CS2010: ALGORITHMS AND DATA STRUCTURES

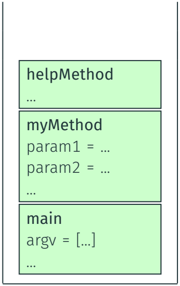
Lecture 10: Recursion vs Iteration

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

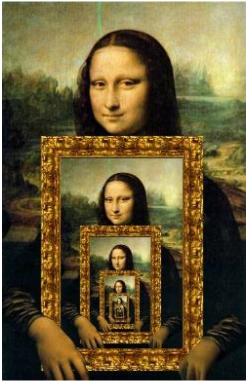
HOW METHODS EXECUTE

- Call stack: is a stack maintained by the Java runtime system
- One call stack frame (aka activation record) for each running instance of a method: contains all information necessary to execute the method
  - **references** to parameter values and local objects, return address etc.
- Objects themselves are stored in another part of memory: the heap
- every time a method is called, a new stack frame is pushed on the call stack.
- every time a method returns, the top-most stack frame is popped.



Recursion: when something is defined in terms of itself.

Infinite Recusion



Well-founded Recursion

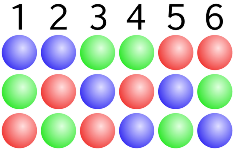
Principle: A method is recursive when its definition **calls the method itself**.

A correct recursive method should be well-founded: it should terminate/must end up at a base case.

Classic example: factorial – in math written as: $n!$

Math definition:

$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \text{when } n > 0$$



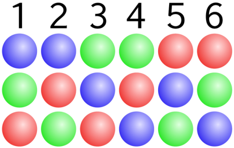
Principle: A method is recursive when its definition **calls the method itself**.

A correct recursive method should be well-founded: it should terminate/must end up at a base case.

Classic example: factorial – in math written as: $n!$

Math definition:

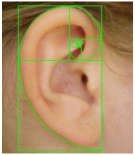
$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \text{when } n > 0$$

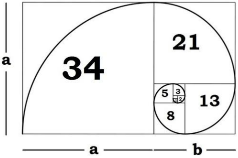


Classic example: Fibonacci numbers

Math definition:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fin(n) = fib(n-1) + fib(n-2) \text{ when } n > 1$$





RECURSION

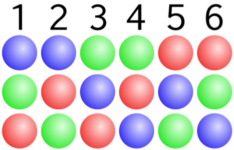
Principle: A method is recursive when its definition **calls the method itself**.

A correct recursive method should be well-founded: it should terminate/must end up at a base case.

Classic example: factorial – in math written as: $n!$

Math definition:

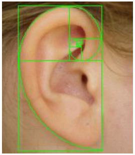
$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \text{when } n > 0$$

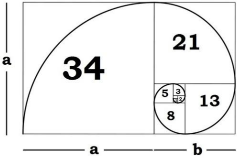


Classic example: Fibonacci numbers

Math definition:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fin(n) = fib(n-1) + fib(n-2) \text{ when } n > 1$$





It is convenient to implement recursive math definitions using recursive methods.

Recursive implementation:

```
int fac(int n) {



}
```

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

→ Closely matches the mathematical definition.

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

→ Closely matches the mathematical definition.

→ follows the divide and conquer approach:

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

- → Closely matches the mathematical definition.
- → follows the divide and conquer approach:
  - → break the implementation of `fac(n)` into smaller and smaller parts: `fac(n-1)`

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

- → Closely matches the mathematical definition.
- → follows the divide and conquer approach:
  - → break the implementation of `fac(n)` into smaller and smaller parts: `fac(n-1)`
  - → only deal with the **smallest possible cases** (here when $n = 0$): the base cases

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

- → Closely matches the mathematical definition.
- → follows the divide and conquer approach:
  - → break the implementation of `fac(n)` into smaller and smaller parts: `fac(n-1)`
  - → only deal with the **smallest possible cases** (here when $n = 0$): the base cases
  - → the rest of the cases are the recursive cases (here when $n > 0$)

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

- → Closely matches the mathematical definition.
- → follows the divide and conquer approach:
  - → break the implementation of `fac(n)` into smaller and smaller parts: `fac(n-1)`
  - → only deal with the **smallest possible cases** (here when $n = 0$): the base cases
  - → the rest of the cases are the recursive cases (here when $n > 0$)
  - → specify how smaller solutions **compose** into the solutions of recursive cases

