Data Structures I (CPCS-204)

Week # 5: Recursion

Recursion: Basic idea

- We have a bigger problem whose solution is difficult to find
- We divide/decompose the problem into smaller (sub) problems
  - Keep on decomposing until we reach to the smallest sub-problem (base case) for which a solution is known or easy to find
  - Then go back in reverse order and build upon the solutions of the sub-problems
- Recursion is applied when the solution of a problem depends on the solutions to smaller instances of the same problem

Recursive Function

- A function which calls itself

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

Finding a recursive solution

- Each successive recursive call should bring you closer to a situation in which the answer is known (cf. n-1 in the previous slide)
- A case for which the answer is known (and can be expressed without recursion) is called a base case
- Each recursive algorithm must have at least one base case, as well as the general recursive case

Recursion vs. Iteration: Computing N!

- The factorial of a positive integer n, denoted $n!$, is defined as the product of the integers from 1 to n. For example, $4! = 4\cdot3\cdot2\cdot1 = 24$.
  - Iterative Solution

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

  - Recursive Solution

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

Recursion: Do we really need it?

- In some programming languages recursion is imperative
  - For example, in declarative/logic languages (LISP, Prolog etc.)
  - Variables can’t be updated more than once, so no looping – (think, why no looping?)
  - Heavy backtracking

Recursion in Action: *factorial(n)*

```
factorial (5) = 5 x factorial (4)
              = 5 x (4 x factorial (3))
              = 5 x (4 x (3 x factorial (2)))
              = 5 x (4 x (3 x (2 x factorial (1))))
              = 5 x (4 x (3 x (2 x (1 x factorial (0)))))
              = 5 x (4 x (3 x (2 x (1 x 1))))
              = 5 x (4 x (3 x (2 x 1)))
              = 5 x (4 x (3 x 2))
              = 5 x (4 x 6)
              = 5 x 24
              = 120
```

Base case arrived
Some concept from elementary maths: Solve the inner-most bracket, first, and then go outward

How to write a recursive function?

- Determine the <u>size factor </u>(e.g. *n* in *factorial(n)*)
- Determine the <u>base case(s)</u>
  - the one for which you know the answer (e.g. 0! = 1)
- Determine the <u>general case(s</u>)
  - the one where the problem is expressed as a smaller version of itself (must converge to base case)
- Verify the algorithm
  - use the "Three-Question-Method" – next slide

Three-Question Verification Method

1. **The Base-Case Question**

   Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case? (cf. `if (n == 0) return 1`)

2. **The Smaller-Caller Question**

   Does each recursive call to the function involve a smaller case of the original problem, leading towards the base case? (cf. `factorial(n-1)`)

- **The General-Case Question**

  Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Linear Recursion

- The simplest form of recursion is ***linear recursion***, where a method is defined so that it makes at most one recursive call each time it is invoked
- This type of recursion is useful when we view an algorithmic problem in terms of a first or last element plus a remaining set that has the same structure as the original set

