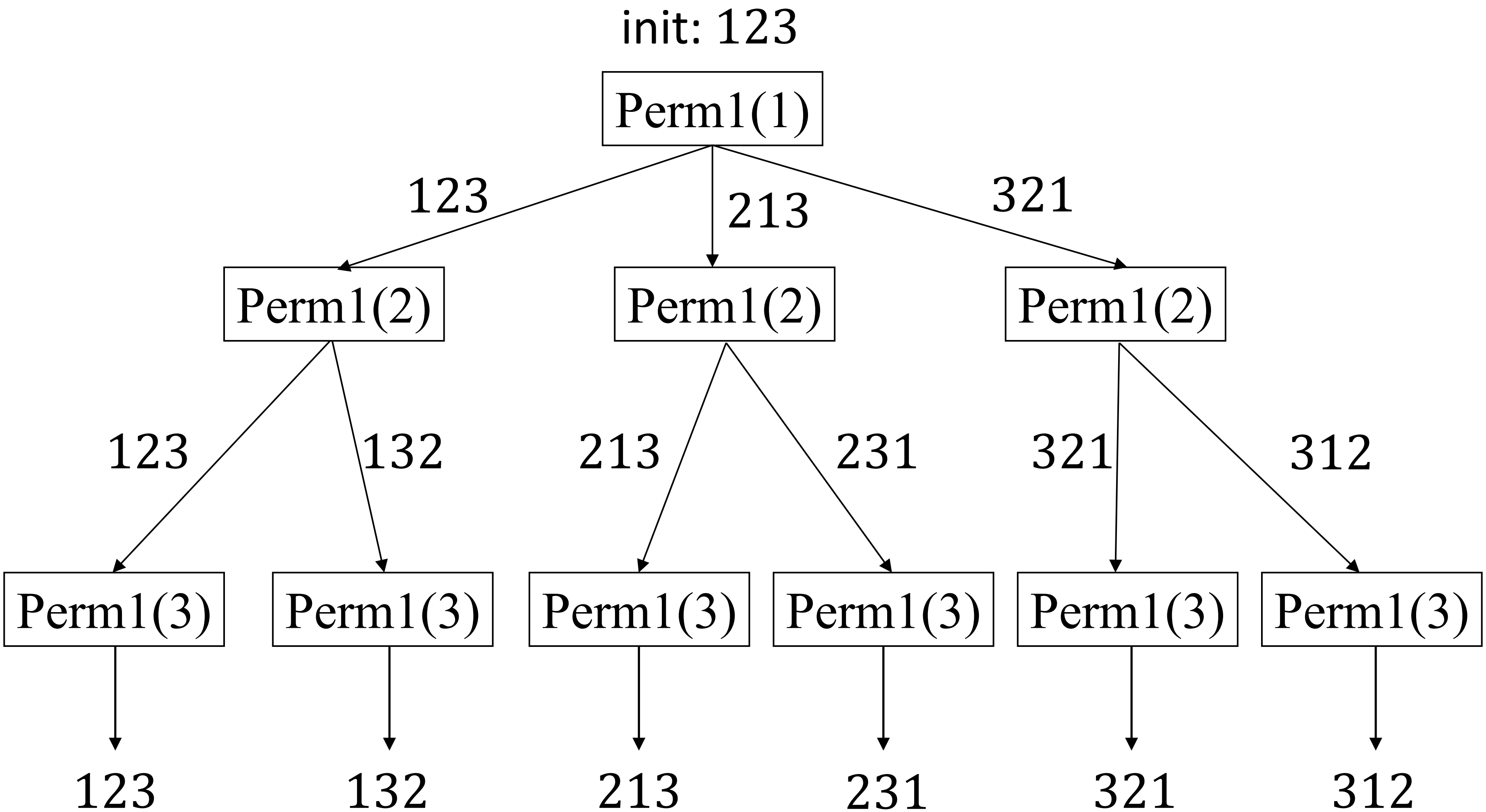
Example 5: Generating permutations (cont'd)

```
Perm1(m)
if m = n then output P[1..n]
else
    for j ← m to n do
        P[j] ↔ P[m]
        Perm1(m + 1)
        P[j] ↔ P[m]
```

```
GeneratingPerm1()
for j ← 1 to n do
    P[j] ← j
Perm1(1)
```

Must switch back. Otherwise it will be messed up!

Illustration of recursion calls for $n = 3$



Try $n = 4$ by yourself

Recursion

Example 4: Generating permutations (cont'd)

Idea 2: Put fixed element on different positions.

- Suppose we can generate all permutations of the numbers $1,2,\ldots,n-1$.
- First, we put n in $P[1]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[2\ldots n]$.
- Next, we put n in $P[2]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[1]$ and $P[3\ldots n]$.
- Then, we put n in $P[3]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[1\ldots 2]$ and $P[4\ldots n]$.
- Repeat the above process until finally we put n in $P[n]$ and generate all the permutations of the first $n-1$ numbers using the subarray $P[1\ldots n-1]$.

