Data Structure:
Recursive Algorithms

2017

Instructor: Prof. Young-guk Ha
Dept. of Computer Science & Engineering

What is Recursion

- Recursion
  - When a function calls itself
- Classic example: "the factorial function"
  - $\boldsymbol{n}! = 1 \cdot 2 \cdot 3 \cdot \ \cdots \ \cdot (\boldsymbol{n}\text{-}1) \cdot \boldsymbol{n}$
    $= (\boldsymbol{n}\text{-}1)! \cdot \boldsymbol{n}$
  - Recursive definition of $f(\boldsymbol{n}) = \boldsymbol{n}!$

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

  - Factorial implementation in Java using recursion

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

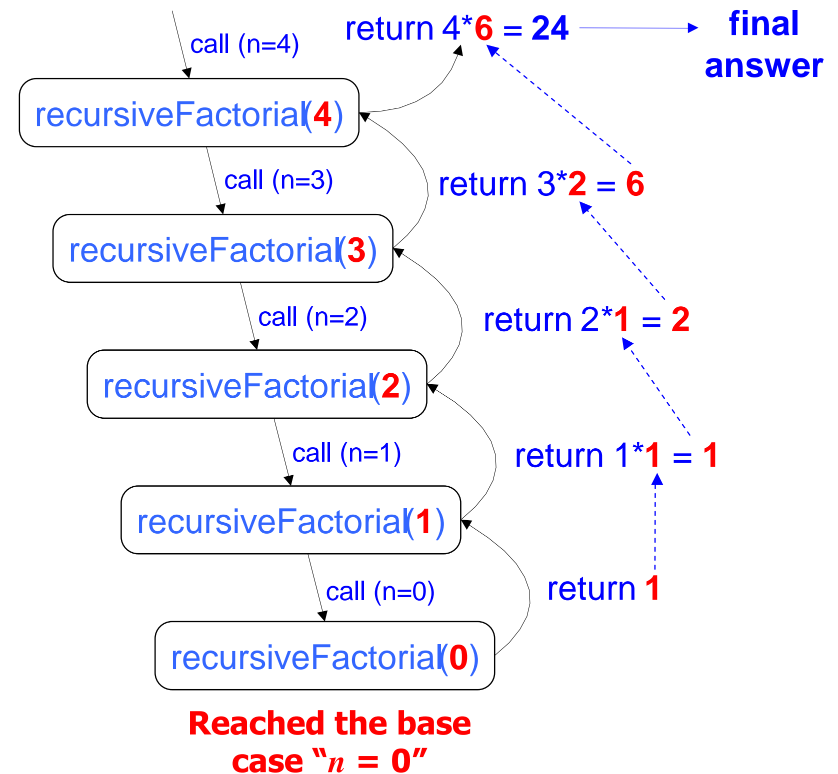
Content of Recursive Function

- ***Base cases***
  - Values of the input variables or specific conditions for which we perform no recursive calls
    - There ***must*** be one or more base cases
    - In the factorial function example: $\boldsymbol{n} = 0$
  - Handling of each base case ***should not*** use recursion

- ***Recursive calls***
  - Calls to the current function itself
    - There are one or more recursive calls
    - In the factorial function example: $f(\boldsymbol{n} - 1)$
  - Each recursive call must be defined so that it makes progress towards a base case
    - I.e., every possible chain of recursive calls eventually reach a base case
    - In the factorial function example: $\boldsymbol{n} \rightarrow (\boldsymbol{n} - 1) \rightarrow (\boldsymbol{n} - 2) \rightarrow (\boldsymbol{n} - 3) \rightarrow \ldots \rightarrow 0$

Visualizing Recursion

- Recursion trace for `recursiveFactorial(4)`
  - A rounded box represents each recursive call
  - Arrows from each caller to callee represent the chain of recursive calls
  - An arrow from each callee to caller shows a return value



Performance Analysis of Recursive Algorithms (1)

- Time complexity of **`recursiveFactorial`** algorithm
  - Let $\mathrm{T}(\boldsymbol{n})$ be the total count of function calls with given argument $\boldsymbol{n}$, we can get the following ***recurrence equation***

    $\mathrm{T}(0) = 1$

    $\mathrm{T}(\boldsymbol{n}) = \mathrm{T}(\boldsymbol{n} - 1) + 1 \quad \text{where } \boldsymbol{n} \geq 1$

