Data Structures and Algorithms

Chapter 5

Recursion

- A recursive function is a function which is defined in terms of itself.
- A recursion, in programming, is a way of implementing repeated execution of statements (or a method), where a method invokes itself.

- Example: Factorial

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

Recursion
Factorial

- Java implementation

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
      throw new IllegalArgumentException( );  // argument must be
                                              // nonnegative
  else if (n == 0)
      return 1;                         // base case
  else
      return n * factorial(n-1);        // recursive case
}
```

Recursion
Factorial

- Recursion trace

factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

1 * factorial(0)

returns 1

returns 1*1 = 1

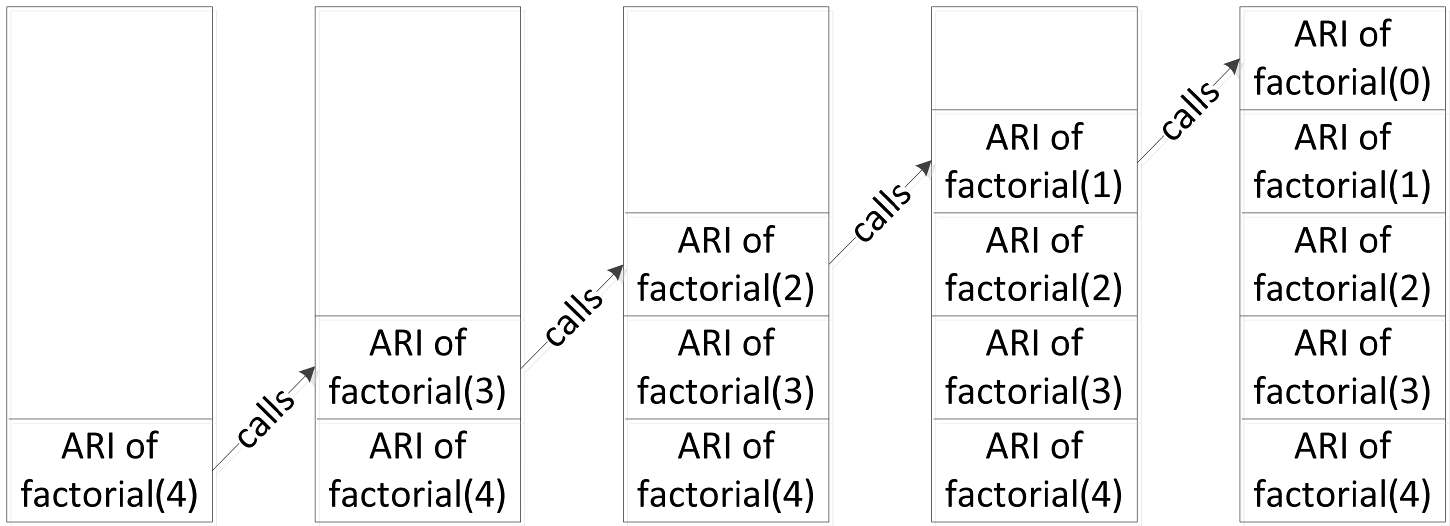
returns 2*1 = 2

returns 3*2 = 6

returns 4*6 = 24

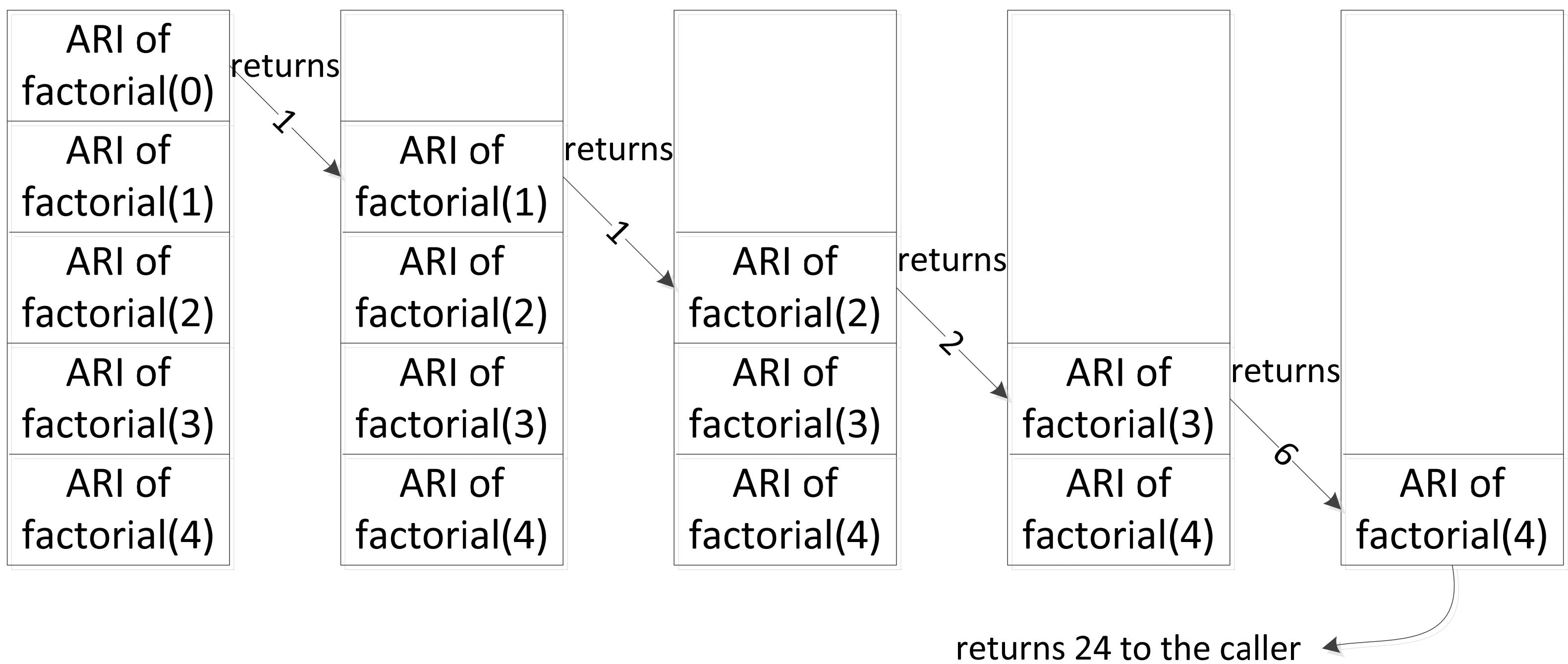
Recursion
Factorial

- Recursive calls



Recursion
Factorial

- Returning from calls



Recursion
Factorial

- Running time of *factorial*: $O(n)$
  - One execution of the method takes $O(1)$
  - It is invoked $n - 1$ times => $O(n)$
  - $O(1) \times O(n) = O(n)$

Recursion
Binary Search

- Search a sequence of n elements for a target element.
- Linear search
  - Examine each element while scanning the sequence
  - Best case: one comparison, or $O(1)$
  - Worst case: n comparisons, or $O(n)$
  - On average: $n/2$ comparisons, or $O(n)$

- Binary search
  - If the sequence is sorted, we can use binary search
  - Running time is $O(\log n)$

Recursion

Binary Search

Search 17

compare 17 with

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	19	23	29	32	41	45	54	66
low							mid							high

compare 17 with

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	19	23	29	32	41	45	54	66
low			mid			high								

compare 17 with

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	19	23	29	32	41	45	54	66
				low	mid	high								

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	19	23	29	32	41	45	54	66
						low = mid = high								

Recursion
Binary Search

- Pseudocode