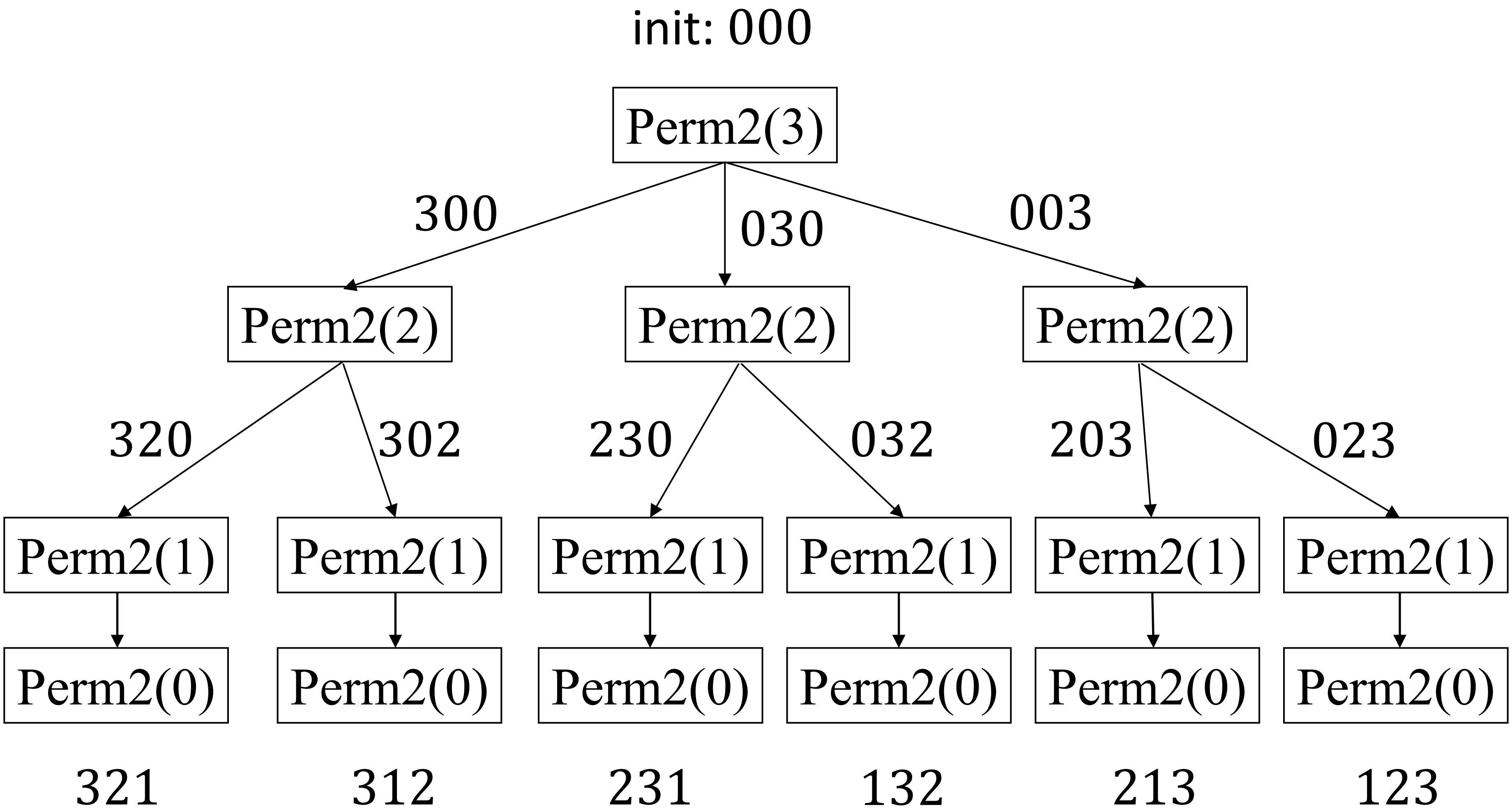
Example 5: Generating permutations (cont'd)

```
Perm2(m)
if m = 0 then output P[1..n]
else
    for j ← 1 to n do
        if P[j] = 0 then
            P[j] ← m
            Perm2(m − 1)
            P[j] ← 0
```

```
GeneratingPerm2()
for j ← 1 to n do
    P[j] ← 0
Perm1(n)
```

Must reset to 0. Otherwise the positions are not enough.

Illustration of recursion calls for $n = 3$



Try $n = 4$ by yourself

Example 5: Generating permutations (cont'd)

- For both ideas, each instance is split into n smaller instance with size $n-1$.
- Therefore, the recursion equation is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ n(T(n-1)+1) & \text{if } n > 1 \end{cases}.$$

Classroom Exercise

Write the pseudocode of recursive linear search.

Classroom Exercise

Solution:

```
RecursiveLinearSearch(i)
if i > n then return 0
if A[i] = x then
    return i
else
    return RecursiveLinearSearch(i + 1)
```

Recursive Analysis

- Goal of recursion analysis: obtain an asymptotic bound Θ or O from the the recursive equation of a recursive algorithm.

$$T(n) = g\big(T(n-k)\big) \quad \text{or} \quad T(n) = g\big(T(n/k)\big)$$

$$\downarrow$$

$$T(n) = f(n)$$

Overview of Recursive Analysis Methods

- Substitution method (替换方法)
  - Guess a bound (directly guess or based on recursion tree);
  - Prove our guess correct using Mathematical Induction.
- Master method (公式法)
  - A theorem with three cases;
  - In each case, the result can be directly obtained without calculation.

