CSE221

Recursion

Fall 2021

Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Linear recursion
- Binary recursion
- Multiple recursion

The Recursion Pattern

- **Recursion**: when a method calls itself
  - E.g., factorial function: n! = 1· 2· 3· ··· · (n-1)· n

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- C++ method:

```
// recursive factorial function
int  recursiveFactorial(int n) {
   if  (n  ==  0)  return  1;          // base case
   else return  n  *  recursiveFactorial(n - 1);          // recursive case
}
```

Linear Recursion

- Perform a single recursive call
  - May choose **one** of several recursive cases
  - Should make progress on the base case
- Base cases
  - Should exist at least one
  - Recursion must reach a base case
  - Handling of each base case should not cause recursion

Sum of Array Elements

Algorithm LinearSum(*A, n*):

Input:

An integer array *A* and an integer *n* such that *A* has at least *n* elements

Output:

Sum of the first *n* integers in *A*

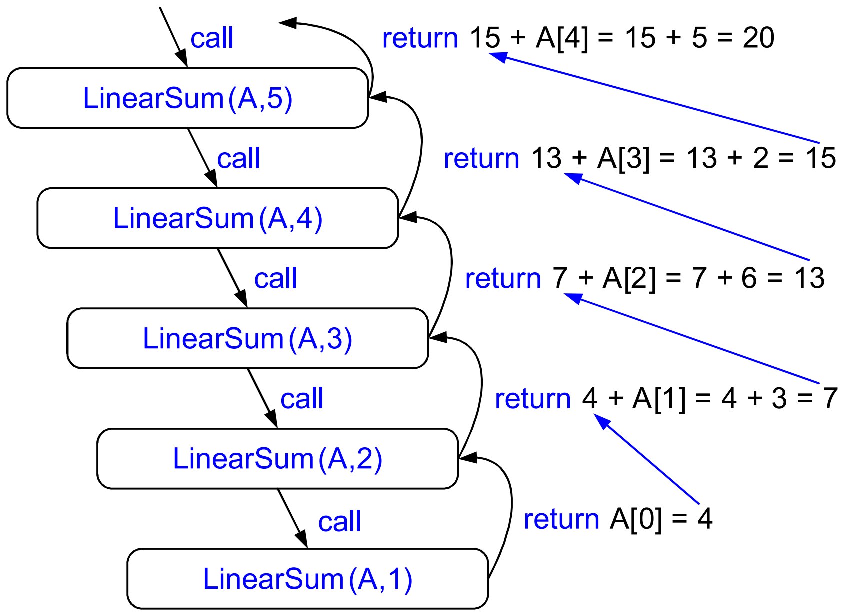
if *n* = 1 **then**

  return *A*[0]

else

  return LinearSum(*A, n* - 1) + *A*[*n* - 1]

Example recursion trace:



A[ ] = {4,3,6,2,5}

Reversing an Array

Algorithm ReverseArray(*A, i, j*):

Input: An array *A* and non-negative integer indices *i* and *j*

Output: The reversal of the elements in *A* starting at index *i* and ending at *j*

if $i < j$ **then**

  Swap $A[i]$ and $A[j]$

  ReverseArray(A, $i + 1$, $j - 1$)

return

Computing Powers

- The power function, $p(x,n) = x^n$

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- O(n) time (for *n* recursive calls)
- Can we do better?

Recursive Squaring

- More efficient linearly recursive algorithm by using repeated squaring

$$p(x,n)=\begin{cases} 1 & \text{if } x=0 \\ x\cdot p(x,(n-1)/2)^2 & \text{if } x>0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x>0 \text{ is even} \end{cases}$$

Recursive Squaring

- More efficient linearly recursive algorithm by using repeated squaring

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

- Examples:

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$

Recursive Squaring

Algorithm Power(x, n):

Input: A number x and integer $n = 0$

Output: The value x^n

if $n = 0$ **then**

  return 1

if n is odd **then**

  y = Power(x, $(n - 1)/2$)

  return $x \cdot y \cdot y$

else

  y = Power(x, $n/2$)

  return $y \cdot y$

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

Recursive Squaring

Algorithm Power(x, n):

Input: A number x and integer $n = 0$

Output: The value x^n

if n = 0 **then**

    return 1

if n is odd **then**

    y = Power(x, (n - 1)/ 2)

    return $x \cdot y \cdot y$

else

    y = Power(x, n/ 2)

    return $y \cdot y$

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

Tail Recursion

- **Definition**: A linearly recursive method that makes its recursive call at its <u>last step</u>
- Tail recursion can be easily converted to non-recursive one (saving resources)

Algorithm ReverseArray(*A, i, j*):
  if *i* < *j* **then**
    Swap *A*[*i*] and *A*[*j*]
    ReverseArray(*A, i+1, j-1*)
  return

Tail Recursion

- **Definition**: A linearly recursive method that makes its recursive call at its <u>last step</u>
- Tail recursion can be easily converted to non-recursive one (saving resources)