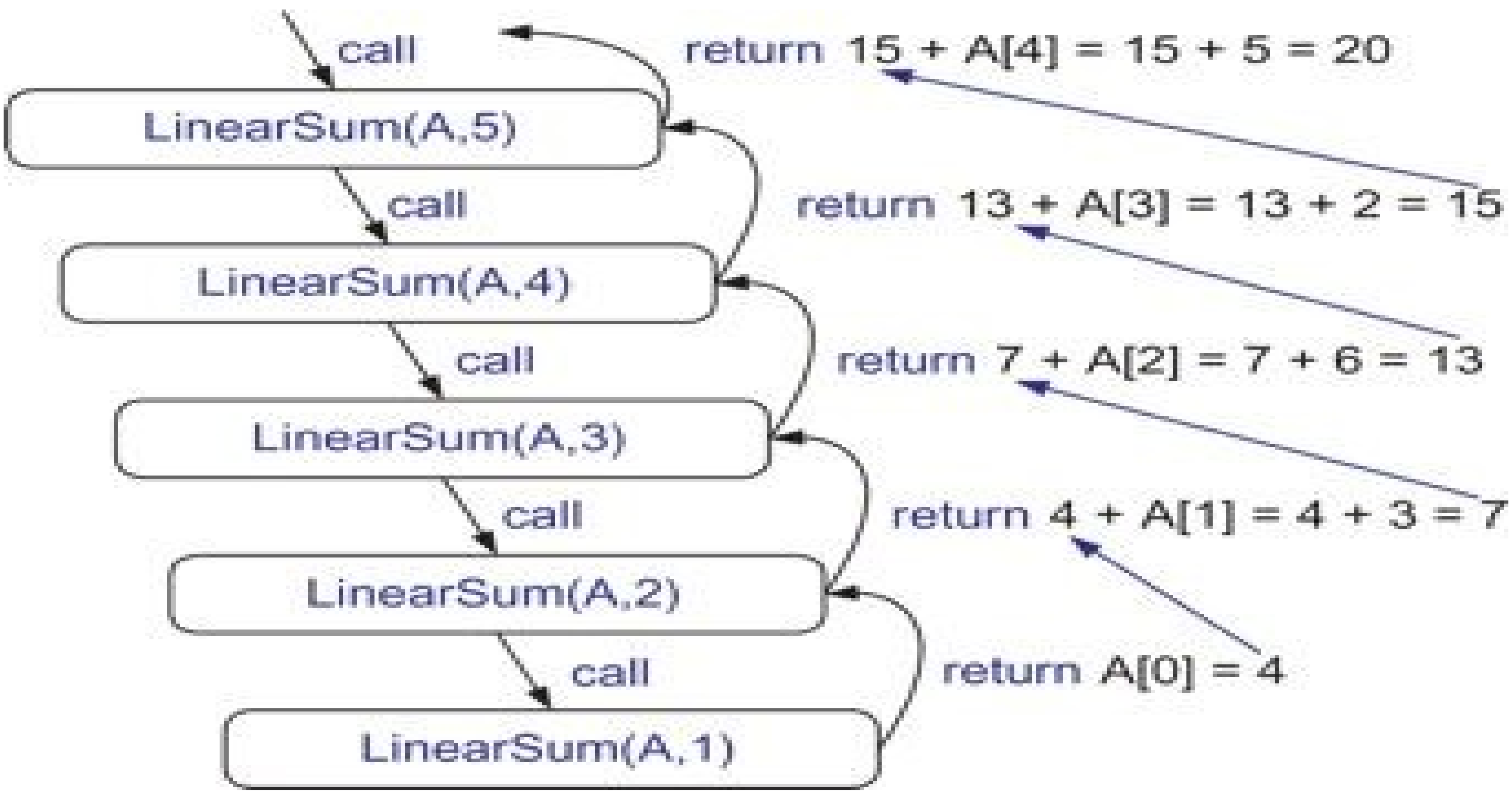
Summing the Elements of an Array

- We can solve this summation problem using linear recursion by observing that the sum of all n integers in an array A is:
  - Equal to $A[0]$, if $n = 1$, or
  - The sum of the first $n - 1$ integers in A plus the last element

```
int LinearSum(int A[], n){
    if n = 1 then
        return A[0];
    else
        return A[n-1] + LinearSum(A, n-1)
}
```