```
Algorithm binarySearch(int[ ] data, int target, int low, int high)
  If low > high                         // target is not found
     return false
  else
     mid = floor((low + high)/2) // median candidate
     if target = data[mid]          // target found
        return true
     else if target < data[mid]
        search data[low .. mid-1] recursively
     else
        search data[mid+1 .. high] recursively
```

Recursion
Binary Search

- Java implementation

```
public static boolean binarySearch(int[ ] data, int target,
                                   int low, int high) {
if (low > high)
    return false;               // interval empty; no match
else {
    int mid = (low + high) / 2;
    if (target == data[mid])
        return true;            // found a match
    else if (target < data[mid]) // recurse left of the middle
        return binarySearch(data, target, low, mid - 1);
    else  // recurse right of the middle
        return binarySearch(data, target, mid + 1, high);
  }
}
```

Recursion
Binary Search

- Running time analysis
  - Execution of one call takes $O(1)$.
  - Each time binary search is (recursively) invoked, the number of elements to be searched is reduced to at most half.
  - Initially, there are n elements.
  - In the first recursive call, there are at most $n/2$ elements.
  - In the second recursive call, there are at most $n/4$ elements.
  - and so on …

Recursion
Binary Search

- Running time analysis (continued)
  - In the *j*-th recursive call, there are at most n / (2^j) elements.
  - In the worst case, the target is not in the sequence. In this case, recursion stops when there is no more elements to be searched.
  - The max. number of recursive calls is the smallest integer *r* such that $\frac{n}{2^r} < 1$
  - Or, *r* is the smallest integer such that $r > \log n$
  - Therefore, $r = \lfloor \log n \rfloor + 1$
  - So, the total running time is $O(\log n)$

Recursion
More Examples

- Print array elements recursively - Pseudocode

```
Algorithm printArrayRecursively(data, i)
    if i = n, return
    else
        print data[i]
        i = i + 1
        printArrayRecursively(data, i)
```

Recursion
More Examples

- Print array elements recursively – Java code

```
public static void printArrayRecursive(int[ ] data, int i){
  if (i == data.length)
    return;
 else{
   System.out.print(data[i++] + " ");
   printArrayRecursive(data, i);
 }
}
```

Recursion
More Examples

- Reverse sequence recursively – Pseudocode

```
Algorithm reverseArray(data, low, high)
   if low >= high, return
   else
      swap data[low] with data[high]
      reverseArray(data, low+1, high-1)
```

Recursion

More Examples

- Reverse sequence recursively – Java code

```
public static void reverseArray(int[ ] data, int low,
                                              int high) {
  if (low < high) {
    int temp = data[low];
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high - 1);
  }
}
```

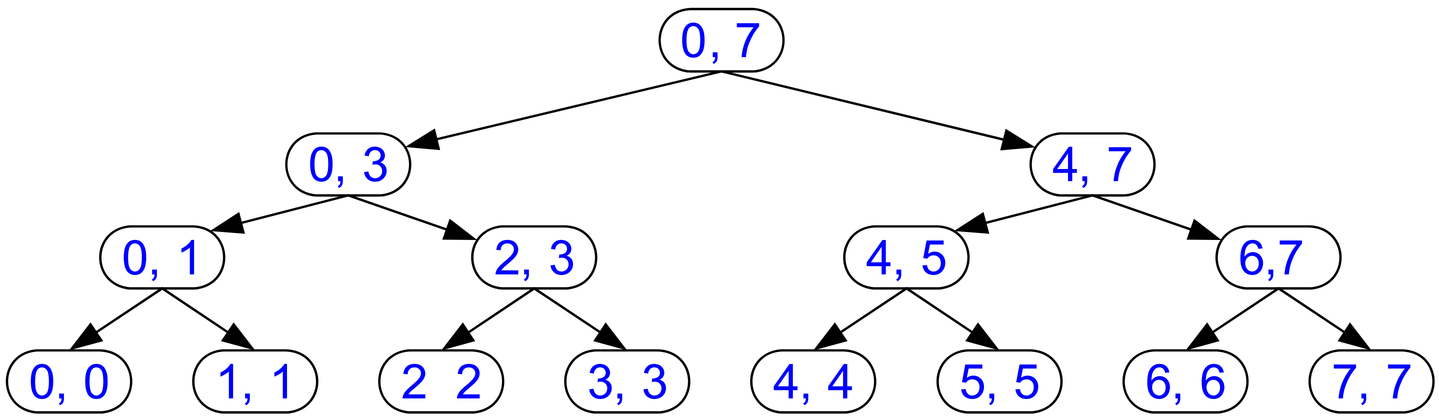
Recursion
More Examples

- Binary sum – Java code

