CS121 Data Structures A, C
Recursion

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Recursion

Recursion is a process to achieve repetition within a computer program

It is a technique by which a method makes one or more calls to itself during execution

By recursion a data structure relies upon smaller instances of the very same type of structure in its representation

Recursion By Example

- the factorial function, $n!$
- an English ruler
- binary search

The Factorial Function

For any integer $n \geq 0$,

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

The **recursive definition**

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

recursive definition = one or more base cases (fixed values) + one or more recursive cases (in terms of itself)

The Factorial Function: A Recursive Implementation

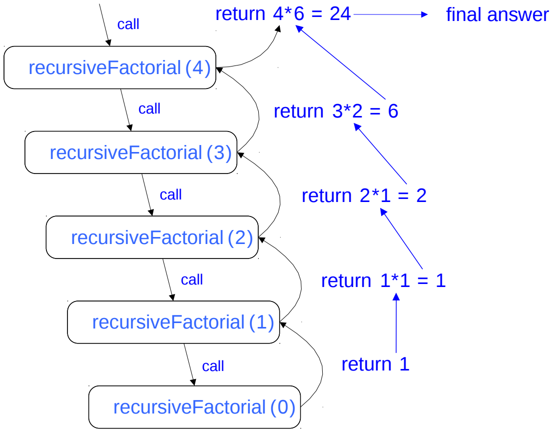
```java
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );
                                // argument must be nonnegative
  else if (n == 0)
    return 1;                                   // base case
  else
    return n * factorial(n-1);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

Recursion Trace

A box for each recursive call

An arrow from each caller to callee

An arrow from each callee to caller showing return value



Drawing an English Ruler

major tick length and minor ticks

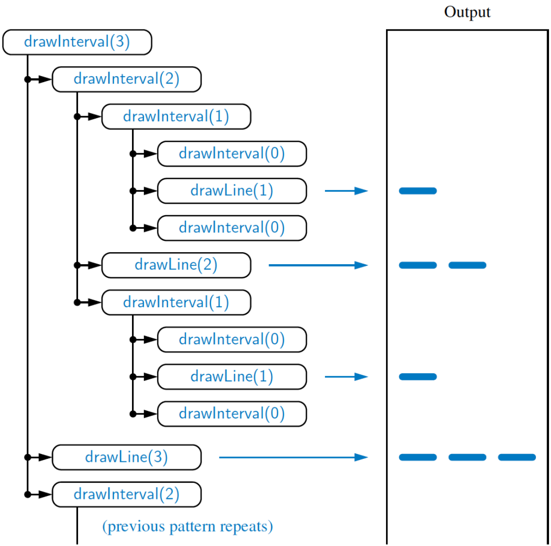
```
---- 0          ----- 0         --- 0
-               -               -
--              --              --
-               -               -
---             ---             --- 1
-               -               -
--              --              --
-               -               -
---- 1          ----            --- 2
-               -               -
--              --              --
-               -               -
---             ---             --- 3
-               -
--              --
-               -
---- 2          ----- 1
```

The English ruler pattern is a simple example of a **fractal**, i.e. a shape that has a self-recursive structure at various levels of magnification

Recursion Trace for English Ruler

An interval with a central tick length $L \geq 1$ is composed of:

- an interval with a central tick length $L - 1$
- a single tick of length L
- an interval with a central tick length $L - 1$

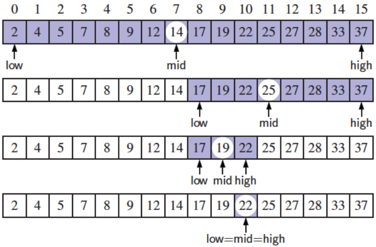


English Ruler: A Recursive Implementation

```java
/** Draws an English ruler for the given number of inches and major tick length. */
public static void drawRuler(int nInches, int majorLength) {
  drawLine(majorLength, 0);                  // draw inch 0 line and label
  for (int j = 1; j <= nInches; j++) {
    drawInterval(majorLength - 1);           // draw interior ticks for inch
    drawLine(majorLength, j);                // draw inch j line and label
  }
}
private static void drawInterval(int centralLength) {
  if (centralLength >= 1) {                  // otherwise, do nothing
    drawInterval(centralLength - 1);         // recursively draw top interval
    drawLine(centralLength);                 // draw center tick line (without label)
    drawInterval(centralLength - 1);         // recursively draw bottom interval
  }
}
private static void drawLine(int tickLength, int tickLabel) {
  for (int j = 0; j < tickLength; j++)
    System.out.print("-");
  if (tickLabel >= 0)
    System.out.print(" " + tickLabel);
  System.out.print("\n");
}
/** Draws a line with the given tick length (but no label). */
private static void drawLine(int tickLength) {
  drawLine(tickLength, -1);
}
```

Binary Search

Binary search is used to efficiently locate a target value within a sorted sequence of n elements stored in an array



We consider three cases:

- If the target equals data[mid], then we have found the target.
- If target $<$ data[mid], then we recur on the first half of the sequence.
- If target $>$ data[mid], then we recur on the second half of the sequence.

Binary Search: A Recursive Implementation

