Lecture 15
Recursion
TDDD86: DALP

Print version of the lecture *Data structures, algorithms and programming paradigms*
1 november 2016

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

15.1

Content

Innehåll

15.2

1 Introduction

Recursion

- **Recursion**: definition of an operation in terms of itself
  - Solving a problem with recursion depends on solving less instances of the same problem

- **Recursive programming**: Writing functions that call themselves to solve a problem recursively
  - An equally powerful substitute for *iteration* (loops)
  - Particularly well suited for solving certain types of problems

15.3

Why we learn recursion?

- **"Cultural experience"**: Another way to think about problem solving
- **Powerful**: can solve certain problems better than iteration
- Leading to elegant, simplistic and short code (if used correctly)
- Many (functional languages like Scheme, ML and Haskell) programming languages use recursion exclusive (no loops)
- A key component of many of the remaining labs in the course

15.4

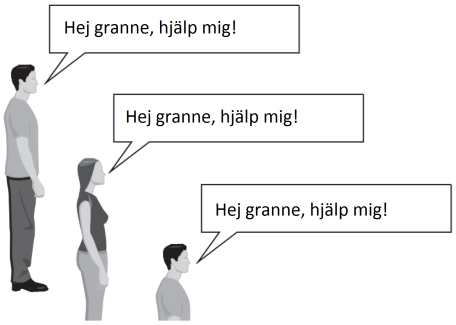
Exercise

- (For a student in the front row) How many students in total are sitting right behind you in your "column"?
  - You have poor eyesight so you can only see people right next to you. You cannot look back and count.
  - But you may ask questions to the people close to you.
  - How can we solve this problem (recursively).

15.5

The idea

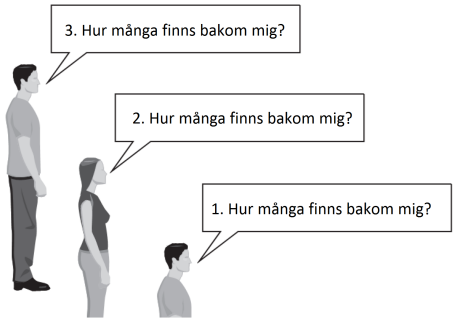
- Recursion is about breaking down a large problem into smaller instances of the same problem.
  - Each person can solve a small part of the problem.
    - What constitutes a smaller version of the problem that would be easier to answer?
    - What information from a neighbor could help me?



15.6

Recursive algoritm

- The number of people behind me:
  - If there is someone behind me, I ask him how many people are behind him.
    - When the person behind me answers with a value N I answer then with $N+1$
  - If nobody is sitting behind me, I simply answer 0.



15.7

Recursion and cases partitioning

- Every recursive algorithm involves at least two cases:
  - **base case**: an instance that can be solved directly.
  - **recursive case**: a more complicate instance of the problem that cannot be solved directly, but it can instead be described in terms of smaller instances of the same problem.
  - Some recursive algorithms have more than one base case and recursive cases, but all have at least one of each.
  - A key to recursive programming is the identification of such cases.

15.8

Another recursive task

- How can we remove exactly half of all M& M in a large bowl without pour out all of them and without being able to count them?
  - Would it help if more people help in solving the problem? Can each person make a small part of the work?
  - Is there any number of M&M that is easy to double without counting?
    - (What is the "base case"?)

15.9

2 Recursion in C++

Recursion i C++

- Consider the following function to print a line of stars

```
// Prints a line containing the given number of stars.
// Precondition: n >= 0
void printStars(int n) {
   for (int i = 0; i < n; i++) {
      cout << "*";
   }
   cout << endl; // end the line of output
}
```

- Write a recursive version of the function (which calls itself).
  - Solve the problem without using loops.
  - Tips: your solution should print only one star at a time.

15.10

Using recursion properly

- Condense the recursive case to a single case:

```
void printStars(int n) {
   if (n == 1) {
      // base case; just print one star
      cout << "*" << endl;
   } else {
      // recursive case; print one more star
      cout << "*";
      printStars(n - 1);
   }
}
```

15.11

"Recursion-zen"

- It is true, even easier, the base case is when n is 0, not 1:

```
void printStars(int n) {
   if (n == 0) {
      // base case; just end the line of output
      cout << endl;
   } else {
      // recursive case; print one more star
      cout << "*";
      printStars(n - 1);
   }
}
```

- **Recursion Zen**: The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.

