RECURSION

Data Structures and Algorithms

Waheed Iqbal

Department of Data Science, FCIT
University of the Punjab, Lahore, Pakistan

Introduction

Introduction (Cont.)

- Recursion is a technique that solves a problem by solving a smaller problem of the same type

- In recursion a method call itself repeatedly to solve a specific problem

Requirements for Recursive Solution

- At least one "*small*" case that you can solve directly
- A way of *breaking* a larger problem down into:
  - One or more *smaller* subproblems
  - Each of the *same kind* as the original
- A way of *combining* subproblem results into an overall solution to the larger problem

General Recursive Design Strategy

- Identify the *base case(s)* (for direct solution)
- Devise a problem *splitting strategy*
  - Subproblems must be smaller
  - Subproblems must work towards a base case
- Devise a solution *combining strategy*

Recursive Hello World!

Let's try to write a recursive hello world

```python
def print_recursive(n):
    if n <= 0:
        return
    else:
        print(f"{n}-Hello World")
        print_recursive(n - 1)

def main():
    print_recursive(10)

if __name__ == "__main__":
    main()
```

```
void print_recursive(int n)
{
   if (n<=0)
      return;
   else {
      cout<<n<<"-Hello World"<<endl;
      print(n-1);

   }
}

int main ()
{
   print_recursive(10);
   return 0;
}
```

Factorial

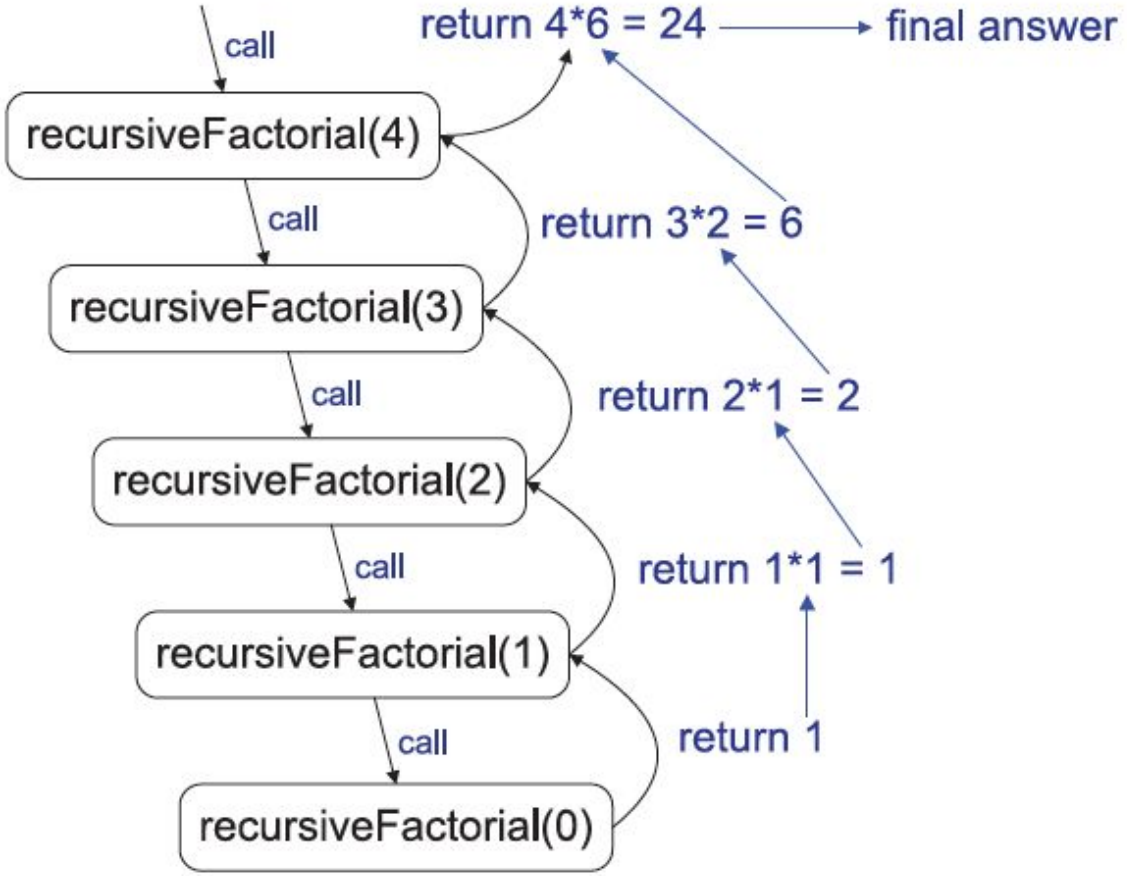
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

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

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

Recursive Factorial

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



A recursion trace for the call **recursiveFactorial(4)**

Array Sum

- we are given an array, **A**, of **n** integers that we want to sum together using recursion!