Technicalities

In practice, we neglect certain technical details when we state and solve recursion. It won’t affect the final asymptotic results.

- Suppose n is an non-negative integer in $T(n)$.
- Omit floors and ceiling.
  - E.g. $T(n) = 2T(\lceil n/2 \rceil)$, and $T(n) = 2T(\lfloor n/2 \rfloor)$ are equivalent to $T(n) = 2T(n/2)$.
- As n is sufficiently small, we regard $T(n) = T(1)$, where $T(1)$ denotes the constant.
  - We can simply set $T(1) = 1$ and $T(0) = 0$.

Substitution Method

Steps of substitution method:

1. Guess the form of the solution.
2. Use mathematical induction (数学归纳法) to find the constants and show that the solution works.

Substitution Method

Example 6

Consider the recursion equation for the number of comparisons of recursive selection sort:

$$T(n) = T(n-1) + (n-1)$$

1. Guess $T(n) = O(n^2)$.
2. Prove: $T(n) \leq cn^2$:
   - Base case: When $n = 1$, $T(1) = 1 \leq c1^2$, for choosing $c \geq 1$.
   - Inductive step: Suppose $T(n-1) \leq c(n-1)^2$.

$$\begin{aligned} T(n) &\leq c(n-1)^2 + n - 1 \\ &= cn^2 - 2cn + c + n - 1 \\ &\leq cn^2 - 2cn + 2c + n - 1 \\ &= cn^2 - (2c-1)(n-1) \\ &\leq cn^2 \text{ (for } c \geq \tfrac{1}{2}) \end{aligned}$$

Substitution Method

Example 7

Consider the recursion equation for the number of moves that solve Towers of Hanoi:

$$T(n) = 2T(n-1) + 1$$

1. Guess $T(n) = O(2^n)$.
2. Prove: $T(n) \le c2^n$:
   - Base case: When $n = 1$, $T(1) = 1 \le c2^1$, for choosing $c \ge \frac{1}{2}$.
   - Induction step: Suppose $T(n-1) \le c2^{n-1}$.

$$\begin{aligned} T(n) &\le 2c2^{n-1} + 1 \\ &= c2^n + 1 \\ &\cancel{\le c2^n}. \end{aligned}$$

   - $T(n) \le c2^n + 1$ can't imply $T(n) \le c2^n$. How can we do?
     (loose) (tight)

Substitution Method

- Sometimes the guess is correct, but somehow the math doesn't seem to work out in the induction.
- Usually, the problem is that the inductive assumption isn't strong enough to prove the detailed bound.
- Revise the guess by subtracting a lower-order term often permits the math to go through.

Substitution Method

Example 7 (cont'd)

- Consider the recursion equation for the number of moves f that solve Towers of Hanoi:

$$T(n) = 2T(n-1) + 1$$

1. Guess $T(n) = O(2^n)$.
2. Prove: $T(n) \leq c2^n - b$:
   - Base case: When $n = 1$, $T(1) = 1 \leq c2^1 - b$, for choosing $c \geq \frac{1+b}{2}$.
   - Induction step: Suppose $T(n-1) \leq c2^{n-1} - b$.

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b) + 1 \\ &= c2^n - 2b + 1 \\ &\leq c2^n - b \text{ (for } b \geq 1\text{)}. \end{aligned}$$

   - $T(n) \leq c2^n - b$ can derive $T(n) \leq c2^n$. Therefore $T(n) = O(n)$ is proved.
     (tight) (loose)

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Classroom Exercise

Use substitution method to give the asymptotic bound of the following recursive equation:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

Classroom Exercise

Solution:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

1. Guess $T(n) = O(n)$
2. Prove: $T(n) \le cn - b$:

- Base case: When $n = 1$, $T(1) = 1 \le c - b$, for choosing any $c \ge 1 + b$.
- Inductive step: Suppose $T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor - b$ and $T(\lceil n/2 \rceil) \le c\lceil n/2 \rceil - b$.

$$\begin{aligned} T(n) &\le c\lfloor n/2 \rfloor - b + c\lceil n/2 \rceil - b + 1 \\ &= cn - 2b + 1 \\ &\le cn - b \ (\text{for } b \ge 1) \end{aligned}$$

- $T(n) \le cn - b$ can derive $T(n) \le cn$. Therefore $T(n) = O(n)$ is proved.

Substitution Method

Example 8

$$T(n) = 8T(n/2) + 5n^2$$

1. Guess $T(n) = O(n^3)$.
2. Prove: $T(n) \le cn^3$:
   - Base case: When $n = 1$, $T(1) = 1 \le c$, for choosing any $c \ge 1$.
   - Inductive step: Suppose $T(n/2) \le c(n/2)^3$.

$$\begin{aligned} T(n) &\le 8c(n/2)^3 + 5n^2 \\ &= cn^3 + 5n^2 \end{aligned}$$

   - $T(n) \le cn^3 + 5n^2$ can't prove $T(n) \le cn^3$. We should subtract a lower-order term.

