Recursion and Backtracking

02393 – Programming in C++

Teacher: Alberto Lluch Lafuente

Sebastian Mödersheim (slides author)

April 11, 2016

Lecture Plan

#	Date	Topic	Chapter *
1	1.2	Introduction	1
2	8.2	Basic C++	1
3	15.2	Data Types	2
4	22.2		
		Libraries and Interfaces	3
5	29.2		
6	7.3	Classes and Objects	4.1, 4.2 and 9.1, 9.2
7	14.3	Templates	4.1, 11.1
		Påskesferie	
8	4.4	Inheritance	14.3, 14.4, 14.5
9	11.4	Recursive Programming	5
10	18.4	Lists and Trees	10.5, 11, 13.1
11	25.4	Trees	13
12	2.5	Graphs	16
13	9.5	Summary	
	17.5	Exam	

* Recall that the book uses sometimes ad-hoc libraries that are slightly different with respect to the standard libraries (e.g. strings and vectors).

Recursion

Definition

Recursion (lat.): *see* Recursion
... or Google *recursion*.

What is Recursion?

- Solution technique that **solves large problems** by **reducing them to smaller problems** **of the same form**
- It is crucial that the smaller problem has the same form
- This means we can use the same technique for the big and the small problem!

Recursion

Definition

Recursion (lat.): *see* Recursion
... or Google *recursion*.

What is Recursion?

- Solution technique that **solves large problems** by **reducing them to smaller problems** **of the same form**
- It is crucial that the smaller problem has the same form
- This means we can use the same technique for the big and the small problem!

Why is Recursion... weird for some people?

- Some people are not used to inductive reasoning/abstraction...
- Other programming concepts are common in normal life:
  - repeat an action several times, on different objects (loops);
  - making decisions (if then else);
  - etc.

Recursion

When using recursion we must ensure:

- Every recursion step reduces to a smaller problem.
- There is a smallest problem (or a set of smallest problems) that can be handled directly, without recursion.
- Every chain of recursion steps eventually reaches one of these smallest problems.

Otherwise?

- Risk of non-termination!

Recursion

Recursive Leap of Faith

- When writing a recursive function, we believe that the recursive call computes the right solution, if the argument to the recursive call is smaller.
- Assuming that any recursive call works correctly is called the *Recursive Leap of Faith*.

Recusion

Rules of thumb

- Checking if you have a simple problem before decomposition.
- Solve the simple cases correctly!
- Check that decomposition makes the problem simpler!
- Ensure that decomposition eventually reaches one of the simple cases.
- The arguments to the recursive calls must be simpler versions of the original argument!
- When you take the recursive leap of faith, do the recursive calls provide with a correct solution to all simpler problems possible?

Examples

Mathematical definitions often use recusion:

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

Examples

Mathematical definitions often use recusion:

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

And can be easily transformed into a C++ program:

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

Induction Proofs

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

Note the similarity of recursion with induction proofs.

Example Theorem and Inductive Proof

$n! > 0$ for all natural numbers n.
Proof:

Induction Proofs

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

Note the similarity of recursion with induction proofs.

Example Theorem and Inductive Proof

$n! > 0$ for all natural numbers n.
Proof: trivial.

Induction Proofs

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

Note the similarity of recursion with induction proofs.

Example Theorem and Inductive Proof

$n! > 0$ for all natural numbers n.
Proof:

- Induction Base: For $n = 0$: $0! = 1$ by definition.

Induction Proofs

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

Note the similarity of recursion with induction proofs.

Example Theorem and Inductive Proof

$n! > 0$ for all natural numbers n.
Proof:

- Induction Base: For $n = 0$: $0! = 1$ by definition.
- Induction Step:
  - ($\star$) Suppose for some number $n - 1$ we have proved that $(n-1)! > 0$.
  - We show: then also $n! > 0$.

Induction Proofs

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

Note the similarity of recursion with induction proofs.