15.12

Exercise - printBinary

- Write a recursive function `printBinary` which takes an integer and prints out its binary representation.
  - Example: `printBinary(7)` prints out `111`
  - Example: `printBinary(12)` prints out `1100`

plats	10	1
värde	**4**	**2**

32	16	8	4	2	1
1	**0**	**1**	**0**	**1**	**0**

  - Example: `printBinary(42)` prints out `101010`
  - Write the function recursively and without loops

15.13

Case analysis

- Recursion is about solving a small part of a big problem..
  - What will be 69,743 in base 2?
    - Do we know *something* about its representation in base 2?
  - Case analysis:
    - What is/are simple numbers to print in base 2?
    - Can we express a greater number in terms of (several) smaller numbers?

15.14

Finding pattern

- Suppose we examine some arbitrary integer N.
  - If the representation of N in base 2 is
  - Thus, the representation of $(N/2)$ is
  - and the representation of $(N\%2)$ is
    - What can we conclude from the fact?

10010101011
1001010101
1

15.15

Solution - printBinary

```
// Prints the given integer's binary representation.
// Precondition: n >= 0
void printBinary(int n) {
   if (n < 2) {
      // base case; same as base 10
      cout << n;
   } else {
      // recursive case; break number apart
      printBinary(n / 2);
      printBinary(n % 2);
   }
}
```

15.16

Exercise - reverseLines

- Write a recursive function `reverseLines` which takes a file stream and prints the lines of the file in reverse.

Exempelindatafil:

```
Roses are red,
Violets are blue.
All my base
Are belong to you.
```

→

Förväntat utdata:

```
Are belong to you.
All my base
Violets are blue.
Roses are red,
```

  - Which cases we have to consider?
    - How can we solve a small part of the problem at a time?
    - How looks a file that is easy to be reversed?

15.17

Pseudocode for reversing

- Reverse lines of a file:
  - Read a line R from the file.
  - Print out the rest of lines in reverse order.
  - Print out R.
- If we have only one way to reverse the rest of the lines in the file...

15.18

Solution - reverseLines

```
void reverseLines(ifstream& input) {
   string line;
   if (getline(input, line)) {
      // recursive case
      reverseLines(input);
      cout << line << endl;
   }
}
```

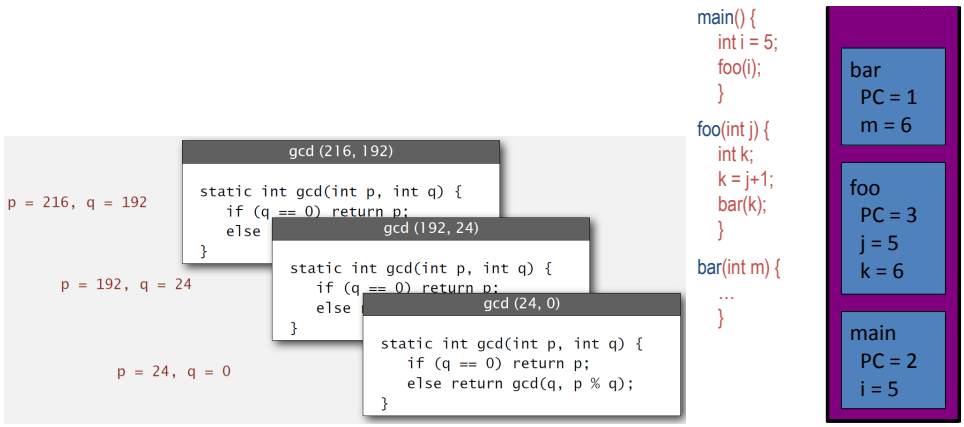
- What is the base case?

15.19

2.1 Implementation of recursion

Remember: use of stack – function call

- To implement the compiler features
  - Function call: *push:a* the local environment and return address
  - Return: *pop:a* return address and the local environment
  - This allows for recursion.



15.20

2.2 Tail call

Tail call recursion

A recursive call is *tail recursive* iff the first instruction to control the flow coming after the call is **return**.

- stack is not needed: everything on the stack can be thrown directly
- Tail recursive functions can be rewritten as iterative functions

The recursive call in FACT is *not* tail recursive:

function FACT(n)
  if $n = 0$ **then return** 1
  else return $n \cdot$FACT($n-1$)

The first instruction after the return from the recursive call is *multiplication* $\Rightarrow n$ *must be maintained on the stack*

15.21

A tail recursive function

function BINSEARCH($v[a,\dots,b],x$)
  if $a < b$ **then**
    $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
    if $v[m].key < x$ **then**
      return BINSEARCH($v[m+1,\dots,b],x$)
    else return BINSEARCH($v[a,\dots,m],x$)
  if $v[a].key = x$ **then return** a
  else return 'not found'