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

- → Closely matches the mathematical definition.
- → follows the divide and conquer approach:
  - → break the implementation of `fac(n)` into smaller and smaller parts: `fac(n-1)`
  - → only deal with the **smallest possible cases** (here when $n = 0$): the base cases
  - → the rest of the cases are the recursive cases (here when $n > 0$)
  - → specify how smaller solutions **compose** into the solutions of recursive cases
  - → it is usually a **top-down calculation**

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

→ Q: asymptotic worst-case running time? (# recursive calls * cost of each call)

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

→ Q: asymptotic worst-case running time? (# recursive calls * cost of each call)
A: $\Theta(n)$ time

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

- Q: asymptotic worst-case running time? (# recursive calls * cost of each call)
  A: $\Theta(n)$ time
- Q: memory space for call stack frames? (max # frames on call stack)

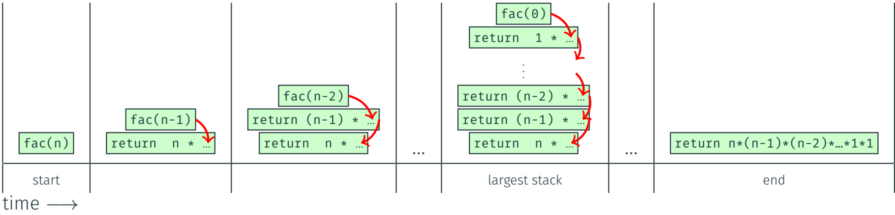
IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n-1);
}
```

→ Q: asymptotic worst-case running time? (# recursive calls * cost of each call)
A: $\Theta(n)$ time

→ Q: memory space for call stack frames? (max # frames on call stack)
A: $\Theta(n)$ space
All this call stack space is needed because of the `return n *` …



Recursive implementation using accumulator: (H/W: can you implement the accumulator version bottom-up?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

Recursive implementation using accumulator: (H/W: can you implement the accumulator version bottom-up?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

→ Q: asymptotic worst-case running time?

Recursive implementation using accumulator: (H/W: can you implement the accumulator version bottom-up?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

→ Q: asymptotic worst-case running time? $\Theta(n)$

Recursive implementation using accumulator: (H/W: can you implement the accumulator version bottom-up?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

→ Q: asymptotic worst-case running time? $\Theta(n)$

→ Q: memory space for call stack frames?

Recursive implementation using accumulator: (H/W: can you implement the accumulator version bottom-up?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

→ Q: asymptotic worst-case running time? $\Theta(n)$

→ Q: memory space for call stack frames?

  → In Java $\Theta(n)$ for stack space

  → In other, mainly functional, languages (ML, Lisp, Haskell, ...) the compiler runs this using $\Theta(1)$ stack space.
  Only the top-most stack frame is necessary because every function call simply returns the inner result: `return facAcc(n-1, acc * n)`
  This is called tail recursion

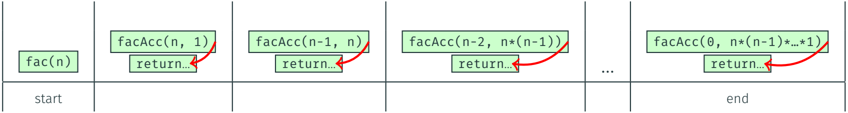
IMPLEMENTATION OF FACTORIAL

Recursive implementation using accumulator: (H/W: can you implement the accumulator version bottom-up?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

- Q: asymptotic worst-case running time? $\Theta(n)$
- Q: memory space for call stack frames?
  - In Java $\Theta(n)$ for stack space
  - In other, mainly functional, languages (ML, Lisp, Haskell, …) the compiler runs this using $\Theta(1)$ stack space.
    Only the top-most stack frame is necessary because every function call simply returns the inner result: `return facAcc(n-1, acc * n)`
    This is called tail recursion



From **tail recursive implementation** ⟶ iterative implementation:

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

```
int fac(int n) {
    int acc = 1;
    for ( ; !(n == 0); n--) {



       acc = acc * n;
    }
    return acc;
}
```

From **tail recursive implementation** $\longrightarrow$ iterative implementation:

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

```
int fac(int n) {
    int acc = 1;
    for ( ; !(n == 0); n--) {



       acc = acc * n;
    }
    return acc;
}
```

→ Running time of iterative implementation: $\Theta(n)$

From **tail recursive implementation** $\longrightarrow$ iterative implementation:

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return facAcc(n-1, acc * n);
}
```

```
int fac(int n) {
    int acc = 1;
    for ( ; !(n == 0); n--) {



       acc = acc * n;
    }
    return acc;
}
```

- Running time of iterative implementation: $\Theta(n)$
- Stack space of iterative implementation: $\Theta(1)$
  In functional languages this simple translation is done by the compiler!