Example Theorem and Inductive Proof

$n! > 0$ for all natural numbers n.
Proof:

- Induction Base: For $n = 0$: $0! = 1$ by definition.
- Induction Step:
  - ($\star$) Suppose for some number $n - 1$ we have proved that $(n-1)! > 0$.
  - We show: then also $n! > 0$.
    - By definition $n! = n \cdot (n-1)!$.

Induction Proofs

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

Note the similarity of recursion with induction proofs.

Example Theorem and Inductive Proof

$n! > 0$ for all natural numbers n.
Proof:

- Induction Base: For $n = 0$: $0! = 1$ by definition.
- Induction Step:
  - ($\star$) Suppose for some number $n-1$ we have proved that $(n-1)! > 0$.
  - We show: then also $n! > 0$.
    - By definition $n! = n \cdot (n-1)!$.
    - We have that $n > 0$ (since $n - 1 \geq 0$) and $(n-1)! > 0$ (by $\star$). It then trivially follows that $n \cdot (n-1)! > 0$.

Induction Proofs

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

Note the similarity of recursion with induction proofs.

Example Theorem and Inductive Proof

$n! > 0$ for all natural numbers n.
Proof:

- Induction Base: For $n = 0$: $0! = 1$ by definition.
- Induction Step:
  - ($\star$) Suppose for some number $n - 1$ we have proved that $(n-1)! > 0$.
  - We show: then also $n! > 0$.
    - By definition $n! = n \cdot (n-1)!$.
    - We have that $n > 0$ (since $n - 1 \geq 0$) and $(n-1)! > 0$ (by $\star$). It then trivially follows that $n \cdot (n-1)! > 0$.
    - Thus $n! > 0$.

Induction proofs also work for recursive programs

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

Example Theorem and Inductive Proof

$fact(n) > 0$ for all natural numbers n.

Induction proofs also work for recursive programs

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

Example Theorem and Inductive Proof

$fact(n) > 0$ for all natural numbers n.

Proof:

Induction proofs also work for recursive programs

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

Example Theorem and Inductive Proof

$fact(n) > 0$ for all natural numbers n.
Proof:

- Induction Base: For $n == 0$: $fact(0) == 1$ by the program.

Induction proofs also work for recursive programs

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

Example Theorem and Inductive Proof

$fact(n) > 0$ for all natural numbers n.

Proof:

- Induction Base: For $n == 0$: $fact(0) == 1$ by the program.
- Induction Step:
  - ($\star$) Suppose for some number $n-1$ we have proved that $fact(n-1) > 0$.
  - We show: then also $fact(n) > 0$.

Induction proofs also work for recursive programs

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

Example Theorem and Inductive Proof

$fact(n) > 0$ for all natural numbers n.

Proof:

- Induction Base: For $n == 0$: $fact(0) == 1$ by the program.
- Induction Step:
  - ($\star$) Suppose for some number $n - 1$ we have proved that $fact(n - 1) > 0$.
  - We show: then also $fact(n) > 0$.
    - By the program $fact(n) == n \cdot fact(n - 1)$.

Induction proofs also work for recursive programs

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

Example Theorem and Inductive Proof

$fact(n) > 0$ for all natural numbers n.
Proof:

- Induction Base: For $n == 0$: $fact(0) == 1$ by the program.
- Induction Step:
  - ($\star$) Suppose for some number $n-1$ we have proved that $fact(n-1) > 0$.
  - We show: then also $fact(n) > 0$.
    - By the program $fact(n) == n \cdot fact(n-1)$.
    - We have that $n > 0$ (since $n-1 \geq 0$) and $fact(n-1) > 0$ (by $\star$). It then trivially follows that $n \cdot fact(n-1) > 0$.

Induction proofs also work for recursive programs

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

Example Theorem and Inductive Proof

$\mathit{fact}(n) > 0$ for all natural numbers n.