Substitution Method

Example 8 (cont'd)

$$T(n) = 8T(n/2) + 5n^2$$

1. Guess $T(n) = O(n^3)$.
2. Prove: $T(n) \leq cn^3 - bn^2$:
   - Base case: When $n = 1$, $T(1) = 1 \leq c - b$, for choosing any $c \geq 1 + b$.
   - Inductive step: Suppose $T(n/2) \leq c(n/2)^3 - b(n/2)^2$.

$$\begin{aligned} T(n) &\leq 8[c(n/2)^3 - b(n/2)^2] + 5n^2 \\ &= cn^3 - 2bn^2 + 5n^2 \\ &= cn^3 - bn^2 - bn^2 + 5n^2 \\ &\leq cn^3 - bn^2 \text{ (for } b \geq 5) \end{aligned}$$

   - $T(n) \leq cn^3 - bn^2$ can derive $T(n) \leq cn^3$. Therefore $T(n) = O(n^3)$ is proved.

Substitution Method

Example 9

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

1. Guess $T(n) = O(n)$.
2. Prove: $T(n) \leq cn$:
   - Base case: When $n = 1$, $T(1) = 1 \leq c1$, for choosing $c \geq 1$.
   - Inductive step: Suppose $T(n/2) \leq 2c(n/2)$.

$$\begin{aligned} T(n) &\leq cn + n \\ &= O(n)? \end{aligned}$$

   - Wrong! The error is that we haven't proved the exact form of the inductive hypothesis, i.e. $T(n) \leq cn$.
   - Try subtracting a lower order term?

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学 计算机科学系 Computer Science Department of Xiamen University

Substitution Method

Example 9 (cont'd)

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

1. Guess $T(n) = O(n \lg n)$.
2. Prove: $T(n) \le cn \lg n$:
   - Base case: When $n = 2$, $T(2) = 2T(1) + 2 = 4 \le c2 \lg 2$, for choosing $c = 2$.
   - Inductive step: Suppose $T(\lfloor n/2 \rfloor) \le c(\lfloor n/2 \rfloor)\lg(\lfloor n/2 \rfloor)$.

$$\begin{aligned} T(n) &\le 2\, c(\lfloor n/2 \rfloor)\lg(\lfloor n/2 \rfloor) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n \quad (\text{for } c \ge 1) \end{aligned}$$

Substitution Method

Example 9 (cont'd)

- In the above proof, we set $n = 2$ at the base case.
- Actually, we usually don't need to set $n = 1$ for all base cases, because it sometimes doesn't work.
  - e.g. can't prove $T(1) = 1 \leq c1\lg 1 = 0$.
- The asymptotic analysis only requires us to prove for some $n \geq n_0$. Therefore, it is ok to set $n = 2$ or $n = 3$ at the base case.

Substitution Method: Changing Variables

Sometimes, a little algebraic manipulation can make an unknown recursion similar to one you have seen before.

Example 10

$$T(n) = 2T\left(\left\lfloor\sqrt{n}\right\rfloor\right) + \lg n$$

- Renaming $m = \lg n$ yields $n = 2^m$ and:

$$T(2^m) = 2T\left(2^{m/2}\right) + m.$$

- We can now rename $S(m) = T(2^m)$ to produce the new recursion:

$$S(m) = 2S(m/2) + m,$$

  which has a solution of $S(m) = O(m \lg m)$.

- Changing back from $S(m)$ to $T(n)$, we obtain:

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n).$$

Substitution Method

How to make a good guess:

- Bad News:
  - No general way to guess the correct solutions to recursion.
  - Good guess = E (experience) + C (creativity) + L (luck).
- Good News:
  - Recursion tree often generates good guesses.

