MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

14 Recursion - continued

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

February 16, 2022

Administration

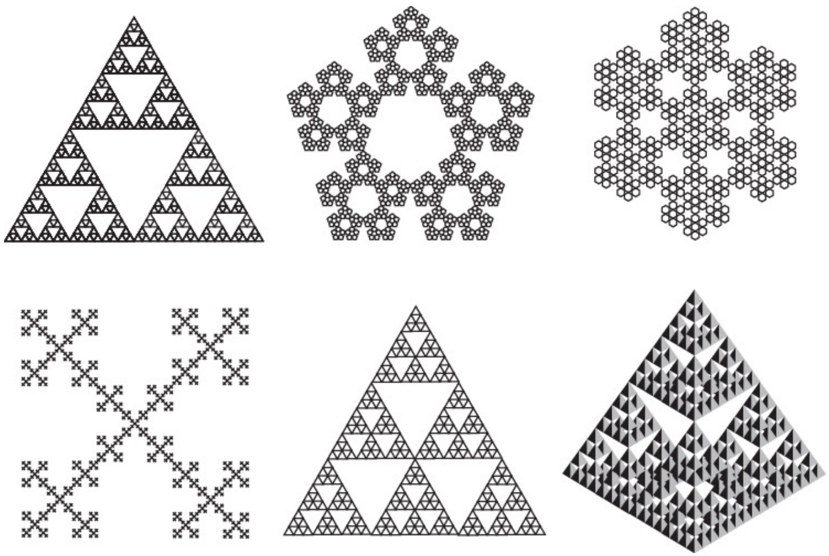
- I will return assignments today! finally!
  - Sorry about the delay
  - TAs did great, I delayed
- My Office Hour:
  - Today at 15:00 in ITB-159 in-person (or virtually using teams as usual)

- We review the recursion quickly

What is Recursion

- **Recursion** is the concept of defining a function that makes a call to itself.
- We call a function **Recursive** if it calls itself.
- When a function calls itself, we refer to this as a **Recursive Call**.
  - if function M calls another function that ultimately leads to a call back to M, this is also a recursive call and function M is recursive.

- A lot of use-cases in real-life:
  - Nature : fractals
  - Mathematics: recursive functions



Recursive Algorithms

- The idea is to avoid loops!
- A recursive implementation can be significantly simpler and easier to understand than an iterative implementation.
- Types of Recursion:
  - Linear Recursion
  - Binary Recursion
  - Tail Recursion
  - Multiple Recursion

Recursive Algorithms - Example

- Factorial of a whole number n is defined as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

  - factorial(4) = 4*(3*2*1) = 4*factorial(3)
  - factorial(4) can be defined in terms of factorial(3)
- Recursive definition of the factorial function is:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

  - base case
    - are non-recursive
  - recursive case
    - no circularity (function terminates)