Math definition:

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fin(n) &= fib(n-1) + fib(n-2) \quad \text{when } n > 1 \end{aligned}$$

Recursive implementation:

Math definition:

$$
\begin{aligned}
fib(0) &= 1 \\
fib(1) &= 1 \\
fin(n) &= fib(n-1) + fib(n-2) \quad \text{when } n > 1
\end{aligned}
$$

Recursive implementation:

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

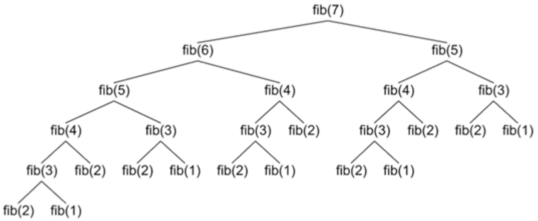
Math definition:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fin(n) = fib(n-1) + fib(n-2) \quad \text{when } n > 1$$

Recursive implementation:

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

- → Running time: non-tight upper bound: $O(2^n)$ tight bound: $\Theta(fib(n))$
  - → # of recursive calls of fib(n) is the size of the binary tree of recursive calls with n levels ($\leq 2^n$); however this is not a complete tree; thus $O(2^n)$.



FIBONACCI NUMBERS

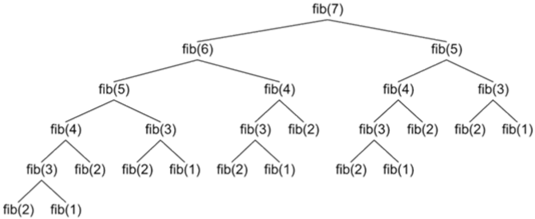
Math definition:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fin(n) = fib(n-1) + fib(n-2) \quad \text{when } n > 1$$

Recursive implementation:

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

- Running time: non-tight upper bound: $O(2^n)$ tight bound: $\Theta(fib(n))$
  - # of recursive calls of fib(n) is the size of the binary tree of recursive calls with n levels ($\leq 2^n$); however this is not a complete tree; thus $O(2^n)$.



- Call stack space: $\Theta(n)$

Math definition:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fin(n) = fib(n-1) + fib(n-2) \quad \text{when } n > 1$$

Recursive implementation with accumulator (tail recursion):

```
int fib(int n) { return fibAcc(n, 1, 1); }

int fibAcc(int n, int last, int secondToLast) {
  if (n <= 1) return last;
  else return fibAcc(n-1, last + secondToLast, last);
}
```

→ This is much trickier! Read it again off-line and understand why it works.

→ It is a bottom-up calculation using two accumulators.

Math definition:

$$
\begin{aligned}
fib(0) &= 1 \\
fib(1) &= 1 \\
fin(n) &= fib(n-1) + fib(n-2) \quad \text{when } n > 1
\end{aligned}
$$

Recursive implementation with accumulator (tail recursion):

```
int fib(int n) { return fibAcc(n, 1, 1); }

int fibAcc(int n, int last, int secondToLast) {
  if (n <= 1) return last;
  else return fibAcc(n-1, last + secondToLast, last);
}
```

→ This is much trickier! Read it again off-line and understand why it works.

→ It is a bottom-up calculation using two accumulators.

→ Running time: $\Theta(n)$

→ Call stack space: $\Theta(n)$ in Java and $\Theta(1)$ in other languages.

Tail-recursive implementation ⟶ iterative implementation

```
int fib(int n) { return fibAcc(n, 1, 1); }

int fibAcc(int n, int last, int secondToLast) {
  if (n <= 1) return last;
  else
    return fibAcc(n-1, last + secondToLast, last);
}
```

```
int fib(int n) {
  int last = 1; int secondToLast = 1;
  for ( ; !(n <= 1); n--) {
    int tmpLast = last;
    last = last + secondToLast;
    secondToLast = tmpLast;
  }
  return last;
}
```

→ Worst Case Asymptotic Running time: $\Theta(n)$

→ Call stack space: $\Theta(1)$

gcd(x,y) is the largest number n such that $x\%n = y\%n = 0$.