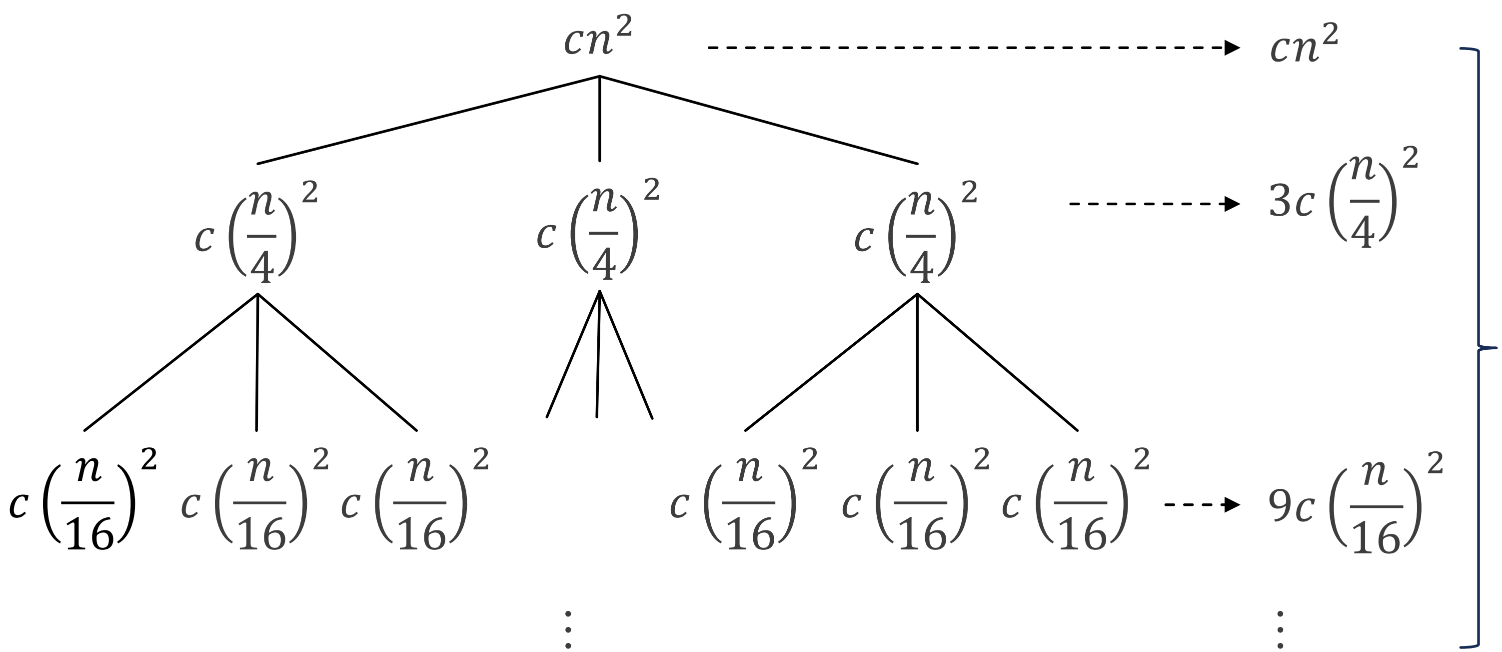
Recursion Tree

- The recursion-tree is a straightforward way to devise a good guess.
- Recursion trees are particularly useful when the recurrence describes the running time of a divide-and-conquer algorithm.
- In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
  1. We sum all the per-node costs within each level of the tree to obtain a set of *per-level costs*;
  2. We sum all the per-level costs to determine the total cost of all levels of the recursion.
- Notice: Recursion tree only provides a guess. It is not a strict proof. Substitution method is still needed after we guess a bound by recursion tree.

Recursion Tree

Example 11

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$$



What is the height of the tree?

Recursion Tree

Example 11 (cont'd)

- The cost sequence of each level is:

$$cn^2, c(n/4)^2, c(n/4^2)^2, \dots, c(n/4^i)^2$$

- Denote height of the recursion tree as k.
- The node at the leaf of the tree is 1. Therefore the leaf is achieved when $\left(n/4^k\right)^2 = 1$ and thus $k = \log_4 n$.

We can simply assume that n is an exact power of 4.

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY
厦门大学计算机科学系
Computer Science Department of Xiamen University

Recursion Tree

Example 11 (cont'd)

- Summing up all levels, the total cost is:

$$
\begin{aligned}
T(n) &= cn^2 + 3c\left(\frac{n}{4}\right)^2 + 9c\left(\frac{n}{16}\right)^2 + 27c\left(\frac{n}{64}\right)^2 + \cdots \\
&= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 \\
&= \sum_{i=0}^{\log_4 n}\left(\frac{3}{16}\right)^i cn^2 < \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 = \frac{1}{1-3/16}cn^2 = O(n^2)
\end{aligned}
$$

Formula of infinity geometric series (无穷几何级数)

Recursion Tree

Example 11 (cont'd)

- Notice again: Recursion tree only provides a guess. It is not a strict proof. We still need substitution method:

1. Guess $T(n) = O(n^2)$.
2. Prove: $T(n) \leq dn^2$:

Why do we use d here rather than c?

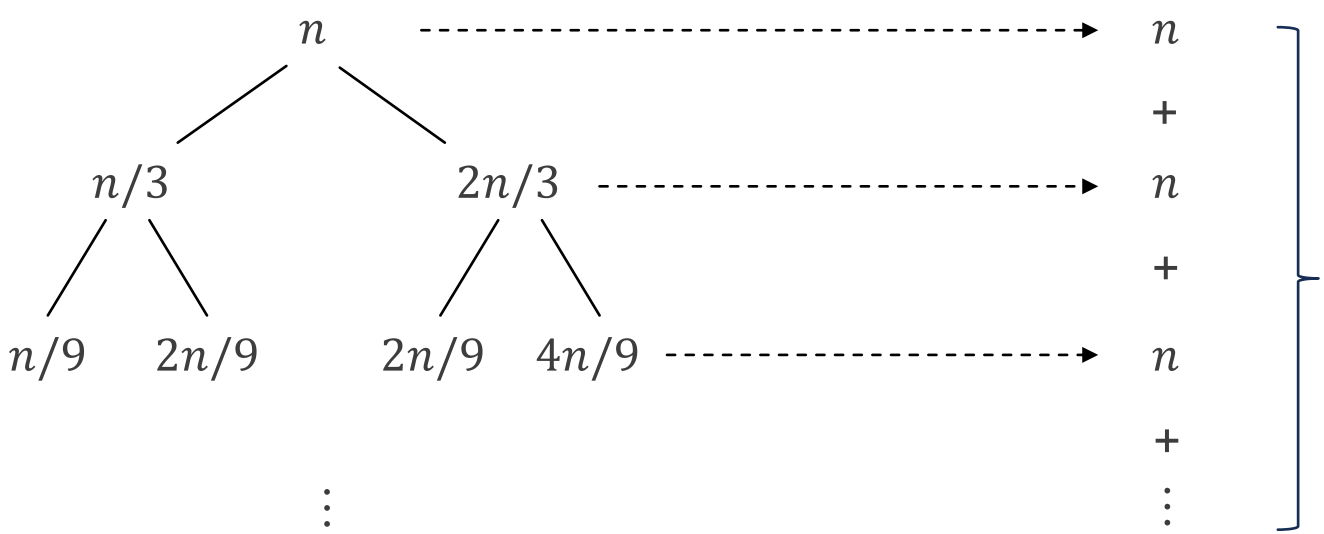
- Base case: When $n = 1$, $T(1) = 1 \leq d1^2$, for choosing $d \geq 1$.
- Inductive step: Suppose $T(n/4) \leq d(n/4)^2$.

