Conditions

using conditional (boolean) expressions

Syntax (and semantics):

- `if` condition `then` command;
- `if` condition `then` `begin` block of statements `end`;
- `if` condition `then` command else command;
  Attention! Before `else` we do *not* place a semicolon!
- `if` condition `then` `begin` block `end` `else` `begin` block `end`;

Příklad:

```
if temperature>25 then
        writeln('Let us go to a pub!');
```

Example

```
if temperature>25 then writeln('Let us go to a
pub!');
```

```
if temperature>25 then
begin
        writeln('Let''s go to a pub!');
end
else
begin
        writeln('Let''s stay at home!');
end;
```

Cycles

- `while` condition `do` command or block;
  Repeat, while `condition` is satisfied (fulfilled).
- `for i:=1 to 10 do` command or block;
  Repeat for each value of the variable starting by the former bound up to the latter one.
- `for i:=100 downto 1 do` command or block;
- `repeat` commands; `until condition;`
  Repeat while the `condition` is **unsatisfied!**

Example:

```
program binary;
var a:integer;
begin
        readln(a);
        while a > 0 do
        begin
                if a mod 2 = 1 then
                        write(1)
                else    write(0);
                a:=a div 2;
        end;
end.
```

Example improved

While programming, it is principal to think on it. Otherwise we tend to perform an **unnecessary operations!**

```
program binary;
var a:integer;
begin
        readln(a);
        while a > 0 do
        begin
                write(a mod 2);
                a:=a div 2;
        end;
end.
```

Example, factorization:

```
program factor;
var a,i:integer;
begin
        i:=2;
        readln(a);
        while i <= a do
        begin   if (a div i)*i = a then
                begin
                        write(i);
                        a:=a div i;
                end
                else    i:=i+1;
        end;
end.
```

Example, the factorization improved:

```
program factor;
var a,i:integer;    repeating:boolean;
begin   i:=2;   repeating:=false;
        readln(a);
        while i <= a do
        begin   if (a div i)*i = a then
                begin   if repeating then
                                write('*')
                        else   repeating:=true;
                        write(i);
                        a:=a div i;
                end  else  i:=i+1;
        end;
end.
```

For algorithms we analyze several types of complexities:

- Static – saying how long a program is (how many characters has a source-code or binary executable file),

For algorithms we analyze several types of complexities:

- Static – saying how long a program is (how many characters has a source-code or binary executable file),
- dynamic – how long does the algorithm run.

For algorithms we analyze several types of complexities:

- Static – saying how long a program is (how many characters has a source-code or binary executable file),
- dynamic – how long does the algorithm run.
- By default we explore the dynamic complexity.

For algorithms we analyze several types of complexities:

- Static – saying how long a program is (how many characters has a source-code or binary executable file),
- dynamic – how long does the algorithm run.
- By default we explore the dynamic complexity.

Definition

Let n denote the length of the input (for an algorithm $\mathcal{A}$). The (dynamic, time, worst-case-) *complexity* of $\mathcal{A}$ is the smallest function f such that for all n, the value $f(n)$ is at least the number of elementary steps performed by algorithm $\mathcal{A}$ for any input of length n.

Examples I

Definition

Let n denote the length of the input (for an algorithm $\mathcal{A}$). The (dynamic, time, worst-case-) *complexity* of $\mathcal{A}$ is the smallest function f such that for all n, the value $f(n)$ is at least the number of elementary steps performed by algorithm $\mathcal{A}$ for any input of length n.

- Sieve of Eratosthenes: For each prime at most linear (w. r. t. array-length), i.e., altogether at most quadratic.

Examples I

Definition

Let n denote the length of the input (for an algorithm $\mathcal{A}$). The (dynamic, time, worst-case-) *complexity* of $\mathcal{A}$ is the smallest function f such that for all n, the value $f(n)$ is at least the number of elementary steps performed by algorithm $\mathcal{A}$ for any input of length n.

- Sieve of Eratosthenes: For each prime at most linear (w. r. t. array-length), i.e., altogether at most quadratic.
- Number-factorization: linear w. r. t. value of the factorized number.

Examples I

Definition

Let n denote the length of the input (for an algorithm $\mathcal{A}$). The (dynamic, time, worst-case-) *complexity* of $\mathcal{A}$ is the smallest function f such that for all n, the value $f(n)$ is at least the number of elementary steps performed by algorithm $\mathcal{A}$ for any input of length n.

- Sieve of Eratosthenes: For each prime at most linear (w. r. t. array-length), i.e., altogether at most quadratic.
- Number-factorization: linear w. r. t. value of the factorized number.
- Attention! We are measuring the complexity in terms of input-length!!