For simplicity assume $x \leq y$.

gcd(x,y) is the largest number n such that $x\%n = y\%n = 0$.

For simplicity assume $x \leq y$.

Attempt 1: try all numbers n from **+inf** down to 1 until you find one that has the above property.

gcd(x,y) is the largest number n such that $x\%n = y\%n = 0$.

For simplicity assume $x \leq y$.

Attempt 1: try all numbers n from **+inf** down to 1 until you find one that has the above property.

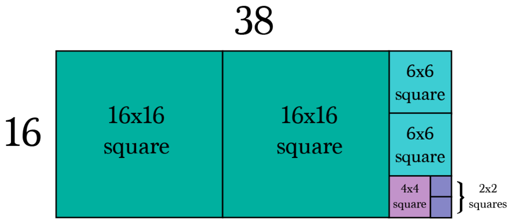
Attempt 2: try all numbers n from **x** down to 1 until you find one that has the above property (because gcd will necessarily be $\leq min(x, y)$).

gcd(x,y) is the largest number n such that $x\%n = y\%n = 0$.

For simplicity assume $x \leq y$.

Attempt 3: Euclid's algorithm a Divide & Conquer approach:

Euclid's Theorem: gcd(x,y) = gcd(y, x % y).



The base case here is gcd(x,0) = x (why is this the base case?).

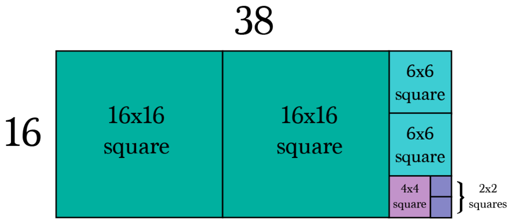
→ Q: How many iterations in the worst case?

gcd(x,y) is the largest number n such that $x\%n = y\%n = 0$.

For simplicity assume $x \leq y$.

Attempt 3: Euclid's algorithm a Divide & Conquer approach:

Euclid's Theorem: gcd(x,y) = gcd(y, x % y).



The base case here is gcd(x,0) = x (why is this the base case?).

- Q: How many iterations in the worst case?
- **Gabriel Lame's Theorem (1844):** #iterations $< 5 \cdot h$
  Where $h =$ digits of $min(x, y)$ (here this is x) in base 10.
- A: $O(lg(x))$

→ It **depends** on the benefits vs the drawbacks each implementation gives us in each case.

→ It **depends** on the benefits vs the drawbacks each implementation gives us in each case.

→ Recursive algorithms usually have easier proofs of correctness (Divide and Conquer is more obvious and similar to mathematical induction)

RECURSION VS ITERATION: WHICH ONE TO CHOOSE?

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.
- Recursive algorithms usually have easier proofs of correctness (Divide and Conquer is more obvious and similar to mathematical induction)
- Iterative algorithms usually have easier proofs of memory usage (and in Java use less memory)

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.
- Recursive algorithms usually have easier proofs of correctness (Divide and Conquer is more obvious and similar to mathematical induction)
- Iterative algorithms usually have easier proofs of memory usage (and in Java use less memory)
- Recursive algorithms are sometimes easier to implement when we have complex data structures.
  - because the compiler maintains a stack of previous calls for us.
  - We will use recursive implementations for most operations over **trees**.

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.
- Recursive algorithms usually have easier proofs of correctness (Divide and Conquer is more obvious and similar to mathematical induction)
- Iterative algorithms usually have easier proofs of memory usage (and in Java use less memory)
- Recursive algorithms are sometimes easier to implement when we have complex data structures.
  - because the compiler maintains a stack of previous calls for us.
  - We will use recursive implementations for most operations over **trees**.
- We *could* have used recursive implementations for operations over lists & arrays but:
  - they are not simpler than the iterative implementations
  - Java will need $O(n)$ call stack space to execute them.

Homework 1: Implement Euclid's algorithm using

→ A recursive method.
→ An iterative method.

Homework 2: give recursive implementations for:

→ binary search over an array
→ linear search over a linked list

Homework 3: Implement **search** on a Binary Search Tree (next lecture) using recursion and compare with the iterative version in the book.

Homework 4**: give an iterative implementation for method **put** on binary search tree.