Recursive Algorithms - Example

- Iterative Factorial

```
int factorial_iter(int n){
    int factorial = 1;
    for (int i = 2; i <= n; i++){
        factorial = factorial * i;
    }
    return factorial;
}
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

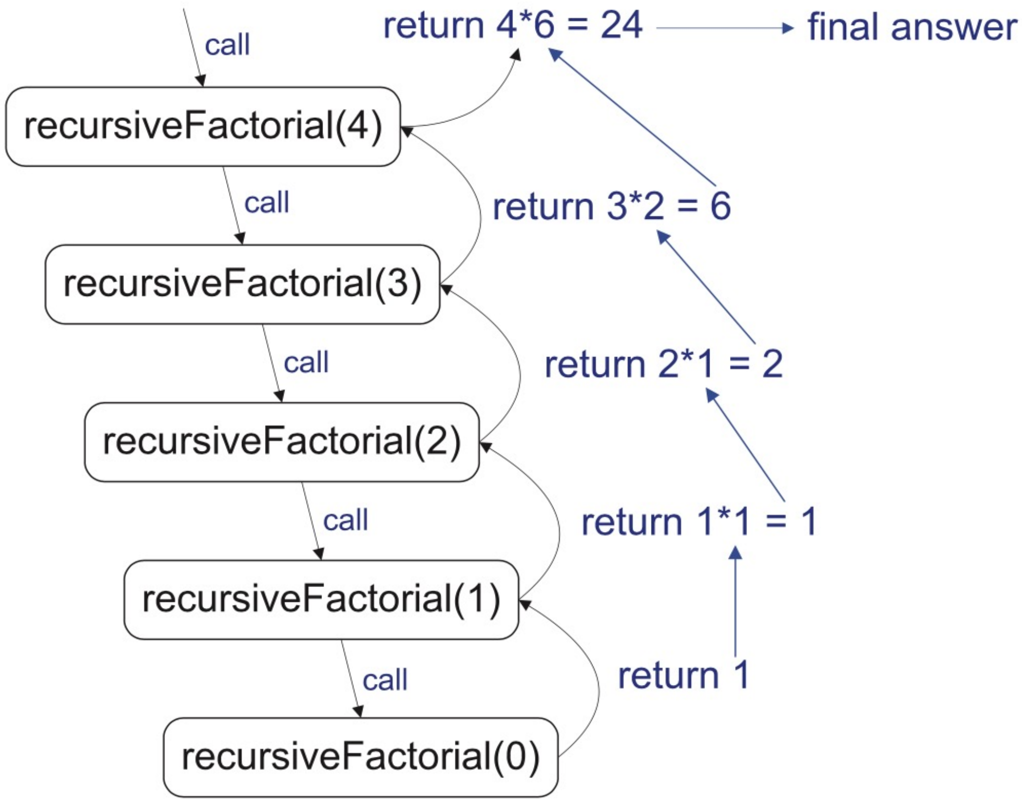
- Recursive Factorial:

```
int factorial_rec(int n){
    if (n==0){
        return 1;
    }
    else{
        return n * factorial_rec(n - 1);
    }
}
```

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Recursive Algorithms - Example

- Recursive Factorial
- **Recursion Trace**:




```
int factorial_rec(int n){
    if (n==0){
        return 1;
    }
    else{
        return n * factorial_rec(n - 1);
    }
}
```

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Linear Recursion

- The simplest form of recursion
- function makes at most one recursive call each time it is invoked
- When we have a sequence, we can view some problems in terms of:
  - a first or last element
  - a remaining sequence that has the same structure as the original sequence

Linear Recursion

- The simplest form of recursion
- function makes at most one recursive call each time it is invoked
- When we have a sequence, we can view some problems in terms of:
  - a first or last element
  - a remaining sequence that has the same structure as the original sequence

Algorithm LinearSum(A, n):

Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements

Output: The sum of the first n integers in A

if $n = 1$ **then**
  return $A[0]$
else
  return LinearSum($A, n-1$) $+ A[n-1]$

Linear Recursion

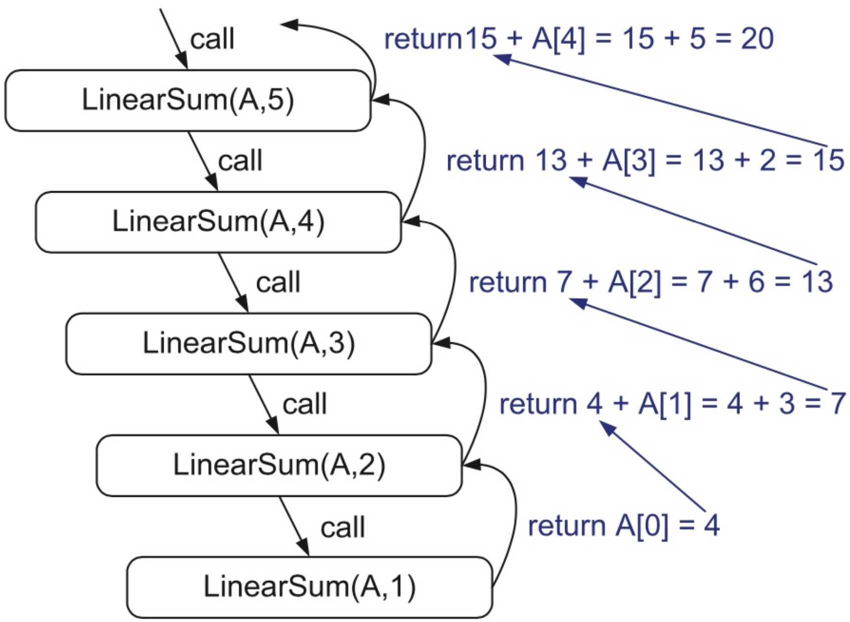
Algorithm LinearSum(A, n):

Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements

Output: The sum of the first n integers in A

if $n = 1$ **then**
  return $A[0]$
else
  return LinearSum$(A, n-1) + A[n-1]$

$A = \{4,3,6,2,5\}$




