Data Structures & Programming

Recursion

Golnar Sheikhshab

Factorial function

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

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

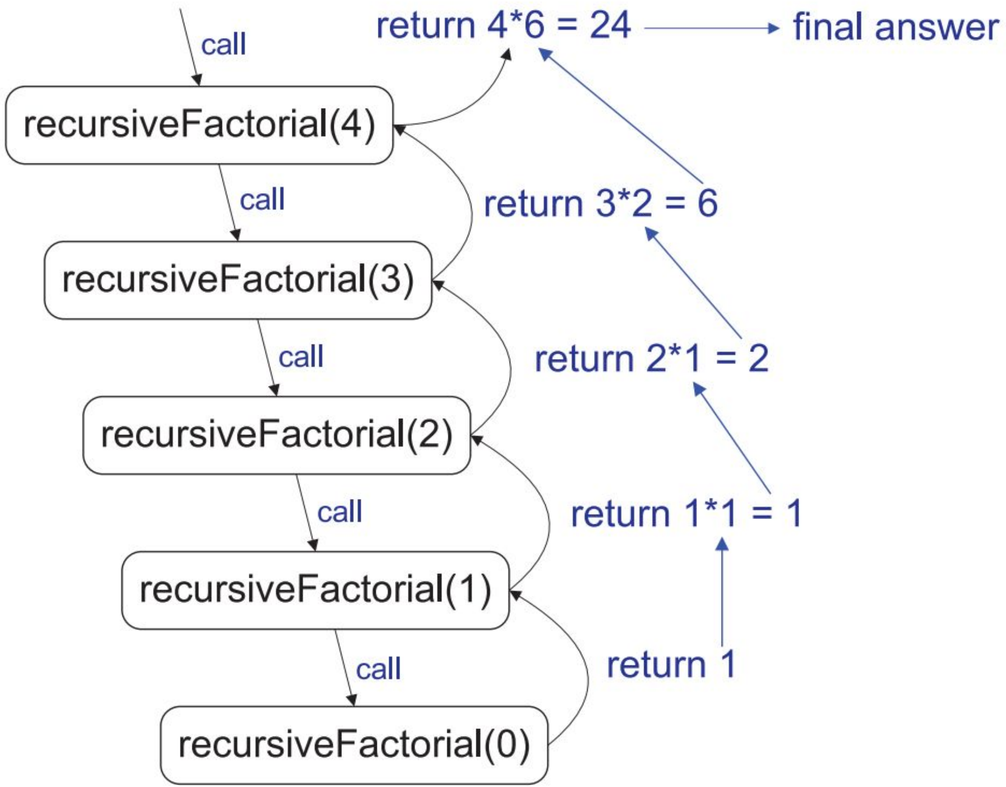
$$\mathsf{factorial}(5) = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot \mathsf{factorial}(4).$$