  - Getting a ***closed-form equation*** of $\boldsymbol{n}$ from the recurrence equation

    $\mathrm{T}(\boldsymbol{n} - 1) = \mathrm{T}((\boldsymbol{n} - 1) - 1) + 1 = \mathrm{T}(\boldsymbol{n} - 2) + 1$

    $\mathrm{T}(\boldsymbol{n} - 2) = \mathrm{T}((\boldsymbol{n} - 2) - 1) + 1 = \mathrm{T}(\boldsymbol{n} - 3) + 1$

    $\mathrm{T}(\boldsymbol{n} - 3) = \quad \ldots$

    thus

    $\mathrm{T}(\boldsymbol{n}) = \mathrm{T}(\boldsymbol{n} - 1) + 1 = \{\mathrm{T}(\boldsymbol{n} - 2) + 1\} + 1 = \mathrm{T}(\boldsymbol{n} - 2) + 2$

    $= \mathrm{T}(\boldsymbol{n} - 3) + 3 = \ldots = \mathrm{T}(\boldsymbol{n} - \boldsymbol{k}) + \boldsymbol{k} = \ldots$ → *This iteration continues until it reaches the base case "$\boldsymbol{n} - \boldsymbol{k} = 0$", i.e., "$\boldsymbol{k} = \boldsymbol{n}$"*

    $= \mathrm{T}(0) + \boldsymbol{n} = 1 + \boldsymbol{n} = \boldsymbol{O}(\boldsymbol{n})$

Performance Analysis of Recursive Algorithms (2)

- Space complexity of **`recursiveFactorial`** algorithm
  - Let S($\boldsymbol{n}$) be the total number of function frames pushed into the system stack with given argument $\boldsymbol{n}$, we can get the following

    $S(0) = 1$

    $S(\boldsymbol{n}) = S(\boldsymbol{n} - 1) + 1$ where $\boldsymbol{n} \geq 1$ → *Same as the time complexity*

  - Getting a closed-form equation of $\boldsymbol{n}$ from the recurrence equation

    $S(\boldsymbol{n} - 1) = S((\boldsymbol{n} - 1) - 1) + 1 = S(\boldsymbol{n} - 2) + 1$

    $S(\boldsymbol{n} - 2) = S((\boldsymbol{n} - 2) - 1) + 1 = S(\boldsymbol{n} - 3) + 1$

    $S(\boldsymbol{n} - 3) = \ \ldots$

    thus

    $S(\boldsymbol{n}) = S(\boldsymbol{n} - 1) + 1 = S(\boldsymbol{n} - 2) + 1 + 1 = S(\boldsymbol{n} - 2) + 2$

    $= S(\boldsymbol{n} - 3) + 3 = \ldots = S(\boldsymbol{n} - \boldsymbol{k}) + \boldsymbol{k} = \ldots$

    $= S(0) + \boldsymbol{n} = 1 + \boldsymbol{n} = \boldsymbol{O}(\boldsymbol{n})$ → *Same as the time complexity*

Classification of Recursion

- Linear recursion
  - When an algorithm makes ***at most one*** recursive call each time it is invoked

    E.g.) **`recursiveFactorial`**

- Binary recursion
  - When an algorithm makes ***two or zero*** recursive calls each time it is invoked

- Multiple recursion
  - Generalization of recursion
  - When an algorithm could make ***multiple*** (potentially more than two) recursive calls

Linear Recursion

- Test for base cases
  - Begins by testing for at least one base case
  - Every possible chain of recursive calls must eventually reach a base case
  - And the handling of each base case should ***not*** use recursion
- Recur once
  - Performs a single recursive call
    - This recursive step may involve a set of tests that decides which of several possible recursive calls to make
    - Though, it should ultimately choose ***just one*** of these calls each time we perform this step
  - Each possible recursive call is defined so that it makes progress towards a base case

Another Simple Example of Linear Recursion

// Eq.: $\text{sum}(\boldsymbol{n}) = \text{sum}(\boldsymbol{n}-1) + \boldsymbol{n}$

Algorithm LinearSum(A, n):