```
int linearSum_iter(int data[], int n){
    int sum = 0;
    for (int i =0; i < n; i++){
        sum += data[i];
    }
    return sum;
}
```

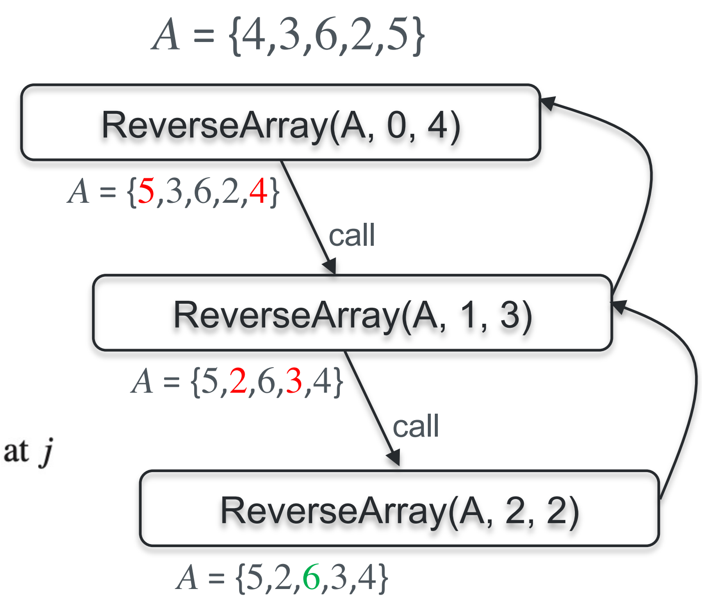
```
int linearSum(int data[], int n){
    if (n <= 0)
        return 0;
    return (linearSum(data, n - 1) + data[n - 1]);
}
```

Recursion in Computer Science

- Recursion is heavily used in Functional Programming
  - In Pure Functional Languages it is the only way to iterate!
    - Like Haskell
  - Non-pure programming languages allow iteration
    - Like Scala
- Recursion has overheads in Imperative Programming Languages
  - Like in C++ and Java
- Pure Functional Language use tail recursion conversions to reduce significant impact on memory consumption of heavy recursive calls
- It has many applications in Big Data Tools
  - Laziness!

Linear Recursion - Reversing an Array

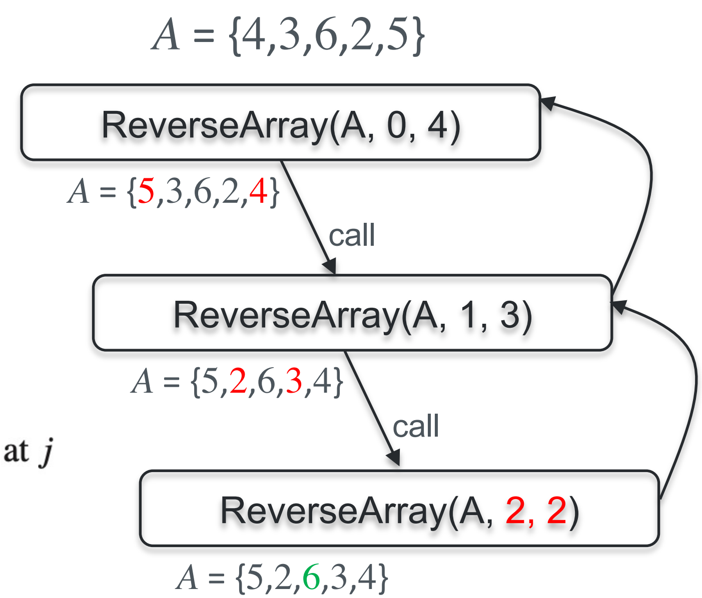
- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?



Algorithm ReverseArray(A, i, j):
    Input: An array A and nonnegative 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

Linear Recursion - Reversing an Array

- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?
  - if **i = j** or **i > j**
    - in either of cases algorithm terminates