```
public static int binarySum(int[ ] data, int low, int high) {
  if (low > high)              // zero elements in subarray
    return 0;
  else if (low == high)        // one element in subarray
    return data[low];
  else {
    int mid = (low + high) / 2;
    return binarySum(data, low, mid) +
                      binarySum(data, mid+1, high);
  }
}
```

Recursion
More Examples

- Binary sum – recursion trace



- Running time?

Recursion
Computing Powers

- Definition: $power(x, n) = x^n$
- Recursive definition

$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x * power(x, n-1) & \text{otherwise} \end{cases}$$

- Direct implementation

```
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
}
```

- Execution of each method call takes $O(1)$.
- The method is invoked (n + 1) times.
- Running time is $O(n)$

Recursion
Computing Powers

- There is an efficient method.
- Let $k = \left\lfloor \frac{n}{2} \right\rfloor$
- If *n* is even, $k = \frac{n}{2}$ and if *n* is odd, $k = \frac{n-1}{2}$
- So,

$$\left(x^k\right)^2 = \left(x^{\frac{n}{2}}\right)^2 = x^n$$ if *n* is even

$$\left(x^k\right)^2 = \left(x^{\frac{n-1}{2}}\right)^2 = x^{n-1}$$ if *n* is odd

Recursion
Computing Powers

- Then, we can redefine *power*(x, n) as follows:

$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n \text{ is even} \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 \cdot x & \text{if } n \text{ is odd} \end{cases}$$

Recursion

Computing Powers

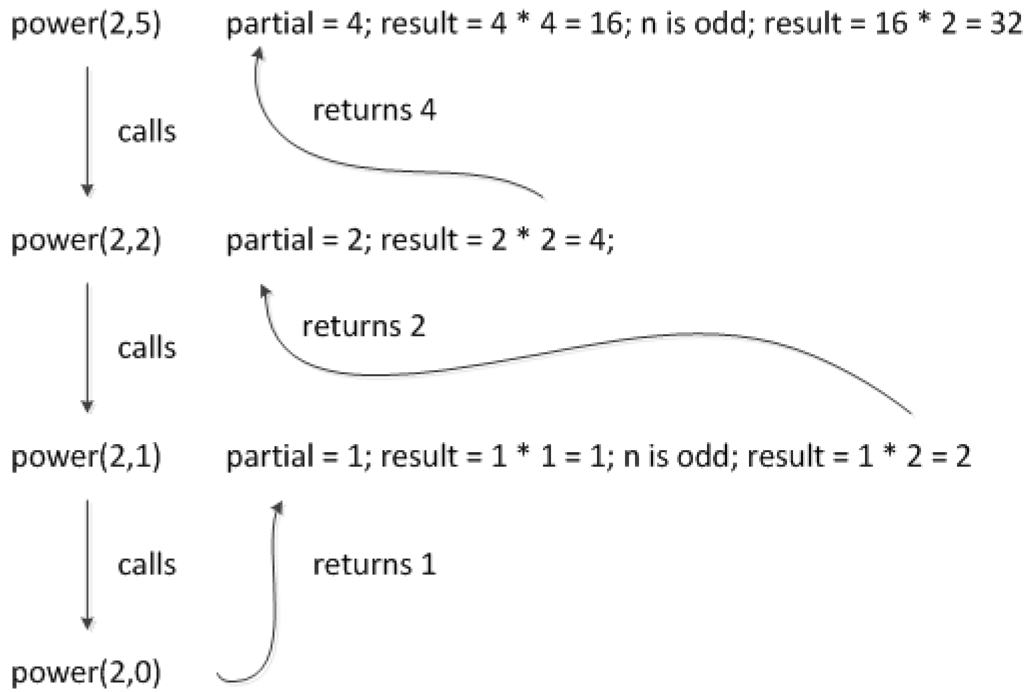
- Implementation