Algorithm IterativeReverseArray(*A, i, j*):
    while *i* < *j* **do**
        Swap *A*[*i*] and *A*[*j*]
        $i = i + 1$
        $j = j - 1$
    return

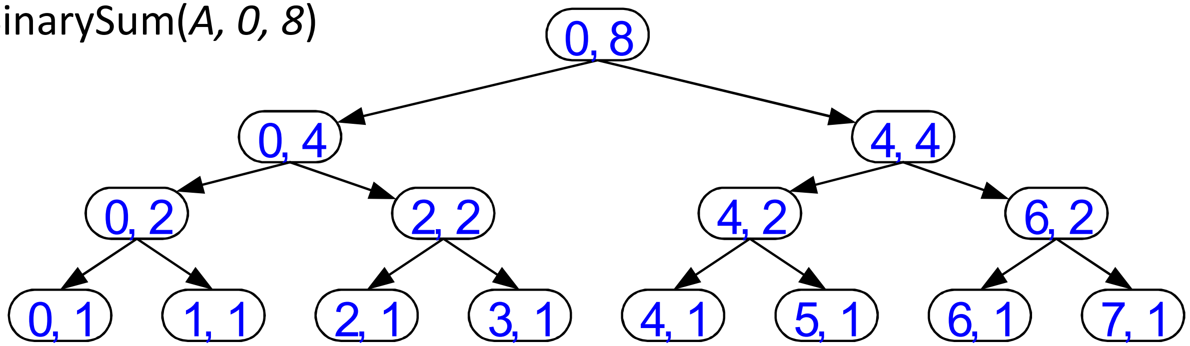
Binary Recursion

- **Definition**: There are **two** recursive calls for each non-base case
- Add all the numbers in an integer array A

Algorithm BinarySum(*A, i, n*):
  Input: An array *A* and integers *i* and *n*
  Output: The sum of the *n* integers in *A* starting at index *i*

  if *n* = 1 **then**
    return *A*[*i*]
  return BinarySum(*A, i, n/* 2) + BinarySum(*A, i* + *n/* 2, *n/* 2)

e.g., BinarySum(*A, 0, 8*)



ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Computing Fibonacci Numbers

- **Definition**:

  $n_0 = 0$
  $n_1 = 1$
  $n_i = n_{i-1} + n_{i-2}$ for $i > 1$

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):
Input: Nonnegative integer k
Output: The kth Fibonacci number F_k
if k <= 1 **then**
  return k
else
  return BinaryFib(k - 1) + BinaryFib(k - 2)

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$

Distance of 2

- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

More than 2 times

A Better Fibonacci Algorithm

- Use linear recursion instead
  - k-1 recursive calls

$F_0 = 0$
$F_1 = 1$
$F_i = F_{i-1} + F_{i-2}$ for $i > 1$

Algorithm LinearFibonacci(k):

Input: A non-negative integer k
Output: Pair of Fibonacci numbers (F_k , F_{k-1})

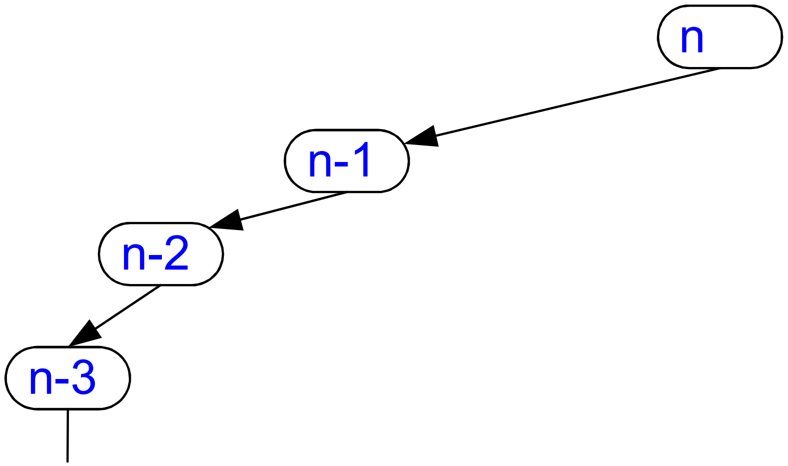
if *k* <= 1 **then**
    return (k, 0)
else
    (i, j) = LinearFibonacci(*k* - 1)
    return (i+j, i)

Multiple Recursion

- Multiple recursion:
  - Makes potentially many recursive calls
  - Not just one or two

Recursion Cost Analysis

- Number of function calls: O(n)
- Height: O(n)