Recursive definition

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

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

A recursion trace



Summing over an array - a 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]
```

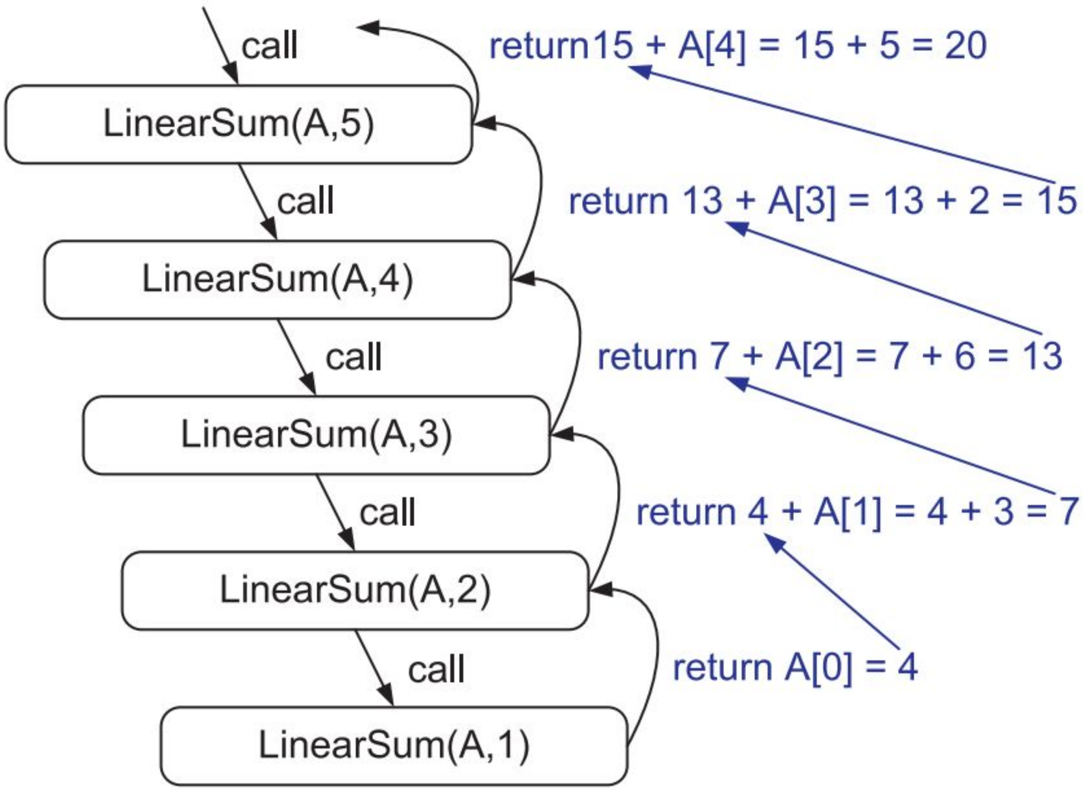


Figure 3.19: Recursion trace for an execution of LinearSum(A,n) with input parameters $A = \{4,3,6,2,5\}$ and $n = 5$.

Summing over an array - a binary recursion

Algorithm $\mathsf{BinarySum}(A, i, n)$:

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

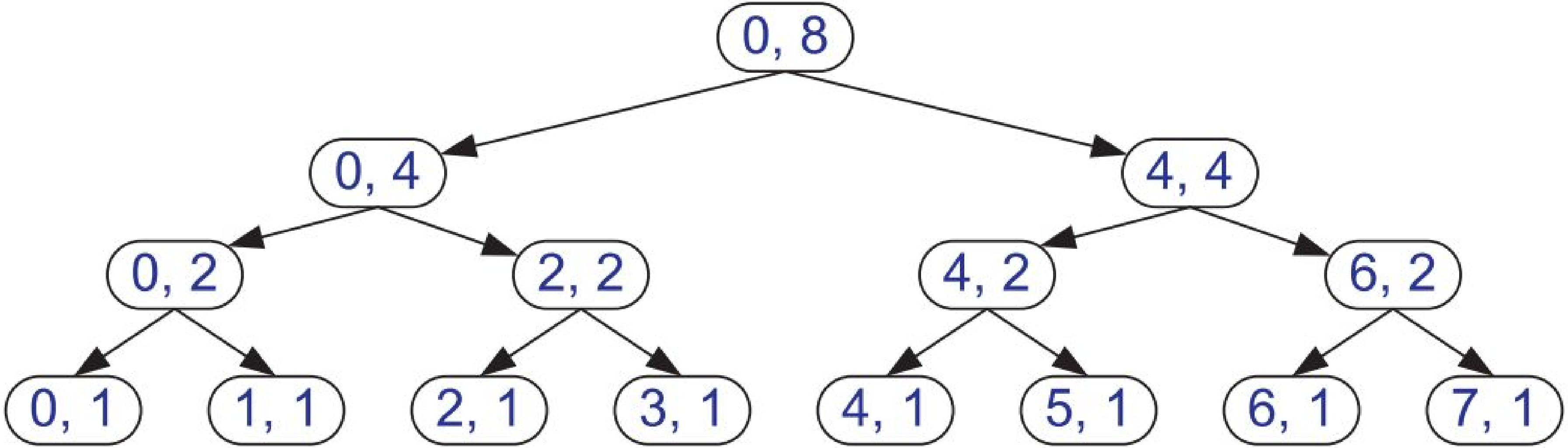


Figure 3.20: Recursion trace for the execution of BinarySum(0,8).

Recursion

Solve the problem using the solution to smaller problems of exact same structure

Steps:

1. Find a high level idea on
   a. what the useful subproblems are
   b. how we can combine their solutions to solve the bigger problem at hand
2. Find the structure the problem and subproblems have in common
   a. we may need to redefine the problem at hand to do this
3. Take care of the base cases

Recall the binary search from ArrayLinkedList class

We redefined the problem: BinarySearch(A, 0, n-1) instead of BinarySearch(A)