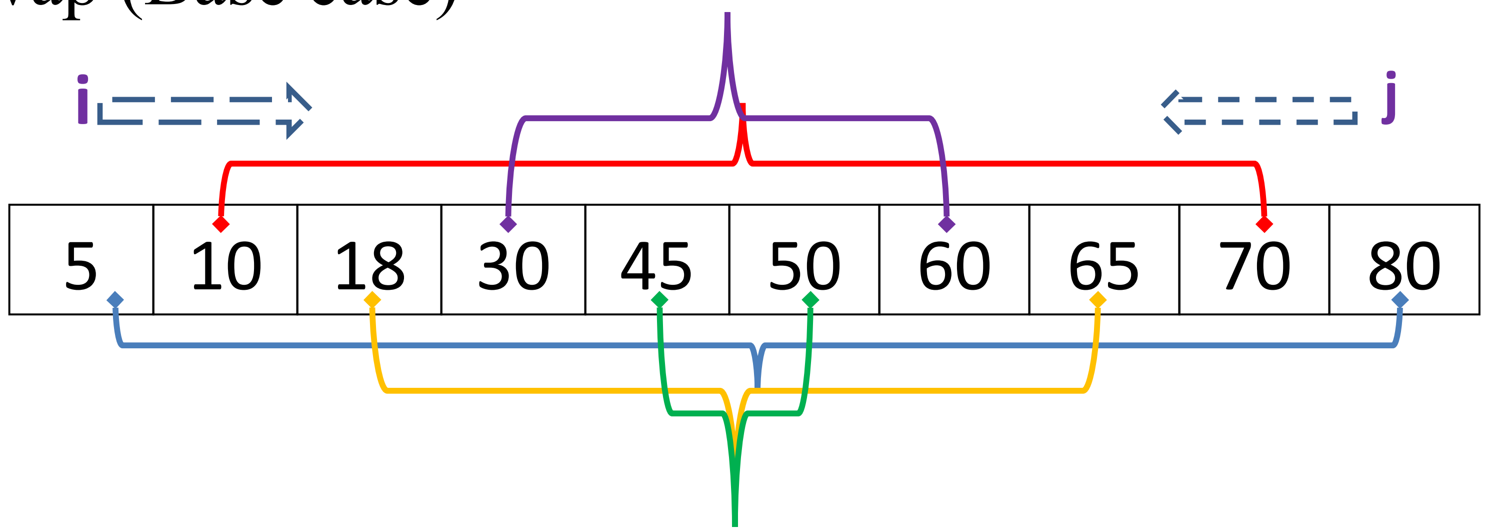
Analyzing Recursive Algorithms using Recursion Traces

- Recursion trace for an execution of *LinearSum(A,n)* with input parameters $A = [4,3,6,2,5]$ and $n = 5$



Linear recursion: Reversing an Array

- Swap 1st and last elements, 2nd and second to last, 3rd and third to last, and so on
- If an array contains only one element no need to swap (Base case)



- Update i and j in such a way that they converge to the base case (i = j)

Linear recursion: Reversing an Array

```
void reverseArray(int A[], i, j){
    if (i < j){
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
        reverseArray(A, i+1, j-1)
    }
    // in base case, do nothing
}
```

Linear recursion: run-time analysis

- Time complexity of linear recursion is proportional to the problem size
  - Normally, it is equal to the number of times the function calls itself
- In terms of Big-O notation time complexity of a linear recursive function/algorithm is $O(n)$

Recursion and stack management

- A quick overview of stack
  - Last in first out (LIFO) data structure
  - Push operation adds new element at the top
  - Pop operation removes the top element

What happens when a function is called?

- The rest of the execution in "caller" is suspended
- An activation record is created on stack, containing
  - Return address (in the caller code)
  - Current (suspended) status of the caller
- Control is transferred to the "called" function
- The called function is executed
- Once the called function finishes its execution, the activation record is popped of, and the suspended activity resumes