$$
\begin{aligned}
T(n) &\leq 3d\left(\frac{n}{4}\right)^2 + cn^2 \\
&= \frac{3}{16}dn^2 + cn^2 \\
&\leq dn^2 \ (\text{for } d \geq \frac{16}{13}c)
\end{aligned}
$$

Recursion Tree

Example 12

$$T(n) = T(n/3) + T(2n/3) + n$$



What is the height of the tree?

Recursion Tree

Example 12 (cont'd)

- If there are different decreasing rate, e.g. $n/3$ and $2n/3$ in this example, we should determine the slowest deceasing rate.
  - The one with slowest deceasing rate goes deepest.
- $2n/3$ is the slowest one. Therefore, the height is calculated by:

$$\left(\frac{2}{3}\right)^k n = 1$$
$$k = \log_{3/2} n$$

- As observed from the tree, the cost of each level is n. But not all levels have cost n because some branches with faster decreasing rate may reach the leaves earlier. The total cost is:

$$T(n) \leq n(k+1) \leq n(\log_{3/2} n + 1) = O(n \lg n).$$

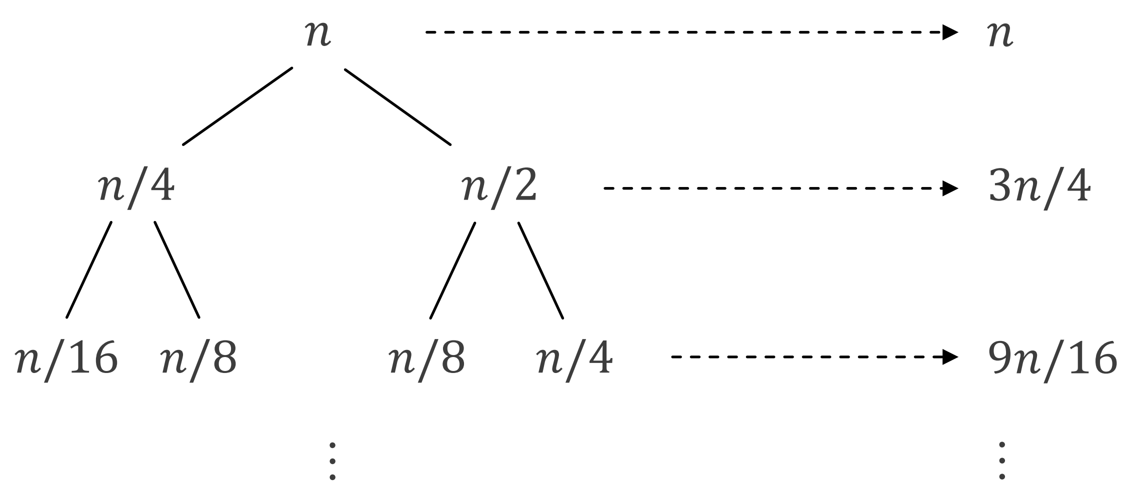
Classroom Exercise

Use recursion tree to guess the asymptotic bound of the following recursion equation:

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

Classroom Exercise

Solution:



- The slowest deceasing rate is $n/2$.
- The height is calculated by: $(1/2)^k n = 1$ and $k = \lg n$.

$$T(n) \le n + \frac{3}{4}n + \left(\frac{3}{4}\right)^2 n + \cdots + \left(\frac{3}{4}\right)^{\lg n} n$$
$$< \frac{1}{1 - 3/4} n = 4n = O(n).$$

Master Method

- The master method provides a "cookbook" method for solving recurrences of the form

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

  - $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.
- The recursion form describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b.
- The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$.

Master Method

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 recursion

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically with three cases:

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

Master Method

What does the master theorem mean?

- In each of the three cases, we are comparing $f(n)$ with $n^{\log_b a}$.
- Intuitively, the solution to the recursion is determined by the order of these two functions.
  - If, as in case 1, $n^{\log_b a}$ has high order, then the solution is $T(n) = \Theta(n^{\log_b a})$.
  - If, as in case 2, the two functions are the same order, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n)$.
  - If, as in case 3, $f(n)$ has high order, then the solution is $T(n) = \Theta\big(f(n)\big)$.