The two calls are *tail recursive*. 15.22

Eliminating the tail recursion

The two recursive calls can be eliminated:

1: **function** BINSEARCH($v[a,\dots,b],x$)
2: **if** $a < b$ **then**
3: $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
4: **if** $v[m].key < x$ **then**
5: $a \leftarrow m+1$ {var: **return** BINSEARCH($v[m+1,\dots,b],x$)}
6: **else** $b \leftarrow m$ {var: **return** BINSEARCH($v[a,\dots,m],x$)}
7: **goto** *(2)*
8: **if** $v[a].key = x$ **then return** a
9: **else return 'not found'**

15.23

Tail recursive factorial function

$fact$ can be rewritten by introducing an auxiliary function:

function FACT(n)
  return FACT2($n,1$)

function FACT2(n,f)
  if $n = 0$ **then return** f
  else return FACT2($n-1, n \cdot f$)

FACT2 is *tail recursive* $\Rightarrow$ *the memory usage for recursion elimination becomes* $O(1)$ 15.24

2.3 Exercise

Exercise - pow

- Write a recursive function `pow` which takes an integer and an exponent and returns the integer raised to exponent.
  - Example: `pow(3, 4)` returns 81
  - Solve this problem recursively and without using loops

15.25

Solution - pow

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
int pow(int base, int exponent) {
   if (exponent == 0) {
      // base case; any number to 0th power is 1
      return 1;
   } else {
      // recursive case: x^y = x * x^(y-1)
      return base * pow(base, exponent - 1);
   }
}
```

15.26

An optimization?

- Note the following mathematical property:

$3^{12} = 531441 = 9^6$

$= (3^2)^6$

$531441 = (9^2)^3$

$= ((3^2)^2)^3$

  - When does this "trick" works?
  - How can we take advantage of this optimization for our code?
  - Why cares about tricks when the code already works?

15.27

Solution 2 - pow

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
int pow(int base, int exponent) {
   if (exponent == 0) {
      // base case; any number to 0th power is 1
      return 1;
   } else if (exponent % 2 == 0) {
      // recursive case 1: x^y = (x^2)^(y/2)
      return pow(base * base, exponent / 2);
   } else {
      // recursive case 2: x^y = x * x^(y-1)
      return base * pow(base, exponent - 1);
   }
}
```

15.28

3 Algorithm analysis

Analysis of Algorithms

What do we analyze?

- Correctness (not in this course)
- Termination (not in this course)
- Efficiency, resources, complexity

time complexity — how long does the algorithm in the worst case?

- as a function of what?
- what is the time step?

Memory Complexity — how much memory does the algorithm need in the worst case?

- as a function of what?
- measured in what?
- remember that function and procedure calls also use memory

15.29

How to compare the effectiveness

- Study the execution time (or memory usage) as a function of the size of the input data..
- When are two algorithms have the same "effectiveness"?
- When is an algorithm better than another?

Comparison with some elementary functions

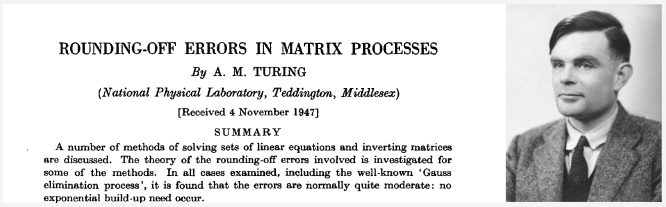
n	$\log_2 n$	n	$n\log_2 n$	n^2	2^n
2	1	2	2	4	4
16	4	16	64	256	$6.5 \cdot 10^4$
64	6	64	384	4096	$1.84 \cdot 10^{19}$

$1.84 \cdot 10^{19} \mu$seconds $= 2.14 \cdot 10^8$ days $= 583.5$ millennia

15.30

Simplify the calculations

> *" It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. "* — Alan Turing



ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

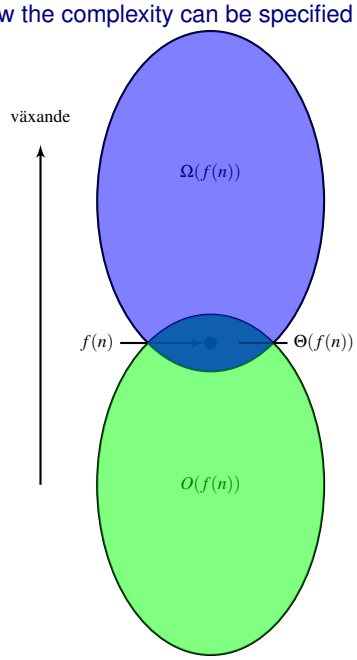
[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

15.31

How the complexity can be specified



- How is the complexity of the growing size of n on the input?
- Asymptotic complexity — what happens when n grows to infinity?
- a lot easier if we ignore the constant factors.

