Föreläsning 13

Recursion

TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
04 November 2024

IDA, Linköpings universitet

13.1

Content

Contents

1 Introduction **1**

2 Recursion in C++ **2**
2.1 Implementation av recursion 4
2.2 Tail recursion 4
2.3 One more exercise 5

3 Algorithm analysis **6**
3.1 Analysis of algorithms 6
3.2 Recursive algorithms 7
3.3 Common growth rates 8

13.2

1 Introduction

Recursion

- **recursion**: Defining an operation in terms of itself
  - To solve a problem recursively requires solving smaller instances of the same problem

- **recursive programming**: Write functions that call themselves to solve problems recursively
  - As powerful as *iteration* (loops)
  - Particularly suitable for certain types of problems

13.3

Why learn recursion??

- **"Cultural experience"**: Another way to think about problem solving.
- **powerful**: can solve certain types of problems better than iteration
- Can result in elegant, simple and short code (if used correctly)
- Many (functional languages such as Scheme, ML and Haskell) programming languages use recursion exclusively (no loops)
- A key component in many of the remaining labs in the course

13.4

Recursion and case analysis

- Any recursive algorithm involves at least two cases:
  - **base case**: A simple instance of the problem that can be solved directly.
  - **recursive case**: A more complex instance of the problem for which the solution can be described in terms of solutions to smaller instances of the same problem..
  - Some recursive algorithms have more than one base case. All have at least one.
  - Key to recursive programming is to identify these cases.

13.5

2 Recursion in C++

Recursion i C++

- Consider the following function to write 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 (it should call itself).
  - Solve the problem without using loops.
  - Tips: Your solution should write a single star at a time.

13.6

Use recursion correctly

- Condense recursive cases to one 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);
   }
}
```

13.7

"Recursion-zen"

- The actual, simpler, 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);
   }
}
```

13.8

Exercise - printBinary

- Write a recursive function `printBinary` that takes a natural number and that writes it in base 2 (binary)
  - Example: `printBinary(7)` prints `111`
  - Example: `printBinary(12)` prints `1100`
  - Example: `printBinary(42)` prints `101010`

plats	10	1
värde	**4**	**2**

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

  - Write a recursive function without loops

13.9

Case analysis

- Recursion is about solving parts of a larger problem
  - what is 69743 in base 2?
    - what do we know about its representation in base 2?
  - Case analysis:
    - Which numbers are simple to write in base 2?
    - Can we express a larger number in terms of (some) smaller one(s)?

13.10

Find the pattern

- Assume an arbitrary number N.
  - If the representation of N in base 2 is
  - Then the representation of $(N/2)$
  - and the representation of $(N\%2)$ is
    - What can we deduce?

```
10010101011
1001010101
          1
```

13.11

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);
   }
}
```

13.12

Exercise - reverseLines

- Write a recursive function `reverseLines` that takes a file stream as input and that prints the lines in reverse order

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 should be considered?
    - How can we solve part of the problem at a time?
    - What would be a file that is easy to reverse?

13.13

Pseudocode for reversing

- Reverse lines in a file:
  - Read a line L from the file
  - Print the rest of the lines in reverse order.
  - Print the line L
- If we only could reverse the or the lines in the file...

13.14

Solution - reverseLines

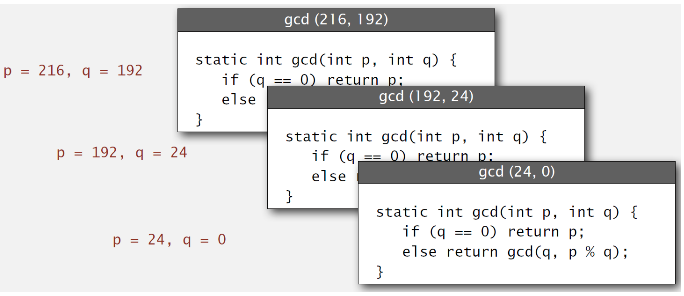
```
void reverseLines(ifstream& input) {
   string line;
   if (getline(input, line)) {
      // recursive case
      reverseLines(input);
      cout << line << endl;
   }
}
```

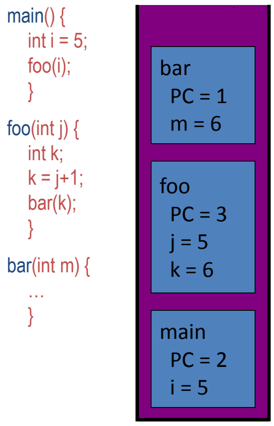
- What is the base case? 13.15

2.1 Implementation av recursion

Recall: stacks and function calls

- Compiler implement functions:
  - Function calls: *push:a* local context and return address
  - Return: *pop:a* return address and local context
  - This enables recursion.





13.16

2.2 Tail recursion

Tail recursion

A recursive call is *tail recursive* iff the first instruction after the control gets back after the call is a **return**.

- The stack is not needed
- Tail recursive functions can be rewritten into 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$(n-1)$

First instruction after the return from the recursive call is a *multiplication* $\Rightarrow$ *n needs to b kept on the stack* 13.17

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 recursive calls are *tail recursive*. 13.18

Eliminating tail recursion

The two tail recursive calls can be eliminated:

```
1: function BINSEARCH(v[a,...,b],x)
2:     if a < b then
3:         m ← ⌊(a+b)/2⌋
4:         if v[m].key < x then
5:             a ← m+1 {was: return BINSEARCH(v[m+1,...,b],x)}
6:         else b ← m {was: return BINSEARCH(v[a,...,m],x)}
7:         goto (2)
8:     if v[a].key = x then return a
9:     else return 'not found'
```

13.19

Tail recursive factorial

fact can be rewritten by using a help 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* $\Rightarrow$ *memory usage after eliminating the recursive in* $O(1)$

13.20

2.3 One more exercise

Exercise - pow

- Write a recursive function `pow` that takes two natural numbers as arguments: a base and an exponent and that returns the base to the power of the exponent.
  - Example: `pow(3, 4)` returns 81
  - Solve the problem recursively without loops

13.21

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);
   }
}
```

13.22

An optimization?

- Observe the following mathematical properties:

  $3^{12} = 531441 = 9^6 = (3^2)^6$

  $531441 = (9^2)^3 = ((3^2)^2)^3$

  - When does this work?
  - How can we leverage on it?
  - Why use it when the code already works?

13.23

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);
   }
}
```

13.24

3 Algorithm analysis

3.1 Analysis of algorithms

Analysis of algorithms

What is analysis?

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

Time complexity — how long it takes an algorithm in the worst case?

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

Memory complexity — how much memory is required?

- as a function of what?
- how is it measured?
- remember that code and function calls also takes memory

13.25

How can you compare different effectiveness

- Study execution time (or memory consumption) in function of the size of input data.
- When can we say that two algorithms have ”similar effectiveness”?
- When can we say that an algorithm is better than an other?

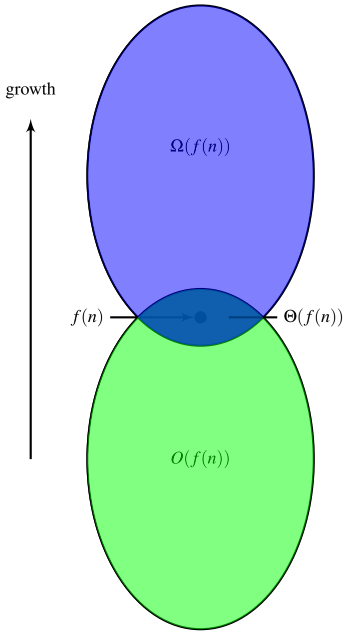
Comparison between some elementary functions

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	$1.84 \cdot 10^{19}$

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

13.26

How complexity can be specified?



- How does the complexity grow with the size n of input data?
- Asymptotic complexity — what happens when n grows to infinity?
- Much easier if we ignore constant factors

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

13.27

Ordo-notation

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

- $f \in O(g)$ if and only if it exists $c > 0, n_0 > 0$ such as $f(n) \le c \cdot g(n)$ for all $n \ge n_0$ Intuition: ignoring the constant factor, f does not grow faster than g
- $f \in \Omega(g)$ if and only if it exists $c > 0, n_0 > 0$ such as $f(n) \ge c \cdot g(n)$ for all $n \ge n_0$ Intuition: ignoring the constant factor, f grows at least as fast as g
- $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$ Intuition: ignoring the constant factor, f and g have similar growth

Note: Ω is the opposite of O, i.e. $f \in \Omega(g)$ if and only if $g \in O(f)$. 13.28

3.2 Recursive algorithms

Execution time for recursive algorithms

- Characterize execution time with a recursive relation
- Find a solution in closed form the recursive relation
- If you do not recognize the recursive relation, you can
  - "Unroll" the relation a number of times to formulate a hypothesis for a possible solution of the form $T(n) = \ldots$
  - Prove the hypothesis about $T(n)$ by induction. If it does not work, modify the hypothesis and try again...

13.29

Example: Factorial function

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

Execution time:

- time for comparison: t_c
- time for multiplication: t_m
- time for calls and returns: t_r

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

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

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

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

13.30

Example: Binary search

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

Let $T(n)$ be the time, in the worst case, to search 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 $T(n) = \Theta(\log n)$.

13.31

Master theorem

Sats 1 ("Master theorem"). Assume $a \geq 1, b > 1, d > 0$. The recursive relation

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

has the following asymptotic solution

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

Examples:

- $T(n) = 9\,T(n/3) + n$
- $T(n) = T(2n/3) + 1$
- $T(n) = 3\,T(n/4) + n \log n$

13.32

3.3 Common growth rates

Common growth rates

Growth	typical code	description	example	$T(2n)/T(n)$
1	`a = b + c`	instruction	add two numbers	1
$\log_2 n$	`while (n > 1) { n = n / 2; ...}`	divide in halves	binary search	≈ 1
n	`for (int i = 0; i < n, i++) { ... }`	loop	find maximum	2
$n \log_2 n$	see lecture on 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	$T(n)$

13.33