Master Method

In short:

- Comparing $f(n)$ with $n^{\log_b a}$, choose the larger order one with big Θ.
- If they have the same order, multiply with $\lg n$.

Master Method

Take a deeper look of the master theorem. Beyond this intuition of comparing order of functions, there are some technicalities that must be understood.

- In case 1, not only must $f(n)$ have lower order than $n^{\log_b a}$, its order must be polynomially lower.
  - The order of $f(n)$ must be asymptotically lower than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant $\epsilon > 0$.
- In case 3, not only must $f(n)$ have higher order than $n^{\log_b a}$, its order must be polynomially higher, and in addition satisfy the "regularity" condition that $af(n/b) \leq cf(n)$.
  - The order of $f(n)$ must be asymptotically higher than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant $\epsilon > 0$.
  - No worry about $af(n/b) \leq cf(n)$, it holds for most of the cases.

Master Method

Example 13

$$T(n) = 9T(n/3) + n$$

- We have $a = 9$, $b = 3$, $f(n) = n$, and thus we have $n^{\log_b a} = n^{\log_3 9} = n^2$.
- We thus compare n and n^2.
- Since $f(n) = n = O(n^{\log_3 9-\epsilon})$ for $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^2)$.

Master Method

Example 14

$$T(n) = T(2n/3) + 1$$

- We have $a = 1$, $b = 3/2$, $f(n) = 1$, and thus we have $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
- We thus compare 1 and 1.
- Since $f(n) = 1 = \Theta(1)$, we can apply case 2 and thus the solution to the recursion is $T(n) = \Theta(\lg n)$.

Master Method

Example 15

$$T(n) = 3T(n/4) + n \lg n$$

- We have $a = 3$, $b = 4$, $f(n) = n \lg n$, and thus we have $n^{\log_b a} = n^{\log_4 3} \approx n^{0.793}$.
- We thus compare $n \lg n$ and $n^{\log_4 3}$.
- Since $f(n) = n \lg n = \Omega(n) = \Omega(n^{\log_4 3 + \epsilon})$ for $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$.
- For sufficiently large n,

$$af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n) \text{ for } c = 3/4.$$

- Consequently, by case 3, the solution to the recursion is $T(n) = \Theta(n \lg n)$.

Master Method

- The three cases do not cover all the possibilities for $T(n)$.
- There is a gap between cases 1 and 2 when the order of $f(n)$ is lower than $n^{\log_b a}$ but not polynomially lower.
- Similarly, there is a gap between cases 2 and 3 when the order of $f(n)$ is higher than $n^{\log_b a}$ but not polynomially higher.
- If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recursion.

Master Method

- Master method is used for the following form of recursion equation

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

- We compare $n^{\log_b a}$ with $f(n)$ and select the larger one.
- Therefore, to reduce the cost of a recursive algorithm, we can:
  - Reduce $f(n)$: reduce the cost of computation in each recursion call.
  - Reduce a: reduce the number of recursion calls.
  - Increase b: reduce the size of small instance.

Classroom Exercise

Can we use master method to give the asymptotic bound of the following recursive equation?

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

Classroom Exercise

Solution:

The master method does not apply to the recursion in the following example.

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

- Even though it has the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$.
- We thus compare $n \lg n$ and n.
- It might seem that case 3 should apply, since the order of $f(n) = n \lg n$ is asymptotically higher than n. The problem is that it is not polynomially higher.
- We can't find a constant $\epsilon > 0$ such that $f(n) = n \lg n = \Omega(n^{1+\epsilon}) = \Omega(n \cdot n^{\epsilon})$

Try to compare the order between $\lg n$ and n^{ϵ}

Empirical Experiment

Example 16: Polynomial Evaluation

- Given a polynomial function

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1},$$

  We want to calculate the value of $p(x)$ at some point x_0.

- We can use Horner's rule (秦九韶算法, 霍纳法则) recursively evaluates the polynomial function by rewriting as:

$$p(x) = a_0 + x\Big(a_1 + x\big(a_2 + \cdots + x(a_{n-2} + x a_{n-1}) \ldots\big)\Big).$$

  Let

$$A_i = \begin{cases} a_{n-1} & i = 1 \\ A_{i-1} x_0 + a_{n-i} & i > 1 \end{cases}$$

Empirical Experiment

Example 16 (cont'd)

Horner(A, x_0, i)

if $i = 1$ **then return** a_{n-1}

else

  return $a_{n-i} + x_0 * \text{Horner}(A, x_0, i-1)$

DirectPloy(A, x_0)