```java
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
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])
      return binarySearch(data, target, low, mid - 1);  // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high); // recur right of the middle
  }
}
```

The Factorial Function: Efficiency

For each invocation of the method, only account for the number of operations that are performed within the body of that activation. Then take the sum over all activations

A total of $n + 1$ activations $(n, n - 1, \ldots, 1, 0)$

a constant number of operations in each activation, i.e. $O(1)$

Thus, the overall number of operations is $O(n)$

Drawing an English Ruler: Efficiency

Proposition: For $c \geq 0$, a call to drawInerval(c) results in precisely $2^c - 1$ lines of output.

Justification: A formal proof by *induction*

base case: drawInterval(0) generates no output, and
$2^0 - 1 = 1 - 1 = 0$

induction step: drawInterval(c) prints lines one more (center line) than twice the number generated by drawInterval($c - 1$);
$1 + 2 \times (2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$

Drawing an English Ruler: Efficiency

Proposition: For $c \geq 0$, a call to drawInerval(c) results in precisely $2^c - 1$ lines of output.

Justification: A formal proof by *induction*

base case: drawInterval(0) generates no output, and $2^0 - 1 = 1 - 1 = 0$

induction step: drawInterval(c) prints lines one more (center line) than twice the number generated by drawInterval($c - 1$); $1 + 2 \times (2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$

Hence, the overall number of operations is $\Omega(2^n)$

By further analysis, it is also $O(2^n)$; thus it is $\Theta(2^n)$

Binary Search: Efficiency

A constant number of operations in each activation, i.e. $O(1)$

Proposition: The binary search algorithm runs in $O(\log n)$ time for a sorted array with n elements.

Justification: With each recursive call the number of candidate elements still to be searched is: $\text{high} - \text{low} + 1$

reduced by at least one-half, i.e.

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}$$

After j^{th} call, the number of candidates at most $n/2^j$

$n/2^r < 1$. Thus we have $r = \lfloor \log n \rfloor + 1$, i.e. binary search runs in $O(\log n)$ time.

Types of Recursion

linear recursion a recursive call starts at most one other

binary recursion a recursive call may start two others

multiple recursion a recursive call may start three or more others

Linear Recursion

Examples we have seen?

Linearity of recursion reflects the structure of the recursion trace, not the asymptotic analysis of the running time

Examples of linear recursion with non-linear running time?

Sum of Array Elements

We want to compute the sum of an array of n integers using recursion

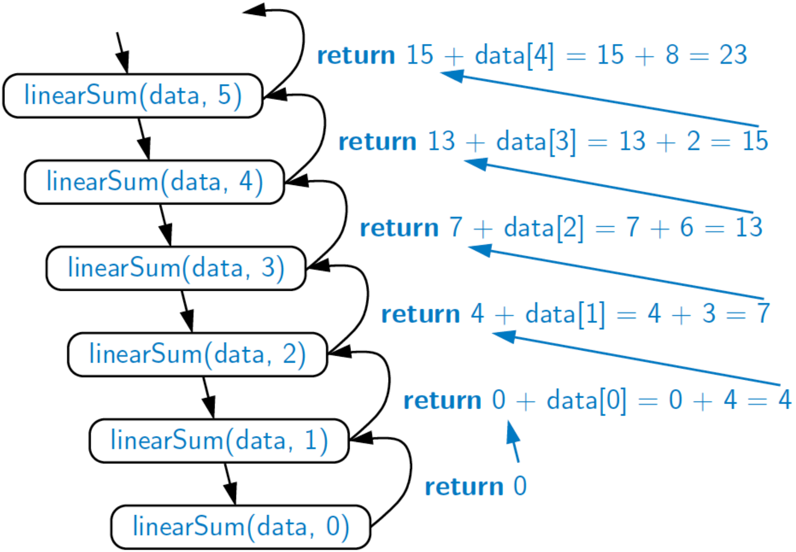
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7

If $n = 0$ the sum is 0; for $n > 0$, add the last number to the sum of the first $n - 1$

Sum of Array Elements: A Recursive Implementation