Input: integer array A and integer n, such that A has at least n elements

Output: sum of the first n integers in A

if $n = 1$ **then** // a base case

  return $A[0]$

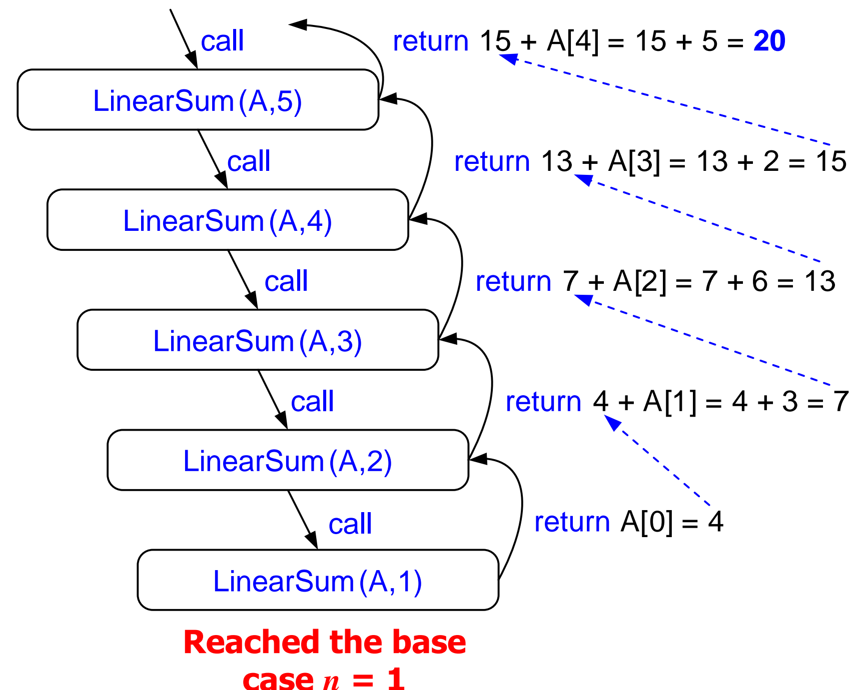
else

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

  // sum of first $\boldsymbol{n}$-1 ints of $\boldsymbol{A}$ + $\boldsymbol{n}$th element of $\boldsymbol{A}$

[Example]

Recursion trace of calling LinearSum(A = {4, 3, 6, 2, 5}, 5)



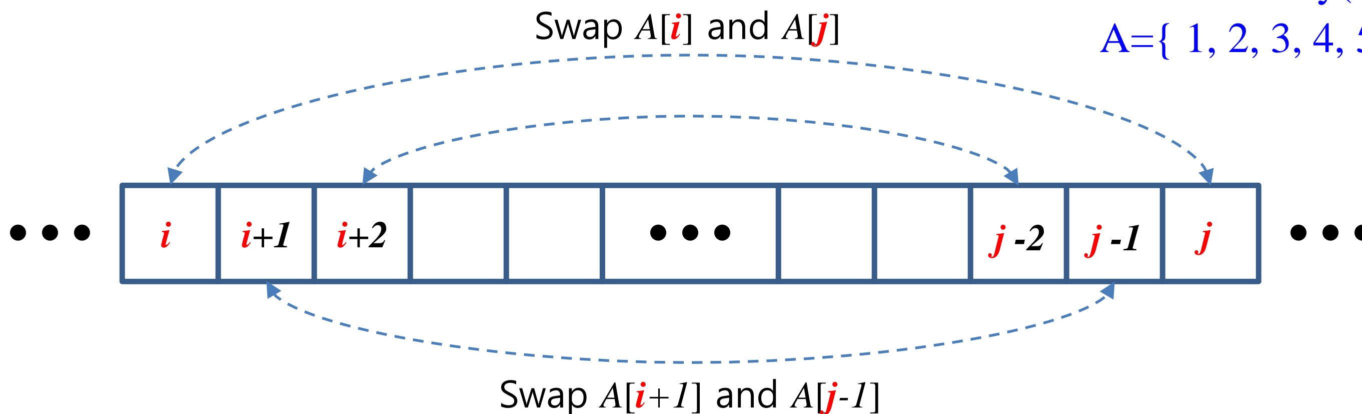
Definition of Arguments for Recursion

- In creating a recursive algorithm, it is important to define the algorithm in a way that facilitate recursion
- This sometimes requires us to define some additional arguments that are passed to the recursive algorithm
- For the next example of reversing array, we will define the array reversal algorithm as **ReverseArray(*A*, *i*, *j*)**, not **ReverseArray(*A*, *n*)**

Reversing an Array

```
Algorithm ReverseArray(A, i, j):
  Input: array A and nonnegative integer indices i and j
  Output: 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
```