```
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else {
    double partial = power(x, n/2); // use integer division of n
    double result = partial * partial;
    if (n % 2 == 1)      // if n odd, include extra factor of x
      result *= x;
    return result;
  }
}
```

- Execution of one call takes $O(1)$.
- The method is invoked $O(\log n)$ times.
- Running time is $O(\log n)$

Recursion
Computing Powers

- Illustration



power(2,16) – power(2,8) – power(2,4) – power(2,2) – power(2,1) – power(2,0)
power(2,15) – power(2,7) – power(2,3) – power(2,1) – power(2,0)

Recursion
Designing Recursive Algorithms

- Two components: base case and recursion
- Base case:
  - Recursive call stops when a certain condition is met.
  - This is usually referred to as *base case*.
- Recursion: When the condition of the base case is not met, the algorithm invokes itself recursively.

- When poorly designed, very inefficient.
- Make sure the base case is always reached to avoid infinite recursion.

Recursion
Parameterizing Recursion

- Design of recursive algorithms sometimes requires the change of signature by adding more parameters.

- Natural signature of binary search:

  binarySearch (data, target)
- Recursive design requires additional parameters:

  binarySearch(data, target, low, high)

- Cleaner public interface:

```
public static boolean binarySearch(int[ ] data, int target) {
    return binarySearch(data, target, 0, data.length – 1);
}
```

Recursion
Tail Recursion

- Recursion allows exploitation of repetitive structure of a problem.
- Makes algorithm description more readable; avoids complex analyses and nested loops.
- Requires more memory.
- Tail recursion: A recursive call is the last operation.
- A tail recursion can be converted to a non-recursive algorithm (or implementation) that does not use additional memory.
- Example: binary search

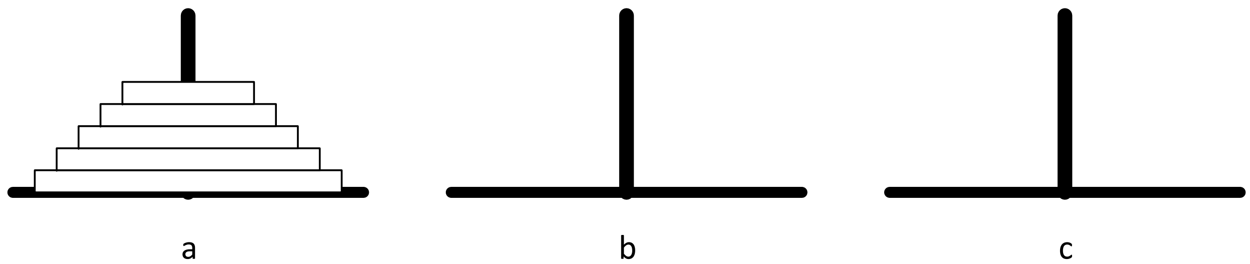
Recursion

Towers of Hanoi (Exercise)

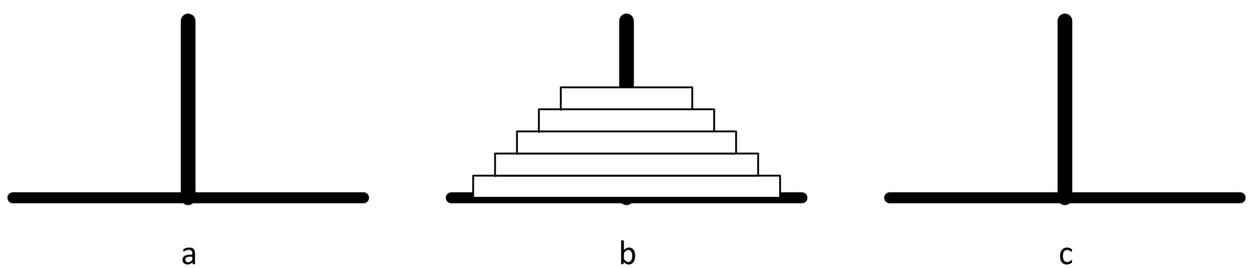
- Well known problem
- Given 3 pegs *a*, *b*, and *c*
- Peg *a* has *n* disks, the smallest on the top and the largest at the bottom
- Move all disks from *a* to *b*
- Use *c* as a temporary peg

Recursion
Towers of Hanoi (Exercise)

- Initial



- Final



Recursion
Towers of Hanoi (Exercise)

- When moving disks:
  - One disk at a time
  - Never place a larger disk on top of a smaller disk
- Each disk has a label
  - Label of the smallest disk is 1
  - Label of the next smallest disk is 2
  - . . .
  - Label of the largest disk is *n*

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, "Data Structures and Algorithms in Java," Sixth Edition, Wiley, 2014.