- $O(f(n))$ – grows high as quickly as $f(n)$
- $\Omega(f(n))$ – grows at least as fast as $f(n)$
- $\Theta(f(n))$ – grows as fast as $f(n)$

15.32

Ordo-notation

f, g: increasing functions from $\mathbb{N}$ to $\mathbb{R}^+$

- $f \in O(g)$ iff there exist $c > 0, n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ Intuition: Aside from the constant factors, the growing of f is not faster than g.
- $f \in \Omega(g)$ iff there exist $c > 0, n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$ Intuition: Aside from the constant factors, the growing of f is at least as fast as g.
- $f(n) \in \Theta(g(n))$ iff $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$ Intuition: Aside from the constant factors, f and g grow with the same rate.

NOTE: Ω is the opposite of O, i.e. $f \in \Omega(g)$ iff $g \in O(f)$.

15.33

3.1 Recursive algorithms

Execution time for recursive algorithms

- Characterize the execution time with a recurrent relationship
- Find a solution (in closed form) to the recurrent relations
- If you do not recognize the recurrent relation, you can
  - "Roll up" the relationship a few times to get forward a hypothesis about a possible solution: $T(n) = \ldots$
  - Prove the hypothesis about $T(n)$ with mathematical induction. If it does not go well, modify the hypothesis and try again ...

15.34

Example: the factorial function

function FACT(n)
  if $n = 0$ **then return** 1
  else return n·FACT($n-1$)

Execution time:

- time for comparison: t_c
- time for multiplication: t_m
- time for call and return is neglected

Total execution time $T(n)$. $T(0) = t_c$ $T(n) = t_c + t_m + T(n-1)$, if $n > 1$ Thus, for $n > 0$:

$$T(n) = (t_c + t_m) + (t_c + t_m) + T(n-2) =$$

$$= (t_c + t_m) + (t_c + t_m) + (t_c + t_m) + T(n-3) = \ldots =$$

$$= \underbrace{(t_c + t_m) + \ldots + (t_c + t_m)}_{n \text{ ggr}} + t_c = n \cdot (t_c + t_m) + t_c \in O(n)$$

15.35

Example: Binary search

function BINSEARCH($v[a,\ldots,b], x$)
  if $a < b$ **then**
    $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
    if $v[m].key < x$ **then**
      return BINSEARCH($v[m+1,\ldots,b], x$)
    else return BINSEARCH($v[a,\ldots,m], x$)
  if $v[a].key = x$ **then return** a
  else return 'not found'

Let $T(n)$ be the time, in the worst case, searching among n numbers with BINSEARCH.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T\left(\lceil \frac{n}{2} \rceil\right) + \Theta(1) & \text{if } n > 1 \end{cases}$$

If $n = 2^m$ we get

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) \text{ if } n > 1 \end{cases}$$

We can then conclude that $T(n) = \Theta(\log n)$.

15.36

Master method

Sats 1 ("Master theorem")**.** If $a \geq 1, b > 1, d > 0$ so the recurrent relation has

$$\begin{cases} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ T(1) &= d \end{cases}$$

the asymptotic solution

- $T(n) = \Theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a - \varepsilon})$ for any $\varepsilon > 0$
- $T(n) = \Theta(n^{\log_b a} \log n)$ if $f(n) = \Theta(n^{\log_b a})$
- $T(n) = \Theta(f(n))$ if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for any $\varepsilon > 0$ and $af\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for any constant $c < 1$ for all sufficiently large n.

15.37

3.2 Typical growth rates

Typical growth rates

growth rate	typical code	description	example	$T(2n)/T(n)$
1	`a = b + c`	sats	put together two numbers	1
$\log_2 n$	`while (n > 1) { n = n / 2; ...}`	share half	binary search	≈ 1
n	`for (int i = 0; i < n, i++) { ... }`	loop	find maximum	2
$n \log_2 n$	see next lecture about mergesort	divide and conquer	mergesort	≈ 2
n^2	`for (int i = 0; i < n, i++)` `for (int j = 0; j < n, j++)` `{ ... }`	double loop	check all pairs	4
n^3	`for (int i = 0; i < n, i++)` `for (int j = 0; j < n, j++)` `for (int k = 0; k < n, k++)` `{ ... }`	triple-loop	Check all triples	8
2^n	see next lecture	total-search	check all subsets	$T(n)$

15.38