Array Sum

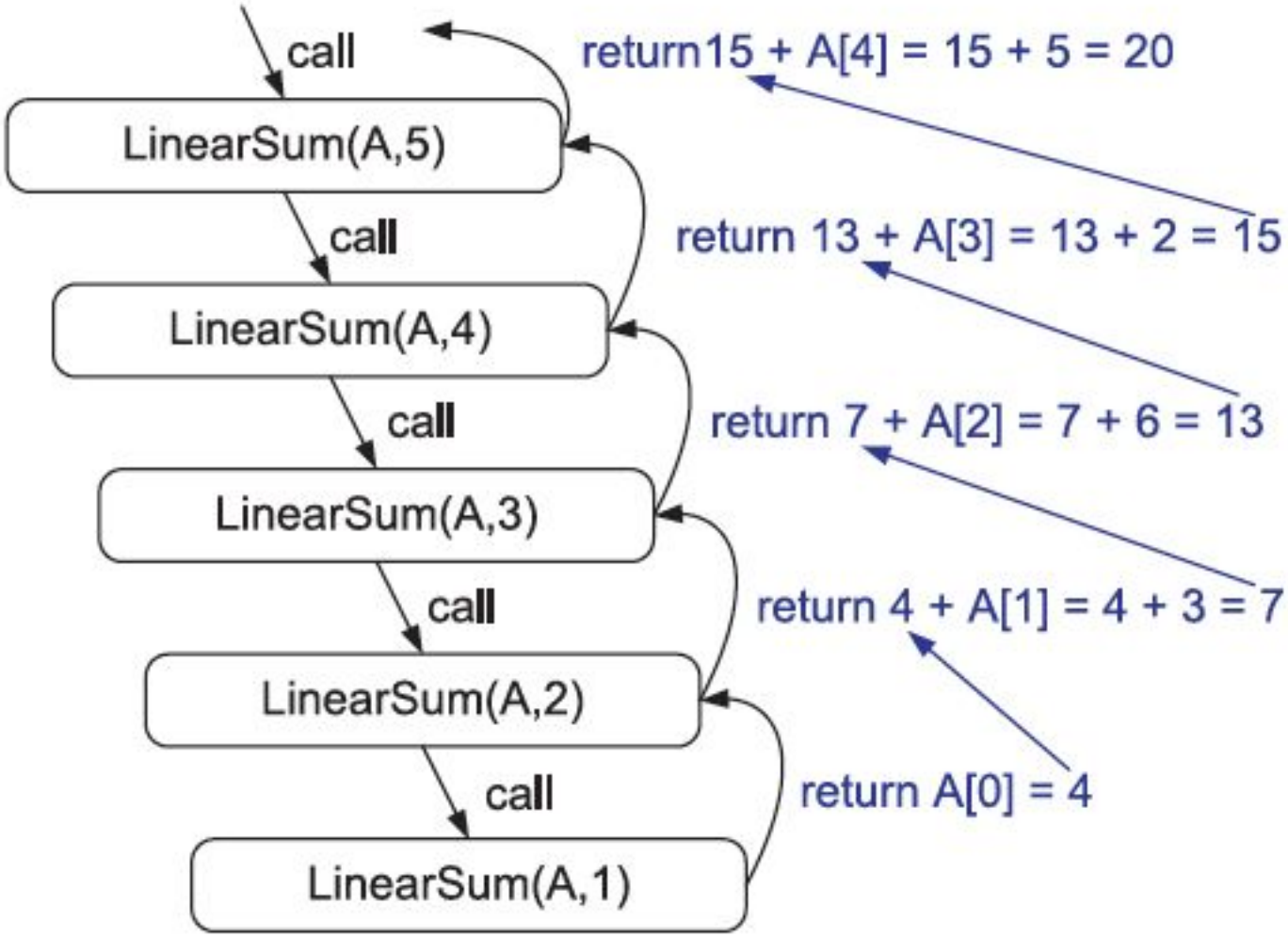
- we are given an array, **A**, of **n** integers that we want to sum together using recursion!

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]
```



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

Fibonacci Series

Recursive Fibonacci

1. Write a code to calculate nth Fibonacci series element

1. Now write a recursive implementation to do the same task

Recursive Fibonacci

```
int fib(int n)
{
     if (n == 0)
         return 0;
    if (n == 1)
         return 1;
    return fib(n-1)+fib(n-2);
}
```

Exercise: Draw recursion tree for fib(4)

Search a file in a folder recursively

Consider a scenario where you need to implement a function to search for a specific file within a folder. The folder may contain multiple subfolders, and the depth of these subfolders is unknown.

Binary Search

- Given a sorted array of length n, find an element by value.

Iterative Binary Search

```
int binary_search(int A[], iknt key, int imin, int imax)
{
  // continue searching while [imin,imax] is not empty
  while (imin <= imax)
    {
      // calculate the midpoint for roughly equal partition
      int imid = midpoint(imin, imax);
      if(A[imid] == key)
        // key found at index imid
        return imid;
      // determine which subarray to search
      else if (A[imid] < key)
        // change min index to search upper subarray
        imin = imid + 1;
      else
        // change max index to search lower subarray
        imax = imid - 1;
    }
  // key was not found
  return KEY_NOT_FOUND;
}
```

Recursive Binary Search

```
int bsearch(ListItem[] L, int k, int low, int high) {
   int mid = (low + high)/2;
   if (low > high)
      return -1;
   else if (L[mid].key == k)
      return mid;
   else if (L[mid].key < k)
      return bsearch(L, k, mid+1, high);
   else
      return bsearch(L, k, low, mid-1);
}
```