Proof:

- Induction Base: For $n == 0$: $\mathit{fact}(0) == 1$ by the program.
- Induction Step:
  - ($\star$) Suppose for some number $n-1$ we have proved that $\mathit{fact}(n-1) > 0$.
  - We show: then also $\mathit{fact}(n) > 0$.
    - By the program $\mathit{fact}(n) == n \cdot \mathit{fact}(n-1)$.
    - We have that $n > 0$ (since $n - 1 \geq 0$) and $\mathit{fact}(n-1) > 0$ (by $\star$). It then trivially follows that $n \cdot \mathit{fact}(n-1) > 0$.
    - Thus $n! > 0$.
    - Thus $\mathit{fact}(n) > 0$.

Examples/Live programming

- Toy examples: factorial, sum.
- Efficient search binary search
  - Naive search (linear search) of an element in a set takes $O(n)$.
  - Binary search is a divide-and-conquer $O(\log n)$ solution.
- Efficient sorting: merge sort
  - The recursion paradigm directly triggers an efficient solution!
  - Naive bubble sort: $O(n^2)$ for array of size n.
  - Merge sort: $O(n \log n)$ (theoretical optimum).
- Efficient exponentiation in cryptography ($a^n \ mod \ p$)
  - Naive exponentiation: $O(n)$
  - Efficient exponentiation: $O(\log n)$
  - Efficient solution is hard to program without recursion!

On Complexity

Resources needed by an algorithm:

- Time: number of operations
- Space: amount of memory/disk
- Depends on the size N of the problem.
- Usually working with asymptotic complexity, e.g. $O(N)$

Complexity of a Problem:

- Given a concrete problem (e.g. sorting a list of numbers)
- What is the best algorithm in terms of time and/or space?

Notes:

- Sometimes trade-off between time and space
- For many problems, the precise complexity is not known:
  - We have a best known algorithm (e.g. $O(2^N)$)
  - We can give a lower bound (e.g. $\Omega(N^5)$)
- Some problems are not computable at all.

Asymptotic Complexity

Big-O notation and Big-Ω notation

- Abstract from constant factors (and minor terms):

$$2N^2 + 17N + 53 \quad \Longrightarrow \quad O(N^2)$$

Asymptotic Complexity

Big-O notation and Big-Ω notation

- Abstract from constant factors (and minor terms):

$$2N^2 + 17N + 53 \quad \Longrightarrow \quad O(N^2)$$

- Intuition:
  - constant factors can be “made up” by faster hardware
  - the asymptotic complexity cannot

Asymptotic Complexity

Big-O notation and Big-Ω notation

- Abstract from constant factors (and minor terms):

$$2N^2 + 17N + 53 \quad \Longrightarrow \quad O(N^2)$$

- Intuition:
  - constant factors can be "made up" by faster hardware
  - the asymptotic complexity cannot
- Example: race between—
  1. slow hardware running a good algorithm, say time: $3000N$
  2. fast hardware running a bad algorithm, say time: $2N^2$

Asymptotic Complexity

Big-O notation and Big-Ω notation

- Abstract from constant factors (and minor terms):

$$2N^2 + 17N + 53 \quad \Longrightarrow \quad O(N^2)$$

- Intuition:
  - constant factors can be "made up" by faster hardware
  - the asymptotic complexity cannot
- Example: race between—
  1. slow hardware running a good algorithm, say time: $3000N$
  2. fast hardware running a bad algorithm, say time: $2N^2$

  (1.) wins for $N > 1500$.

Definition (Big-O Notation)

$O(f)$ is the class of functions that asymptotically grow no faster than f:

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ . \exists N_0 \in \mathbb{N} . \forall N \geq N_0 \, . \, g(N) \leq c \cdot f(N)\}$$

For instance:

$$2N^2 + 17N + 53 < 73N^2$$

so for $c = \frac{1}{73}$ and $N_0 = 1$, we can see

$$2N^2 + 17N + 53 \in O(N^2)$$

Dually, for giving lower-bounds on complexity, one uses $\Omega(f)$, which is the class of functions that grow at least as fast as f.