E.g.) Reversing the order of array elements
ReverseArray(A, 0, 4)
A={ 1, 2, 3, 4, 5 } → { 5, 4, 3, 2, 1 }



❖ *Q: What is the base case of this recursion?*

Computing Powers

- The power function $p(\boldsymbol{x}, \boldsymbol{n}) = \boldsymbol{x}^{\boldsymbol{n}}$ can be defined recursively: $\boldsymbol{x}^{\boldsymbol{n}} = \boldsymbol{x} \cdot \boldsymbol{x}^{\boldsymbol{n}\text{-}1}$, i.e.,

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

- We intuitively know the power function runs in $\boldsymbol{O}(\boldsymbol{n})$ time because we make at most $\boldsymbol{n}$ recursive calls
  - ✓ [Intuition #1] if $\boldsymbol{n}$ decreases by 1 until the recursion reaches the base case $\boldsymbol{n} = \mathbf{c}$ (where $\mathbf{c}$ is a constant) → then the algorithm runs in $\boldsymbol{O}(\boldsymbol{n})$ time

- However, we can compute powers faster than the above, i.e., in $\boldsymbol{O}(\log \boldsymbol{n})$ time!

Computing Powers by Repeated Squaring

- We can derive a more efficient linearly recursive power algorithm by using ***repeated squaring***

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

- For example
  - $2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = (((2 \cdot 2) \cdot 2) \cdot 2)$ → multiply four 2s linearly
    $= (2 \cdot 2)^2$ → vs. (multiply two 2s)^2
  - $2^5 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = ((((2 \cdot 2) \cdot 2) \cdot 2) \cdot 2)$ → multiply five 2s linearly
    $= 2 \cdot (2 \cdot 2)^2$ → vs. (multiply two 2s)^2 * 2
  - …

Analyzing the Recursive Squaring Method

```
Algorithm Power(x, n):
  Input: number x and integer n
  Output: value of x^n
   if n = 0 then
       return 1
   if n mod 2 then     // n is odd?
       y  = Power(x, (n - 1) / 2)
       return x · y · y
   else
       y = Power(x, n / 2)
       return y · y
```

- ✓ [Intuition #2] Each time we make a recursive call, we divide n by a constant c
  → hence, we make $\log_c n$ recursive calls and this means the algorithm runs in $O(\log n)$ time

- ✓ It is important that we use the variable y twice here rather than calling the **Power** function twice

❖ *Q: Is this a linear recursion or binary recursion?*

Tail Recursion

- ***Tail recursion*** occurs when a linear recursion algorithm makes its recursive call as the ***last*** step
  - Such algorithms can be easily converted to a non-recursive version, which save some system resources (e.g., Java system stack)
    - Sufficient condition to convert to non-recursive version
  - The array reversal algorithm is an example

```
Algorithm IterativeReverseArray(A, i, j)
Input: array A and nonnegative integer indices i and j
Output: 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  = i + 1
        j  = j - 1
    return
```

❖ *Q: What about the* **`LinearSum`**. *A tail recursion or not?*

Binary Recursion

- Binary recursion occurs whenever there are two recursive calls for each non-base case
  - E.g., adding all the numbers in an integer array ***A*** using binary recursion

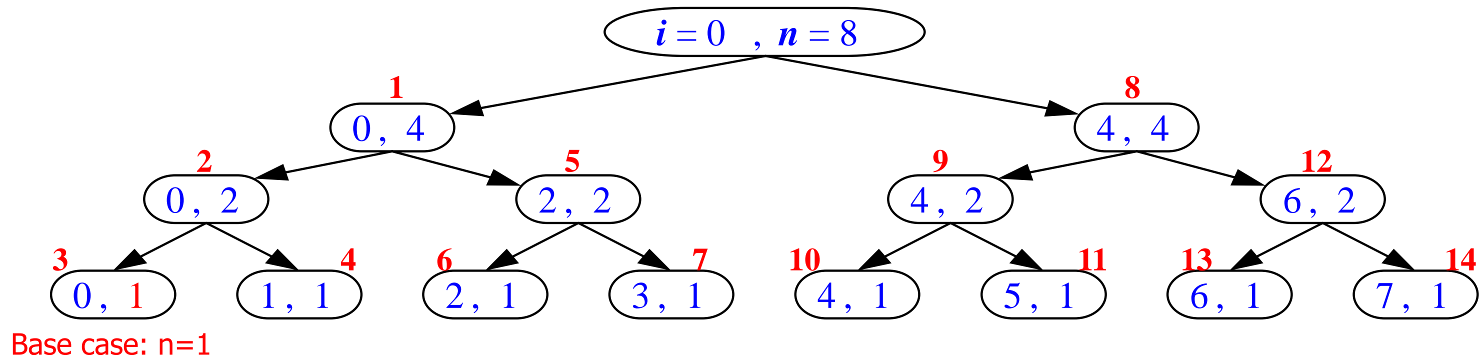
```
Algorithm BinarySum(A, i, n):
Input: array A and int i and n
Output: sum of n int in A starting at i
   if n = 1 then
      return A[i]
   else
      return BinarySum(A, i, ⌈n/2⌉) + BinarySum(A, i + ⌈n/2⌉, n/2)
```