```java
/** Returns the sum of the first n integers of the given array. */
public static int linearSum(int[ ] data, int n) {
  if (n == 0)
    return 0;
  else
    return linearSum(data, n-1) + data[n-1];
}
```

Sum of Array Elements: Recursion Trace



time $O(n)$; memory $O(n)$

Reversing a Sequence

We want to reverse the n elements of an array

0	1	2	3	4	5	6	7
4	3	6	2	7	8	9	5
5	3	6	2	7	8	9	4
5	9	6	2	7	8	3	4
5	9	8	2	7	6	3	4
5	9	8	7	2	6	3	4

swap the first and last elements and so on

Reversing a Sequence: A Recursive Implementation

```java
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                   // if at least two elements in subarray
    int temp = data[low];             // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high - 1);  // recur on the rest
  }
}
```

time $O(n)$; memory $O(n)$

Computing Powers

We want to raise a number x to an arbitrary nonnegative integer n, i.e. $power(x, n) = x^n$

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

since $x^n = x \cdot x^{n-1}$ for $n > 0$

Computing Powers: A Recursive Implementation

```
/** Computes the value of x raised to the nth power,
    for nonnegative integer n. */
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
}
```

time $O(n)$; memory $O(n)$

Computing Powers: A Faster Approach

We want to raise a number x to an arbitrary nonnegative integer n, i.e. $power(x, n) = x^n$

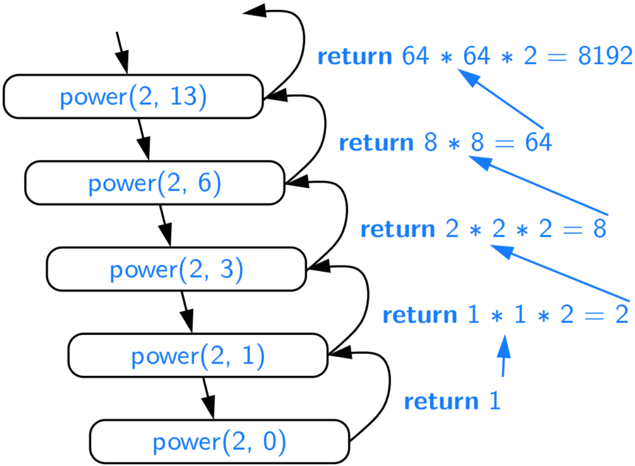
$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ (power(x, \lfloor \frac{n}{2} \rfloor))^2 \cdot x & \text{if } n > 0 \text{ is odd} \\ (power(x, \lfloor \frac{n}{2} \rfloor))^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

since $x^n = (x^k)^2$ for even n, and $x^n = (x^k)^2 \cdot x$ for odd n, where $k = \lfloor \frac{n}{2} \rfloor$

Computing Powers: A Fast Recursive Implementation

```
/** Computes the value of x raised to the nth power, for nonnegative integer n. */
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else {
    double partial = power(x, n/2);          // rely on truncated division of n
    double result = partial * partial;
    if (n % 2 == 1)                          // if n odd, include extra factor of x
      result *= x;
    return result;
  }
}
```

Computing Powers: Recursion Trace



time $O(\log n)$; memory $O(\log n)$

Binary Recursion

Examples we have seen?

Sum of Array Elements

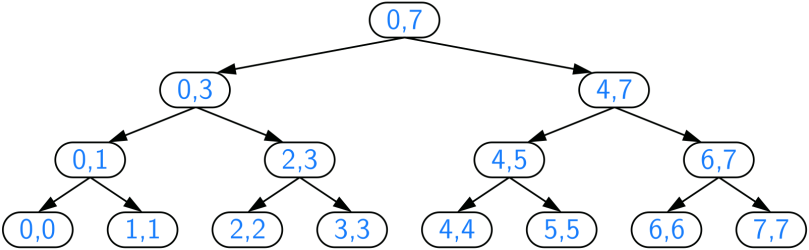
We want to compute the sum of an array of n integers using *binary* recursion

Recursively compute the sum of the first half, and the sum of the second half, and add those sums together

Sum of Array Elements: A Binary Recursive Implementation

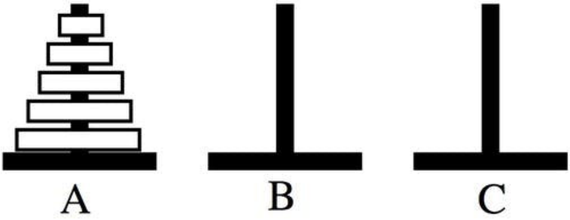
```java
/** Returns the sum of subarray data[low] through data[high] inclusive. */
public static int binarySum(int[ ] data, int low, int high) {
  if (low > high)                    // zero elements in subarray
    return 0;
  else if (low == high)              // one element in subarray
    return data[low];
  else {
    int mid = (low + high) / 2;
    return binarySum(data, low, mid) + binarySum(data, mid+1, high);
  }
}
```

Sum of Array Elements: Binary Recursion Trace



time $O(n)$; memory $O(\log n)$ (excluding the array)

Towers of Hanoi



Move all the disks from peg A to peg C, moving one disk at a time, so that we never place a larger disk on top of a smaller one

Towers of Hanoi: A Binary Recursive Implementation