Algorithm ReverseArray(A, i, j):

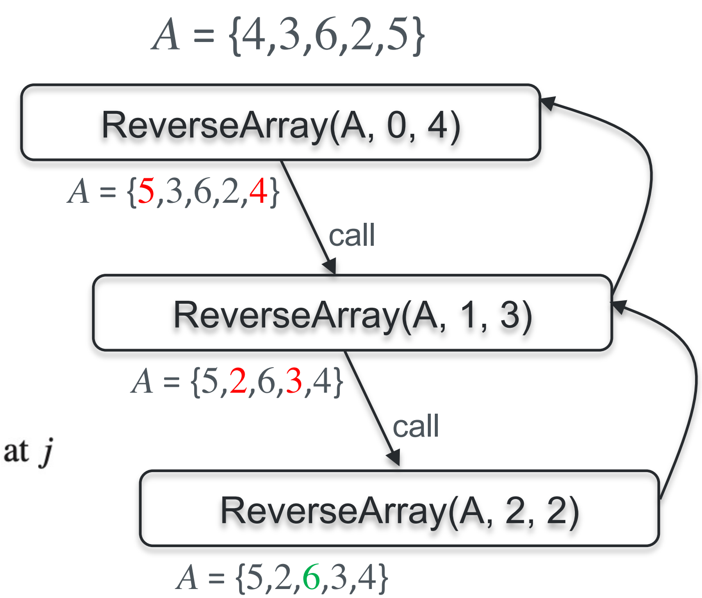
Input: An array A and nonnegative integer indices i and j

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

Linear Recursion - Reversing an Array

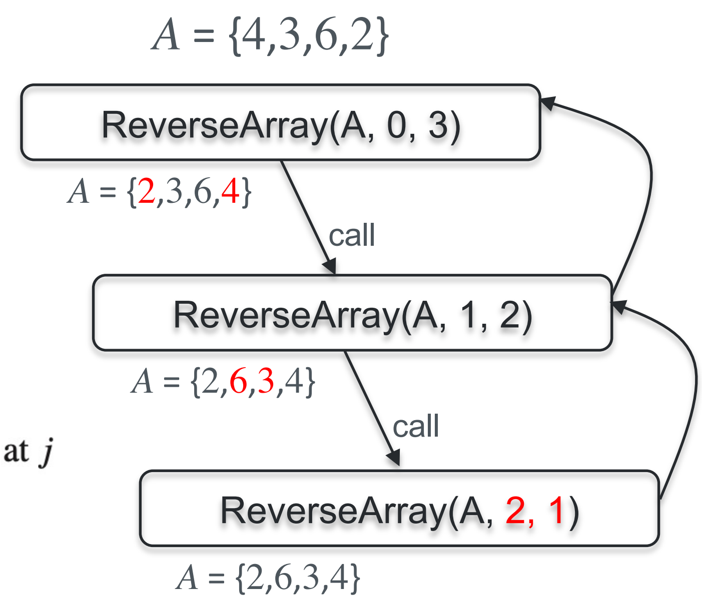
- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?
  - if **i = j** or **i > j**
    - in either of cases algorithm terminates
- Does algorithm terminate?



Algorithm ReverseArray(A, i, j):
  Input: An array A and nonnegative 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

Linear Recursion - Reversing an Array

- Reversing the n elements of an array
- Reversal: by swapping the first and last elements and then recursively reversing the remaining elements in the array.
  - Note how subproblems are defined! (initial call ReverseArray(A, 0, n-1)
    - other than array, we have two other index inputs
- Base cases?
  - if **i = j** or **i > j**
    - in either of cases algorithm terminates
- Does algorithm terminate?



Algorithm ReverseArray(A, i, j):
    Input: An array A and nonnegative 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

Tail Recursion

- Recursive calls are costly
  - Stack memory keeps track of the state of each active recursive call