E.g.) sum(1, 2, 3, 4, 5, 6, 7, 8)

→ sum(1, 2, 3, 4) + sum(5, 6, 7, 8)

→ { sum(1, 2) + sum(3, 4) } +
{ sum(5, 6) + sum(7, 8) }

→ …



[Visual trace of BinarySum(A, 0, 8)]

Time Complexity of BinarySum

- Let $T(\boldsymbol{n})$ be the total count of function calls with given argument $\boldsymbol{n}$, then we have

$$T(1) = 1 \qquad \text{where } \boldsymbol{n} = 1$$

$$T(\boldsymbol{n}) = T(\boldsymbol{n}/2) + T(\boldsymbol{n}/2) + 1 = 2T(\boldsymbol{n}/2) + 1 \qquad \text{where } \boldsymbol{n} > 1$$

and

$$T(\boldsymbol{n}/2) = 2T((\boldsymbol{n}/2)/2) + 1 = 2T(\boldsymbol{n}/4) + 1$$

$$T(\boldsymbol{n}/4) = 2T((\boldsymbol{n}/4)/2) + 1 = 2T(\boldsymbol{n}/8) + 1$$

$$T(\boldsymbol{n}/8) = \ \ldots$$

thus

$$T(\boldsymbol{n}) = 2T(\boldsymbol{n}/2) + 1 = 2\{2T(\boldsymbol{n}/4) + 1\} + 1 = 4T(\boldsymbol{n}/4) + 2 + 1$$

$$= 8T(\boldsymbol{n}/8) + 6 + 1 = \ldots = \boldsymbol{k}T(\boldsymbol{n}/\boldsymbol{k}) + 3\boldsymbol{k}/4 + 1 = \ldots \qquad \rightarrow \text{until } \boldsymbol{k} = \boldsymbol{n}$$

$$= \boldsymbol{n}T(1) + 3\boldsymbol{n}/4 + 1 = \boldsymbol{n} + 3\boldsymbol{n}/4 + 1 = 7/4 * \boldsymbol{n} + 1 = \boldsymbol{O}(\boldsymbol{n})$$

❖ *Q: It runs in $\boldsymbol{O}(\log \boldsymbol{n})$ time, not $\boldsymbol{O}(\log \boldsymbol{n})$. Why?*

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively

$$F_0 = 0,\ F_1 = 1 \qquad \text{where } \boldsymbol{n} \le 1$$
$$F_{\boldsymbol{n}} = F_{\boldsymbol{n}-1} + F_{\boldsymbol{n}-2} \qquad \text{where } \boldsymbol{n} > 1$$

E.g.) 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

- Fibonacci numbers as a binary recursive algorithm

```
Algorithm BinaryFib(n)
Input: nonnegative integer n
Output: nth Fibonacci number Fn
    if n ≤ 1 then return n
    else return BinaryFib(n - 1) + BinaryFib(n - 2)
```

Intuitive Analysis of the Binary Recursive Fibonacci Algorithm

- Let $\mathrm{T}(\boldsymbol{n})$ denote number of recursive calls made by BinaryFib($\boldsymbol{n}$), then
  - T(0) = **1**
  - T(1) = **1**
  - T(2) = T(1) + T(0) + 1 = 1 + 1 + 1 = **3**
  - T(3) = T(2) + T(1) + 1 = 3 + 1 + 1 = **5**
  - T(4) = T(3) + T(2) + 1 = 5 + 3 + 1 = **9**
  - T(5) = T(4) + T(3) + 1 = 9 + 5 + 1 = **15**
  - T(6) = T(5) + T(4) + 1 = 15 + 9 + 1 = **25**
  - T(7) = T(6) + T(5) + 1 = 25 + 15 + 1 = **41**
  - T(8) = T(7) + T(6) + 1 = 41 + 25 + 1 = **67**
  - …

- We can observe that the value of $\mathrm{T}(\boldsymbol{n})$ grows almost double, more precisely, $2^{\boldsymbol{n}} > \mathrm{T}(\boldsymbol{n}) > 2^{\boldsymbol{n}/2}$, thus $\mathrm{T}(\boldsymbol{n}) = O(2^{\boldsymbol{n}})$
  → It has exponential complexity!