$total \leftarrow a_0$

for $i \leftarrow 1$ **to** $n-1$ **do**

  $total \leftarrow total + a_i * \text{power}(x_0, i)$

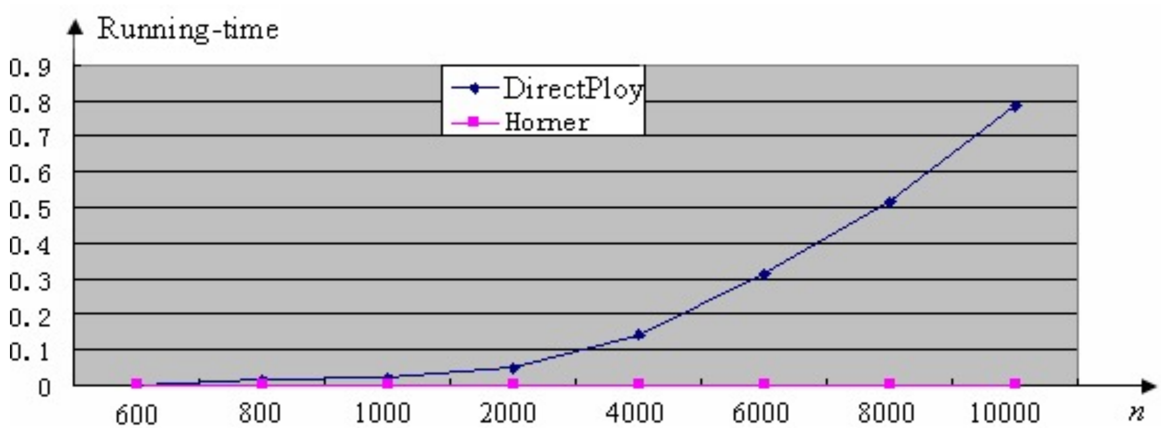
return $total$

Empirical Experiment

Example 16 (cont'd)

- Running-time comparison of DirectPloy and Horner:

n	600	800	1000	2000	4000	6000	8000	10000
DirectPloy	0.0	0.015	0.018	0.046	0.141	0.312	0.515	0.785
Horner	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0



Why?

Conclusion

After this lecture, you should know:

- How to devise a recursive algorithm?
- What is a recursive equation?
- How to derive the asymptotic result from the recursive equation?
- How to draw a recursive tree?

Homework

- Page 48-49

  4.3

  4.5

  4.7

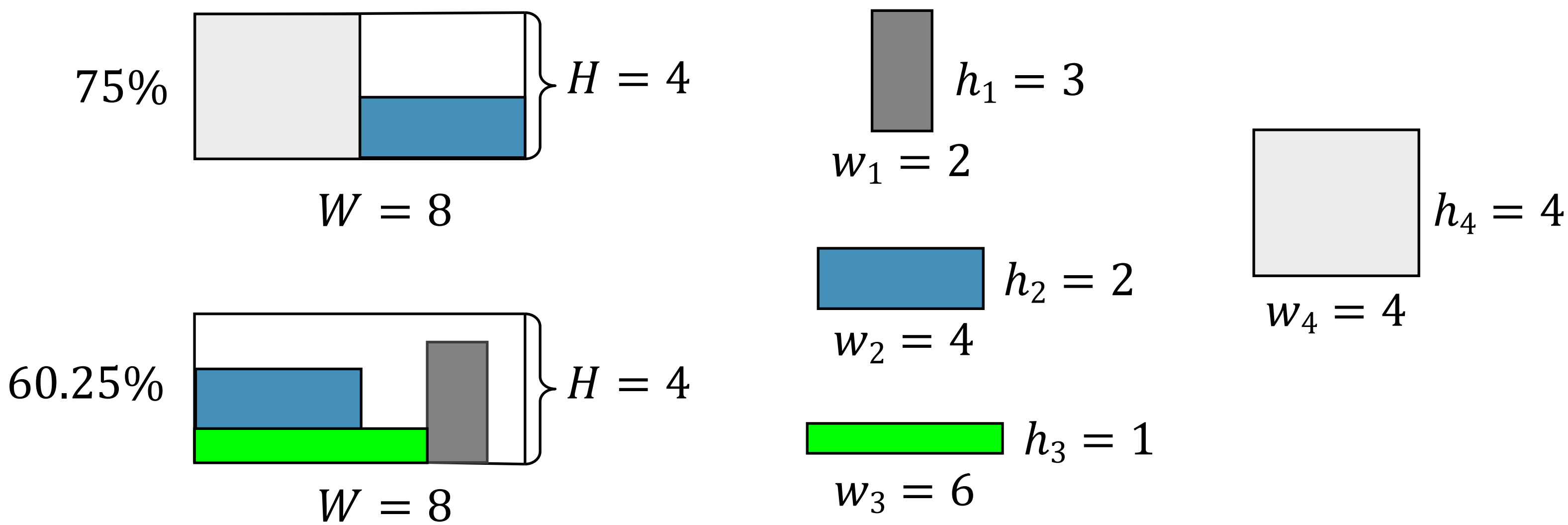
  4.12

  4.15

Experiment

石材切割问题

- 给定一块长为H,宽度为W的石板. 现需要从板上分别切割出n个长度为h_i, 宽度为w_i的石砖. 切割的规则是石砖的长度方向与石板的长度方向保持一致, 同时满足一刀切的约束. 问如何切割使得所使用的石材利用率最高?
- 例如:



Experiment

- 请设计出一个递归算法
- 程序设计
  - 能用图形演示切割的过程 (录制演示视频, 并上传视频文件)
- 数据集在spoc上下载

谢谢

有问题欢迎随时跟我讨论

厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY
厦门大学计算机科学系
Computer Science Department of Xiamen University