- We can convert some recursive algorithms into non-recursive version
  - Suitable for imperative languages like C++
- There are two conditions:
  - The recursion is Linear
  - The recursive call is the last thing that function does

Algorithm ReverseArray(A, i, j):
  Input: An array A and nonnegative 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$(A, i+1, j-1)$ Recursive call is the last thing!
  return

Tail Recursion

- Recursive calls are costly
  - Stack memory keeps track of the state of each active recursive call
- We can convert some recursive algorithms into non-recursive version
  - Suitable for imperative languages like C++
- There are two conditions:
  - The recursion is Linear
  - The recursive call is the last thing that function does

Algorithm LinearSum(A, n):

Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements

Output: The sum of the first n integers in A

if $n = 1$ **then**
  return $A[0]$
else
  return LinearSum$(A, n-1) + A[n-1]$

Recursive call is NOT the last thing!

Tail Recursion

- There are two conditions:
  - The recursion is Linear
  - The recursive call is the last thing that function does
- **Iterate** through the recursive calls rather than calling them explicitly

- Linear Recursive version:

Algorithm ReverseArray(A, i, j):
  Input: An array A and nonnegative 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

- Tail recursion conversion:

  (iterative)

Algorithm IterativeReverseArray(A, i, j):
  Input: An array A and nonnegative integer indices i and j
  Output: The reversal of the elements in A starting at index i and ending at j
  while $i < j$ **do**
    Swap $A[i]$ and $A[j]$
    $i \leftarrow i+1$
    $j \leftarrow j-1$
  return

Binary Recursion

- When an algorithm makes **two** recursive calls
  - Useful to solve <u>two similar halves</u> of a problem
- Again! summing the **n** elements of an integer array A:
  - recursively
    - summing first half
    - summing second half
    - adding the two

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

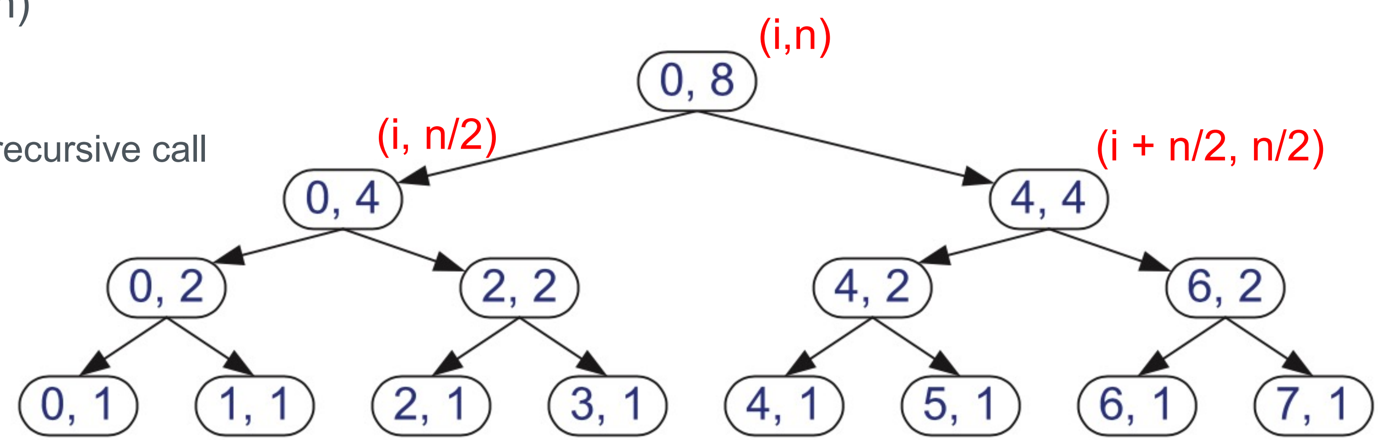
if $n = 1$ **then**
    return $A[i]$
return BinarySum$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

Binary Recursion

- Again! summing the **n** elements of an integer array **A**:
  - recursively
    - summing first half
    - summing second half
    - adding the two

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$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

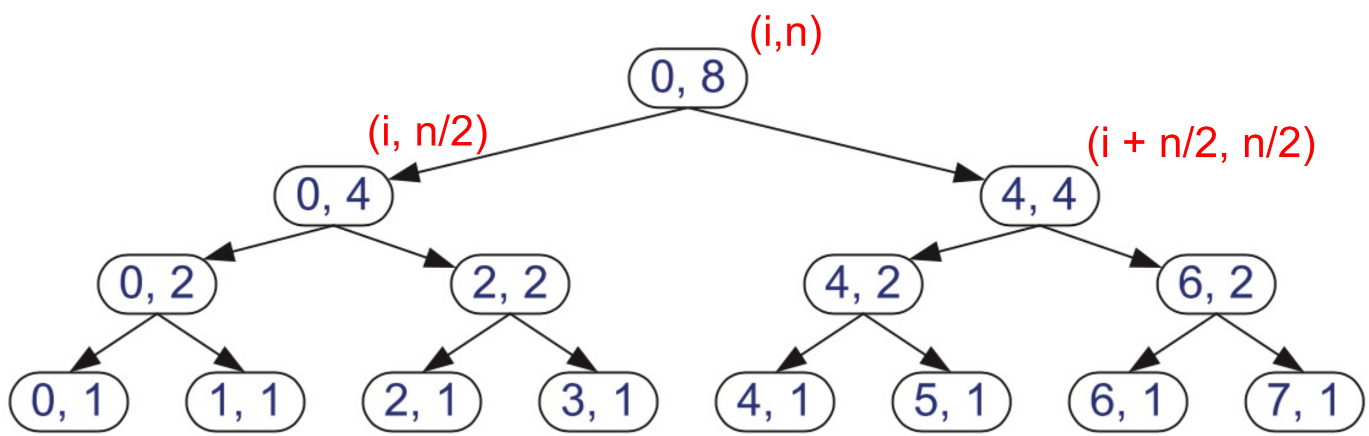


Binary Recursion

- Analysis of the algorithm:
  - We assume n is a power of 2
  - BinarySum(A, 0, n)
    - BinarySum(A, 0, 8)
  - *n* is halved at each recursive call

Algorithm $\text{BinarySum}(A, i, n)$:
  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 $\text{BinarySum}(A, i, \lceil n/2 \rceil) + \text{BinarySum}(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

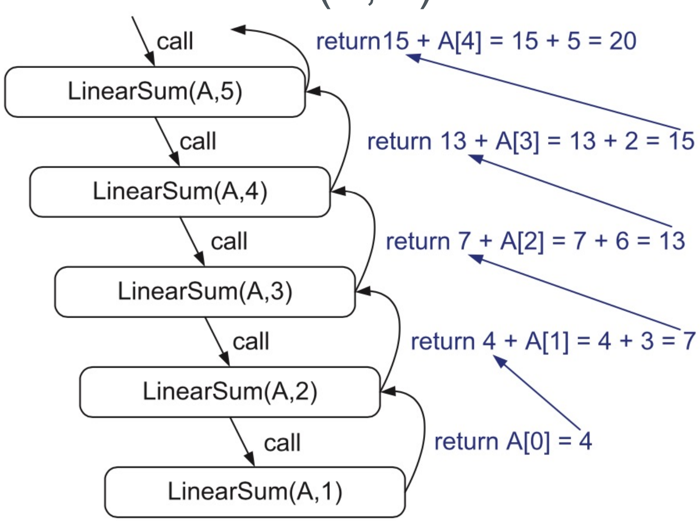
- The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is $\mathbf{1 + \log_2 \mathit{n}}$



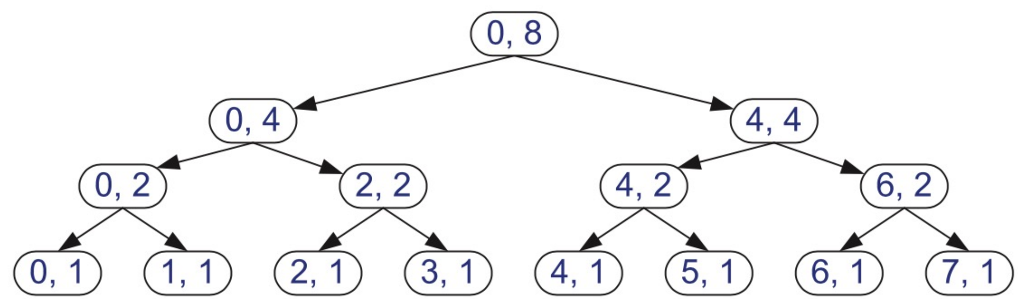
Binary Recursion

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Memory consumption** (Space complexity)
  - The depth of the recursion, that is, the maximum number of function instances that are active at the same time, is $\mathbf{1 + \log_2 \mathit{n}}$
  - Remember that this depth was **n** for **LinearSum**

LinearSum(A, 5)



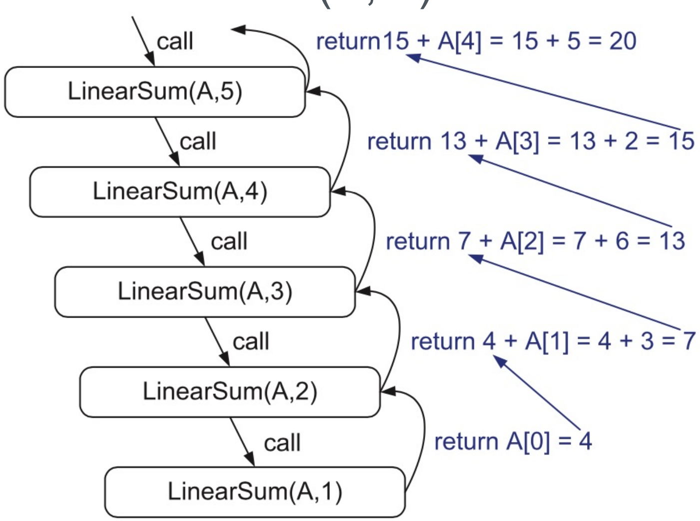
BinarySum(A, 0, 8)



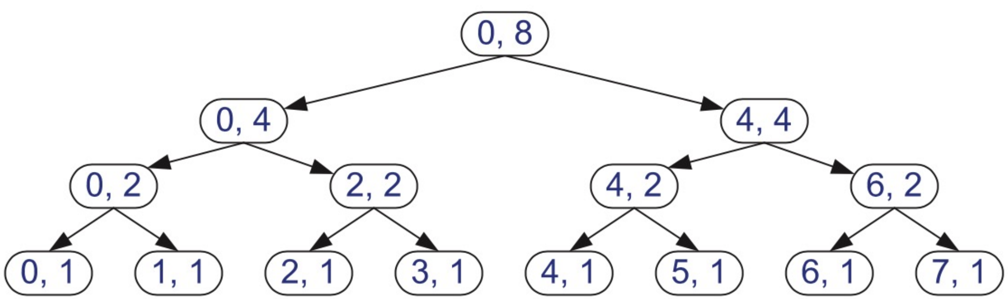
Binary Recursion

- Analysis of the algorithm:
  - We assume n is a power of 2
- **Runtime** (Time complexity)
  - There are **2n-1** boxes (calls) in **BinarySum**
  - There are **n** boxes (calls) in **LinearSum**
- Assume each calls is visited in a constant time

LinearSum(A, 5)



BinarySum(A, 0, 8)



Questions?