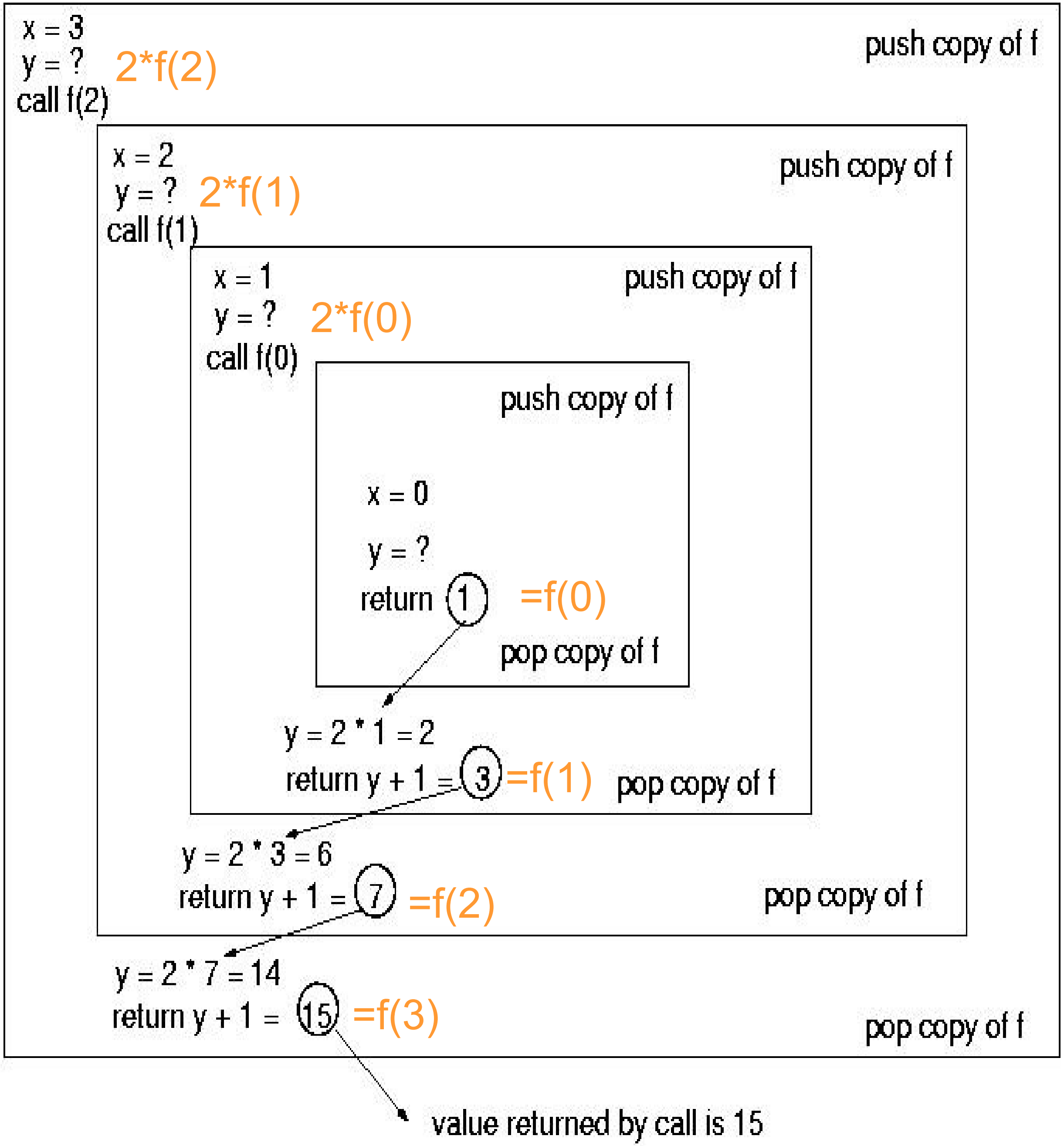
```
int a(int w)
{
 return w+w;
}

int b(int x)
{
 int z,y;
 z = a(x) + y;
 return z;
}
```

What happens when a recursive function is called?

- Except the fact that the calling and called functions have the same name, there is really no difference between recursive and non-recursive calls

```
int f(int x){
{
   int y;
   if(x==0)
      return 1;
   else{
     y = 2 * f(x-1);
     return y+1;
   }
}
```



Recursion: Run-time stack tracing

Let the function is called with parameter value 3, i.e. f(3)

```
int f(int x){
{
   int y;
   if(x==0)
      return 1;
   else{
      y = 2 * f(x-1);
      return y+1;
   }
}
```

Recursion and stack management

- A quick overview of stack
  - Last in first out (LIFO) data structure
  - Push operation adds new element at the top
  - Pop operation removes the top element