```
bool binarySearch(const E& elem, int start_index, int end_index){
        if (start_index > end_index)    // taking care of the base case
            return false;
        int middle_index = (start_index + end_index)/2;
        if (this->listArray_[middle_index]>elem)
            return binarySearch(elem, start_index, middle_index-1);
        else if (this->listArray_[middle_index]<elem)
            return binarySearch(elem, middle_index+1, end_index);
        else // this->listArray_[middle_index]==elem
            return true;
    }
```

Reverse an array recursively

- Idea:
  - swap the first and last item
  - reverse what remains in the middle recursively

- The exact common structure:
  - Reverse(A, i, j)
  - Redefining the problem from Reverse(A) to Reverse(A, 0, n)

- Base case
  - when i > j

Reverse an array - pseudo code

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

```
void ReverseArray(int* A, int i, int j){
	if (i<j){
		int temp = A[i]; A[i]= A[j-1]; A[j-1]=temp;
		ReverseArray(A, i+1, j-1);
	}
}
```

Printing all subsets of {1 ... n}

- idea:
    - the subsets either
        - contain n
        - don't contain n
    - subproblem
        - subsets of {1 ... n-1}
    - combining
        - get a copy of subsets of {1 ... n-1}
        - get another copy of the subsets and and n to each subset
        - get the union of the above

- common structure
    - vector<set<int>> getSubsetsOfOneToN(int n)

- base case
    - n = 0 (or 1)

Printing all subsets of {1 ... n} - implementation

```
vector<set<int>> getSubsetsOfOneToN(int n){
        vector<set<int>> v;    set<int> s;
        if (n<=0){
                v.push_back(s);
                return v;
        }
        else{
                v = getSubsetsOfOneToN(n-1); // all subsets without n
                int freezed_size = v.size();
                for (int i=0; i<freezed_size; i++){
                        s = v[i];    s.insert(n);    v.push_back(s); }
                return v;
        }
}
```

Computing Fibonacci numbers

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_i &= F_{i-1} + F_{i-2} \quad \text{for } i > 1
\end{aligned}
$$

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The kth Fibonacci number F_k

if $k \leq 1$ **then**

  return k

else

  return BinaryFib($k-1$) + BinaryFib($k-2$)

It solves the same subproblems again and again

If you call BinaryFib(5) how many times would BinaryFib(0) be called? How about BinaryFib(1)? BinaryFib(2)? ...

How many function calls would there be in total?

$$
\begin{aligned}
n_0 &= 1 \\
n_1 &= 1 \\
n_2 &= n_1 + n_0 + 1 = 1 + 1 + 1 = 3 \\
n_3 &= n_2 + n_1 + 1 = 3 + 1 + 1 = 5 \\
n_4 &= n_3 + n_2 + 1 = 5 + 3 + 1 = 9 \\
n_5 &= n_4 + n_3 + 1 = 9 + 5 + 1 = 15 \\
n_6 &= n_5 + n_4 + 1 = 15 + 9 + 1 = 25 \\
n_7 &= n_6 + n_5 + 1 = 25 + 15 + 1 = 41 \\
n_8 &= n_7 + n_6 + 1 = 41 + 25 + 1 = 67
\end{aligned}
$$

Efficient recursive Fibonacci number generation

Algorithm LinearFibonacci(k):
 Input: A nonnegative 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)
```

Efficient recursion

Memoization

Dynamic programming

More on this later in the semester ...

Recursion & memory

- Stack memory is used to keep the state of the active function call and the history
- Potentially, we can use up this memory (stack overflow)
- Theoretically, all recursive functions can be changed to non-recursive ones
    - using stack data structure that we'll see in near future
- Some recursive functions are easier to transform
    - tail recursion (when the last statement is the recursive call (and only the recursive call))

```
void RecursiveReverseA(int* A, int i, int j){
	if (i<j){
		swap (A[i], A[j-1]);
		ReverseArray(A, i+1, j-1);
	}
}
```

→

```
void IterativeReverseA(int* A, int i, int j){
	while (i<j){
		swap (A[i], A[j-1]);
		i = i+1;   j=j-1;
	}
}
```

Multiple recursion

- We can call arbitrarily many subproblems
- Most important examples of this
  - Breadth first search (We'll see this in chapter 13)
  - Brute force search (checking all configurations to find the best)
    - more on this soon ...

Reading material

Recursion from chapter 3