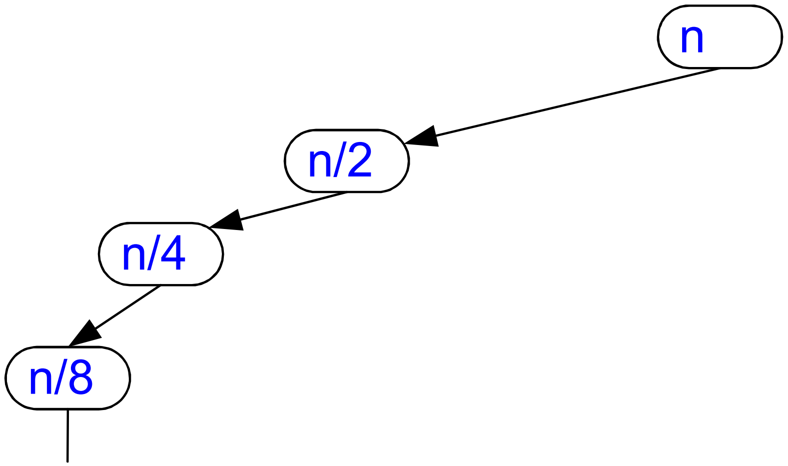


Alright!

Recursion Cost Analysis

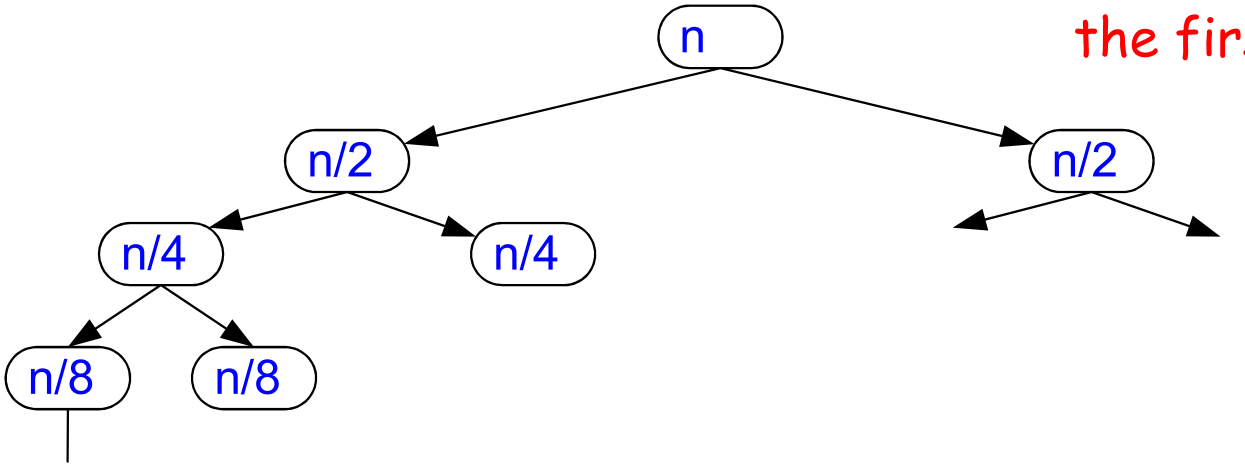
- Number of function calls: O(logn)
- Height: O(logn)

Very good!



Recursion Cost Analysis

- Number of function calls: O(n)
- Height: O(logn)



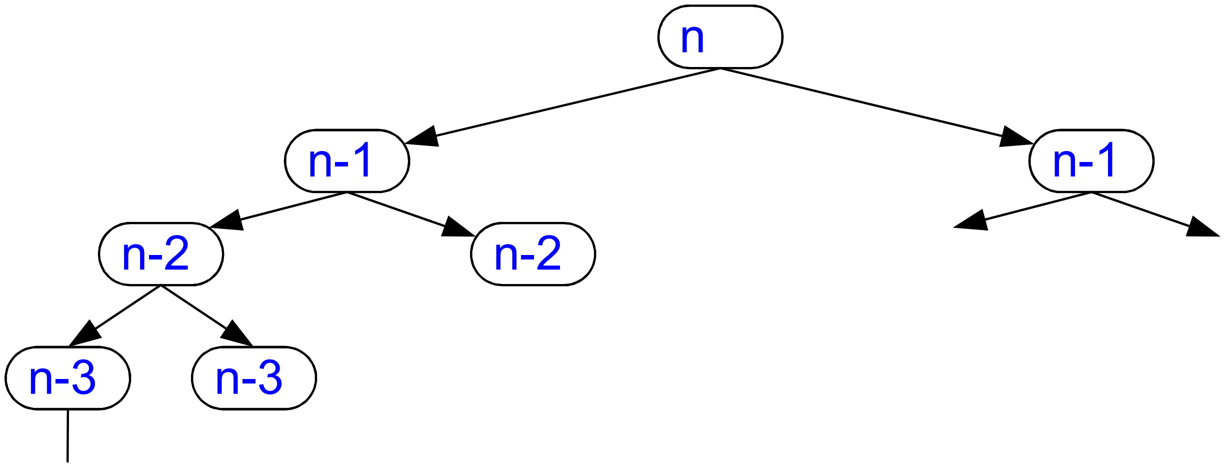
Better than
the first case!

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Recursion Cost Analysis

- Number of function calls: $O(2^n)$
- Height: $O(n)$

Go to the hell..



Questions?