Time Complexity of BinaryFib

- Let $\mathrm{T}(\boldsymbol{n})$ denote the total number of recursive calls made by BinaryFib($\boldsymbol{n}$)

$$\mathrm{T}(0) = 1,\ \mathrm{T}(1) = 1 \qquad \text{where } \boldsymbol{n} \leq 1$$

$$\mathrm{T}(\boldsymbol{n}) = \mathrm{T}(\boldsymbol{n}-1) + \mathrm{T}(\boldsymbol{n}-2) + 1 \qquad \text{where } \boldsymbol{n} > 1$$

- Analysis of the upper bound of $\mathrm{T}(\boldsymbol{n})$

$$\begin{aligned}
\mathrm{T}(\boldsymbol{n}) &= \mathrm{T}(\boldsymbol{n}-1) + \mathrm{T}(\boldsymbol{n}-2) + 1 \\
&\leq \mathrm{T}(\boldsymbol{n}-1) + \mathrm{T}(\boldsymbol{n}-1) + 1 \\
&= 2\mathrm{T}(\boldsymbol{n}-1) + 1 \leq 2\{2\mathrm{T}(\boldsymbol{n}-2) + 1\} + 1 \\
&= 2^2\mathrm{T}(\boldsymbol{n}-2) + 2 + 1 \leq \ldots \\
&= 2^{\boldsymbol{k}}\mathrm{T}(\boldsymbol{n}-\boldsymbol{k}) + 2^{\boldsymbol{k}-1} + 2^{\boldsymbol{k}-2} + \ldots + 2 + 1 \leq \ldots \\
&= 2^{\boldsymbol{n}}\mathrm{T}(0) + 2^{\boldsymbol{n}-1} + 2^{\boldsymbol{n}-2} + \ldots + 2 + 1 \\
&= 2^{\boldsymbol{n}} + \underline{2^{\boldsymbol{n}-1} + 2^{\boldsymbol{n}-2} + \ldots + 2 + 1} = 2^{\boldsymbol{n}+1} - 1
\end{aligned}$$

