Recursion

- Refers to methods that call themselves
- Can be direct or indirect
- Many applications in computer science

Definitions

- A recursive method contains the following:
  - Anchor case: the terminating condition
  - Inductive step: re-calls the method, building upon previously-solved cases
- Example: factorial
  - \( f(n) = n! = n(n-1)(n-2)...(2)(1) \)
  - anchor case: \( n = 1 \)
  - inductive step: \( n(n-1)! \)

Factorial example: code

```java
public long factorial(long n) {
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n-1));
}
```
How recursion is executed

- Each call to the recursive method results in an activation record being added to the run-time stack
- An activation record contains
  - local variables
  - parameter values
  - a return value holder (if applicable)
  - the return address (of the caller)

How recursion is executed (cont.)

- Stack pointer points to most recent call to the method
- Anchor case is solved and its activation record is popped from the stack
- One by one, each step builds upon the previous solution, and the corresponding activation record is popped, until the solution is reached

Types of recursion

- Tail recursion: call to self is at the end of a method
  - example: factorial calculation
- Non-tail recursion: call to self is at the beginning or in the middle of a method
  - example: reversing the order of a string
- Indirect recursion: call to self through another method
  - example: A calls B, B calls C, C calls A
  - example: sine, cosine, and tangent

Nested recursion

- Method calls itself
- A parameter to this method is also a call to the method
- Example:
  - \( f(n) = 1, \quad n \leq 2 \)
  - \( f((n-2)/3), \quad n > 2 \)
Recursion vs. iteration

- We can usually rewrite recursive methods as *iterative methods* (methods that do not call themselves).
- Sometimes this is preferable, but sometimes it is more work.

Example: fibonacci numbers

```java
public long fibonacci(long n) {
    if (n == 0 || n == 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Problems with the recursive fibonacci method

- Each step requires 2 recursive calls.
- Complexity is $2^n$!
- Calculate `fibonacci(1)` and `fibonacci(0)` over and over again.
- A better solution: use an approximation or use an iterative method.

A non-recursive fibonacci calculation

```java
public long fibonacci(long n) {
    if (n < 2)
        return n;
    else {
        long tmp;
        long current = 1;
        long last = 0;
        for (long i = 2; i <= n; ++i) {
            tmp = current;
            current += last;
            last = temp;
        }
        return current;
    }
}
```
Backtracking

- Application of recursion
- Try a solution; if it doesn't work, “backtrack” over previous solutions and try another path
- Many AI applications
- Example: 8 Queens problem

Recursion vs. iterative methods

- Advantages
  - elegant, brief
  - system handles the stack
- Disadvantages
  - slower
  - may repeat calculations unnecessarily (less efficient)
  - easy to make mistakes and have a recursive method that goes on forever