Binary recursion

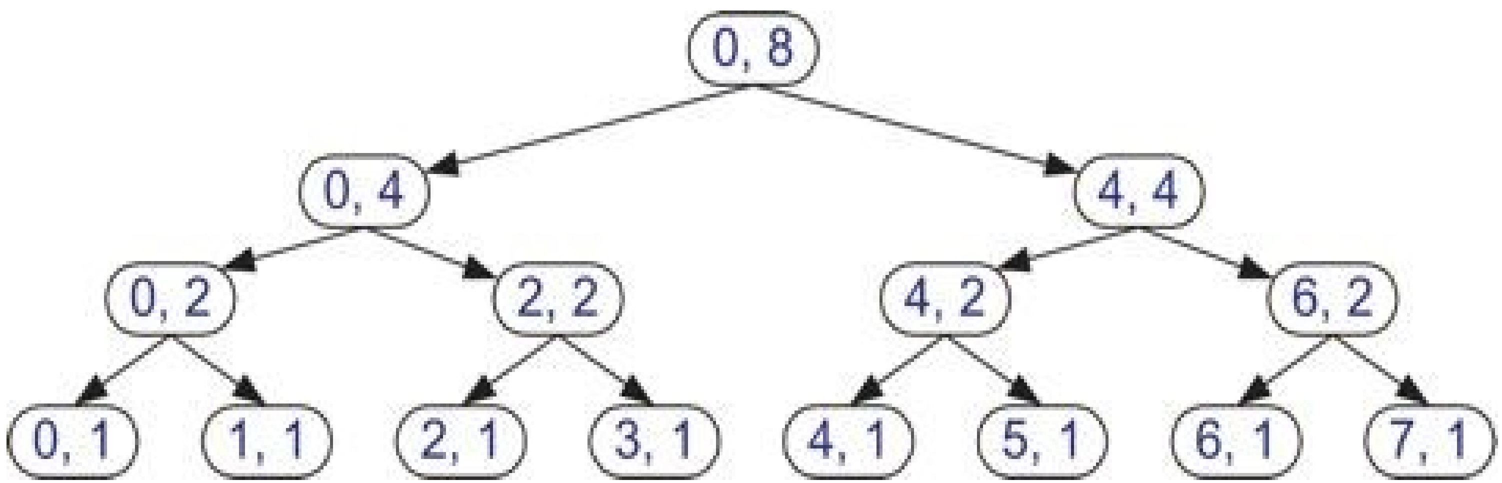
- Binary recursion occurs whenever there are **two** recursive calls for each non-base case
- These two calls can, for example, be used to solve two similar halves of some problem
- For example, the LinearSum program can be modified as:
    - recursively summing the elements in the first half of the Array
    - recursively summing the elements in the second half of the Array
    - adding these two sums/values together

Binary Recursion: Array Sum

- A is an array, i is initialised as 0, and n is initialised as array size

```
int BinarySum(int A[], int i, int n){
    if (n == 1)then                       // base case
        return A[i];
    else                                  // recursive case I
        return BinarySum(A, i, n/2) + BinarySum(A, i+n/2, n/2);
}
```

- Recursion trace for BinarySum, for n = 8   [Solve step-by-step]



Binary Search using Binary Recursion

- A is an array, key is the element to be found, LI is initialised as 0, and HI is initialised as array size - 1

```
int BinarySearch(int key, int A[], int LI, int HI){
    if (LI > HI)then                    // key does not exist
         return -1;
    if (key == A[mid])                  // base case
        return mid;
    else if (key < A[mid])              // recursive case I
            BinarySearch(key, A, LI, mid - 1);
         else                           // recursive case II
            BinarySearch(key, A, mid + 1, HI);
}
```

Tail Recursion

- An algorithm uses tail recursion if it uses linear recursion and the algorithm makes a recursive call as its very last operation
- For instance, our reverseArray algorithm is an example of tail recursion
- Tail recursion can easily be replaced by iterative code
  - Embed the recursive code in a loop
  - Remove the recursive call statement

Efficiency of recursion

- Recursion is not efficient because:
  - It may involve much more operations than necessary (Time complexity)
  - It uses the run-time stack, which involves pushing and popping a lot of data in and out of the stack, some of it may be unnecessary (Time and Space complexity)
- Both the time and space complexities of recursive functions may be considerably higher than their iterative alternatives

Recursion: general remarks

- Use recursion when:
  - The depth of recursive calls is relatively "shallow" compared to the size of the problem. (factorial is deep)
  - The recursive version does about the same amount of work as the non-recursive version. (fibonacci does more work)
  - The recursive version is shorter and simpler than the non-recursive solution (towers of hanoi)

Home work

- Write a recursive function to compute first N Fibonacci numbers. Test and trace for N = 6

  1 1 2 3 5 8

- Write a recursive function to compute power of a number (x^n). Test and trace for 4^5.

Outlook

Next week, we’ll discuss recursive sort