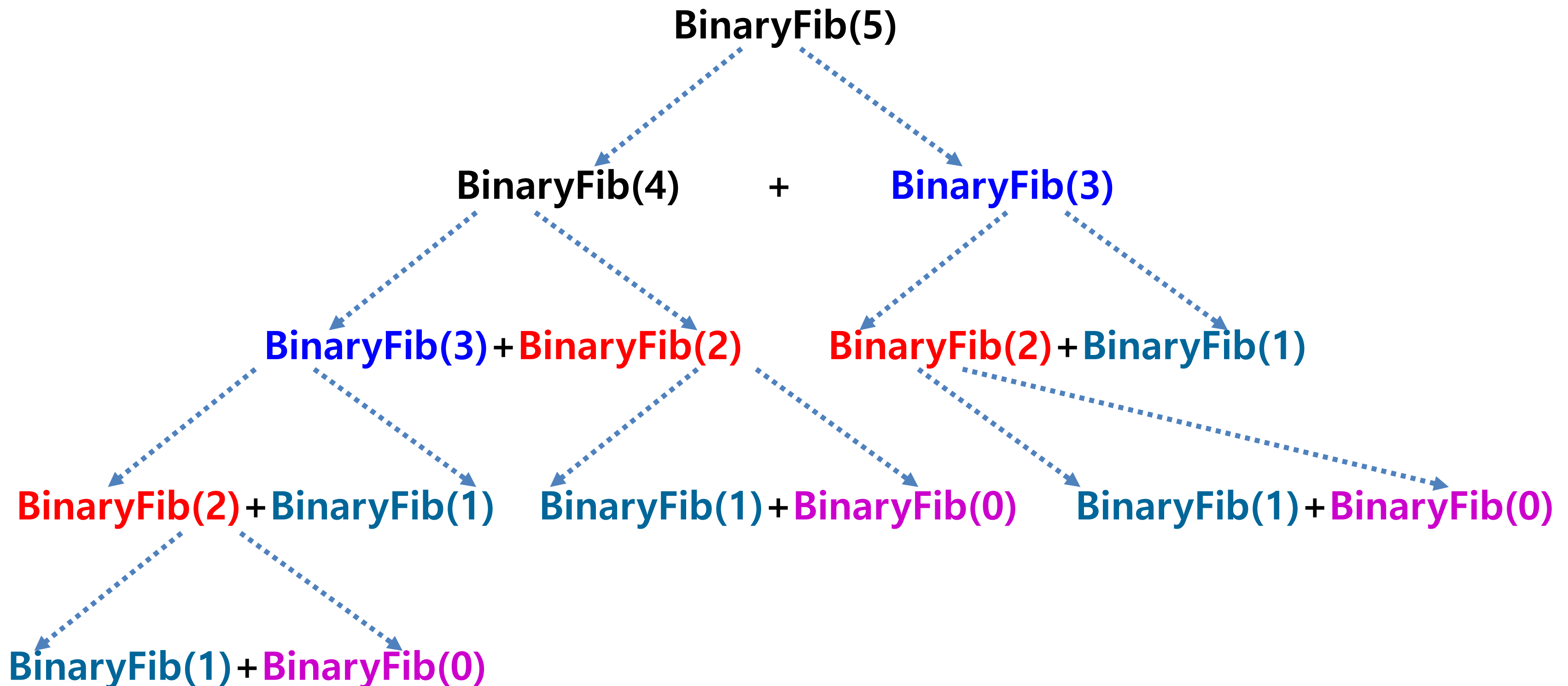
→ $T(n-1) = T(n-2) + T(n-3) + 1 > T(n-2)$

→ *This iteration continues until it reaches the base case "$\boldsymbol{n} - \boldsymbol{k} = 0$", i.e., "$\boldsymbol{k} = \boldsymbol{n}$"*

→ *According to the **Geometric Sum** proposition* $\underline{\sum_{0 \leq i \leq n-1} 2^i = 2^n - 1}$

I.e., $\mathrm{T}(\boldsymbol{n}) \leq 2^{\boldsymbol{n}+1} - 1$, $\mathrm{T}(\boldsymbol{n}) = O(2^{\boldsymbol{n}})$

Side Effect of Overlapping Calls



- ✓ Lessons learned: we should first try to **fully partition** the problem in two when using binary recursion
- ✓ Otherwise, we suffer from overlapping recursive calls (e.g., `BinaryFib` algorithm → runs in exponential time)

A Better Fibonacci Algorithm (1)

- Use linear recursion instead

```
Algorithm LinearFib(n):
    Input: nonnegative integer n
    Output: a pair of Fibonacci numbers (F_n, F_n-1)
      if n ≤ 1 then
          return (n, 0)
      else
          (i, j) = LinearFib(n - 1)
          return (i + j, i)      // i = F_n-1, j = F_n-2
```

- This algorithm runs in $O(n)$ time!
  - I.e., n decreases by 1 until the recursion reaches the base case $n \leq 1$

A Better Fibonacci Algorithm (2)

- Use simple iterative algorithm (non-recursive)

```
Algorithm LinearFib2(n):
    Input: nonnegative integer n
    Output: nth Fibonacci number F_n
      if n ≤ 1 then return (n)
      prev = 0, curr = 1
      for 2 ≤ i ≤ n
        next = curr + prev
        prev = curr, curr = next
      return next
```

- **LinearFib2** also runs in $O(n)$ time!

Multiple Recursion

- Multiple recursion makes potentially **multiple** (more than two) recursive calls
- E.g., solving a "letter combinatorial puzzle"

```
Algorithm PuzzleSolve(k, S, U):
Input: integer k, sequence S, and set U (the universe of letters to test)
Output: an enumeration of all k-length letters to S using letters in U
        without repetitions
 for all e in U do
   Remove e from U         // letter e is now being used
   Add e to the end of S
   if k = 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then return "Solution found: " S
   else
        PuzzleSolve(k - 1, S, U)     // called max |U| times in the loop
   Add e back to U         // letter e is now unused
   Remove e from the end of S
```

Visualizing PuzzleSolve Algorithm