Examples II

Definition

Let n denote the length of the input (for an algorithm $\mathcal{A}$). The (dynamic, time, worst-case-) *complexity* of $\mathcal{A}$ is the smallest function f such that for all n, the value $f(n)$ is at least the number of elementary steps performed by algorithm $\mathcal{A}$ for any input of length n.

- Minotaurus in the Labyrinth: Linear in the number of corridors (edges).

Examples II

Definition

Let n denote the length of the input (for an algorithm $\mathcal{A}$). The (dynamic, time, worst-case-) *complexity* of $\mathcal{A}$ is the smallest function f such that for all n, the value $f(n)$ is at least the number of elementary steps performed by algorithm $\mathcal{A}$ for any input of length n.

- Minotaurus in the Labyrinth: Linear in the number of corridors (edges).
- Stable matching: At most quadratic w. r. t. number of ladies (gentlemen).

Asymptotic analysis

- It is dubious what means an *elementary step.* Moreover, not in all CPUs the elementary step would be defined in the same way. Thus we introduce following abstraction (independent on multiplicative constant):

Asymptotic analysis

- It is dubious what means an *elementary step.* Moreover, not in all CPUs the elementary step would be defined in the same way. Thus we introduce following abstraction (independent on multiplicative constant):
- For functions f, g, we say that $f \in O(g)$, if $\exists_{c,n_0}$ s. t. $\forall_{n>n_0} f(n) \leq cg(n)$,

Asymptotic analysis

- It is dubious what means an *elementary step.* Moreover, not in all CPUs the elementary step would be defined in the same way. Thus we introduce following abstraction (independent on multiplicative constant):
- For functions f, g, we say that $f \in O(g)$, if $\exists_{c,n_0}$ s. t. $\forall_{n>n_0} f(n) \leq cg(n)$,
- $f \in \Omega(g)$, if $\exists_{c>0,n_0}$ s. t. $\forall_{n>n_0} f(n) \geq cg(n)$.,

Asymptotic analysis

- It is dubious what means an *elementary step.* Moreover, not in all CPUs the elementary step would be defined in the same way. Thus we introduce following abstraction (independent on multiplicative constant):
- For functions f, g, we say that $f \in O(g)$, if $\exists_{c,n_0}$ s. t. $\forall_{n>n_0} f(n) \leq cg(n)$,
- $f \in \Omega(g)$, if $\exists_{c>0,n_0}$ s. t. $\forall_{n>n_0} f(n) \geq cg(n)$.,
- $f \in \Theta(g)$, if $f \in O(g)$ and simultaneously $f \in \Omega(g)$.

Examples

- Is $n \in O(n^2)$?

Examples

- Is $n \in O(n^2)$?
- Is $n^2 \in O(n)$?

Examples

- Is $n \in O(n^2)$?
- Is $n^2 \in O(n)$?
- Is $3n^5 + 2n^3 + 1000 \in \Theta(n^5)$?

Examples

- Is $n \in O(n^2)$?
- Is $n^2 \in O(n)$?
- Is $3n^5 + 2n^3 + 1000 \in \Theta(n^5)$?
- Is $n^{1000} \in O(2^n)$?

Examples

- Is $n \in O(n^2)$?
- Is $n^2 \in O(n)$?
- Is $3n^5 + 2n^3 + 1000 \in \Theta(n^5)$?
- Is $n^{1000} \in O(2^n)$?
- Is $2^n \in O(n^{2000})$?

Examples

- Is $n \in O(n^2)$?
- Is $n^2 \in O(n)$?
- Is $3n^5 + 2n^3 + 1000 \in \Theta(n^5)$?
- Is $n^{1000} \in O(2^n)$?
- Is $2^n \in O(n^{2000})$?
- Example with cards showing how quickly the exponential function grows.

Further notions

related to the computational complexity

- Best-case complexity,

Further notions

related to the computational complexity

- Best-case complexity,
- average-case complexity – average number of steps for input-instances of a given length,

Further notions

related to the computational complexity

- Best-case complexity,
- average-case complexity – average number of steps for input-instances of a given length,
- amortized complexity – average number of steps for (potentially) infinite sequence of operations – we consider the worst possible sequence,

Further notions

related to the computational complexity

- Best-case complexity,
- average-case complexity – average number of steps for input-instances of a given length,
- amortized complexity – average number of steps for (potentially) infinite sequence of operations – we consider the worst possible sequence,
- complexity of a problem – complexity of the best possible algorithm (solving a given problem).