```java
/** Java recursive function to solve tower of hanoi puzzle */
public static void towerOfHanoi(int n, char from, char to, char aux) {
  if (n == 1) {
    System.out.println("Move disk 1 from rod " +  from + " to rod " + to);
    return;
  }
  towerOfHanoi(n-1, from, aux, to);
  System.out.println("Move disk " + n + " from rod " +  from
     + " to rod " + to);
  towerOfHanoi(n-1, aux, to, from);
}
```

time $O(2^n)$; memory $O(n)$

Designing Recursive Algorithms

A typical form for a recursive algorithm:

- Test for base cases. Test for a set of base cases (at least one) without recursion. Make sure every possible chain of recursive calls reaches a base case.
- Recur. If not a base case, perform one or more recursive calls. May involve a test to choose a recursive call. Make sure each possible recursive call makes progress towards a base case.

Work out the problem on a few concrete examples to see how the subproblems should be defined

Parameterizing a Recursion

Sometimes we need to reparameterize the signature of the method to define recursive subproblems

binarySearch(data, target, low, high) vs. binarySearch(data, target)

Other examples: reverseArray, linearSum, binarySum

A standard technique for a cleaner public interface: make the recursive version private, and introduce a cleaner public method

```
/** Returns true if the target value is found in the data array. */
public static boolean binarySearch(int[ ] data, int target) {
  return binarySearch(data, target, 0, data.length - 1);
}
```

Misuse of Recursion

- inefficient recursion, i.e. large execution times as a result of bad recursive design
- infinite recursion, i.e. making recursive calls without reaching a base case
- large recursive depths reaching the memory limit

Fibonacci Numbers: Inefficient Recursion

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

Fibonacci Numbers: Inefficient Recursion

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

```java
/** Returns the nth Fibonacci number (inefficiently). */
public static long fibonacciBad(int n) {
  if (n <= 1)
    return n;
  else
    return fibonacciBad(n-2) + fibonacciBad(n-1);
}
```

The number of calls $c_n > 2^{n/2}$ is exponential in n

Fibonacci Numbers: Efficient Recursion

Let's rely on linear recursion instead!

Fibonacci Numbers: Efficient Recursion

Let's rely on linear recursion instead!

```
/** Returns array containing the pair of Fibonacci numbers,
 F(n) and F(n-1). */
public static long[ ] fibonacciGood(int n) {
  if (n <= 1) {
    long[ ] answer = {n, 0};
    return answer;
  } else {
    long[ ] temp = fibonacciGood(n - 1);
                                        // returns {F(n-1), F(n-2)}
    long[ ] answer = {temp[0] + temp[1], temp[0]};
                                        // we want {F(n), F(n-1)}
    return answer;
  }
}
```

time $O(n)$; memory $O(n)$

Tail Recursion

Tail recursion occurs when a linearly recursive method makes its recursive call as its last step

Tail recursion examples: binarySearch, reverseArray

Such methods can be easily converted to non-recursive methods (which saves on some resources)

```
/** Returns true if the target value is found in the data array. */
public static boolean binarySearchIterative(int[ ] data, int target) {
  int low = 0;
  int high = data.length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (target == data[mid])                  // found a match
      return true;
    else if (target < data[mid])
      high = mid - 1;                         // only consider values left of mid
    else
      low = mid + 1;                          // only consider values right of mid
  }
  return false;                               // loop ended without success
}
```

Summary

Reading

Chapter 5 Recursion

Questions?