Arrays

- ... when we need to store many elements of the same number (e.g., 1 000 of integer numbers),
- we define in the section of variables (i.e., `var`)),
- gets defined using keyword `array` followed by an interval defining its bounds and underlying data-type.
- Example:
```
var a:  array [1..100] of integer;
        file_example:array[5..50] of string;
```
- Individual members get accessed using square brackets:
  Example:
```
a[1]:=10;
file_example[6]:='xxx';
{Beware:} file_example[1]:='out of bounds!';
```

Sieve of Eratosthenes

```
var primes: array[2..1000] of boolean;        i,j:integer;
begin
for i:=2 to 1000 do primes[i]:=true;
for i:=2 to 1000 do
begin
    if primes[i] then
    begin writeln(i,' is a prime');
        j:=2;
        while(i*j<=1000) do
        begin
            primes[i*j]:=false;
            j:=j+1;
        end;
    end;
end
```

Searching in an array

- Unsorted array $\Rightarrow$ simple upper and lower bound (pass through the whole array until found),

Searching in an array

- Unsorted array $\Rightarrow$ simple upper and lower bound (pass through the whole array until found),
- sorted array:

Searching in an array

- Unsorted array $\Rightarrow$ simple upper and lower bound (pass through the whole array until found),
- sorted array:
  - unary search (browse through the array like through a book),

Searching in an array

- Unsorted array $\Rightarrow$ simple upper and lower bound (pass through the whole array until found),
- sorted array:
  - unary search (browse through the array like through a book),
  - binary search (start in the middle, in each step halve the input),

Searching in an array

- Unsorted array $\Rightarrow$ simple upper and lower bound (pass through the whole array until found),
- sorted array:
    - unary search (browse through the array like through a book),
    - binary search (start in the middle, in each step halve the input),
    - quadratic search, generalized quadratic search...

Unary search

- Simple algorithm, simple analysis, its complexity:

Unary search

- Simple algorithm, simple analysis, its complexity:
- $\Theta(n)$.

Binary search

- What's the complexity of the algorithm? When we have to add an extra step?

Binary search

- What's the complexity of the algorithm? When we have to add an extra step?
- $\Theta(\log n)$.

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:
- Naive algorithm – Easily implementable, simple complexity-analysis.

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:
- Naive algorithm – Easily implementable, simple complexity-analysis.
- Strassen's algorithm – hard to implement, hard to analyze, hard to understand, but it has a better complexity.

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:
- Naive algorithm – Easily implementable, simple complexity-analysis.
- Strassen's algorithm – hard to implement, hard to analyze, hard to understand, but it has a better complexity.
- Coppersmith-Vinograd's algorithm – yet even more complicated with yet better complexity.

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:
- Naive algorithm – Easily implementable, simple complexity-analysis.
- Strassen's algorithm – hard to implement, hard to analyze, hard to understand, but it has a better complexity.
- Coppersmith-Vinograd's algorithm – yet even more complicated with yet better complexity.

- Finding largest zero-submatrix:

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:
- Naive algorithm – Easily implementable, simple complexity-analysis.
- Strassen's algorithm – hard to implement, hard to analyze, hard to understand, but it has a better complexity.
- Coppersmith-Vinograd's algorithm – yet even more complicated with yet better complexity.

- Finding largest zero-submatrix:
- Naive algorithm: $O(n^6)$

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:
- Naive algorithm – Easily implementable, simple complexity-analysis.
- Strassen's algorithm – hard to implement, hard to analyze, hard to understand, but it has a better complexity.
- Coppersmith-Vinograd's algorithm – yet even more complicated with yet better complexity.

- Finding largest zero-submatrix:
- Naive algorithm: $O(n^6)$
- Any ideas how to beat this complexity?

Further examples

of array-operating algorithms and the complexity-analysis:

- Matrix-multiplication:
- Naive algorithm – Easily implementable, simple complexity-analysis.
- Strassen's algorithm – hard to implement, hard to analyze, hard to understand, but it has a better complexity.
- Coppersmith-Vinograd's algorithm – yet even more complicated with yet better complexity.

- Finding largest zero-submatrix:
- Naive algorithm: $O(n^6)$
- Any ideas how to beat this complexity?
- Exercise (think about it at home, solution appears later).

Horner's Method

- We want to convert a number stored as string into an integer.

Number $a_n a_{n-1} a_{n-2} \dots a_0$ in decimal (position) system means: $a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0$. It holds:

$$a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0 = (\dots((a_n * 10) + a_{n-1} * 10) + \dots + a_1) * 10 + a_0$$

In the same way we may evaluate numbers in other position systems (binary, ternary, quaternary, decimal, hexadecimal...).

Horner's Method

- We want to convert a number stored as string into an integer.
- Naive approach: We may start from the least important digit, keep track of an exponent by 10 and sum up.

Number $a_n a_{n-1} a_{n-2} \dots a_0$ in decimal (position) system means: $a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0$. It holds:

$$a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0 = (\dots((a_n * 10) + a_{n-1} * 10) + \dots + a_1) * 10 + a_0$$

In the same way we may evaluate numbers in other position systems (binary, ternary, quaternary, decimal, hexadecimal...).

Horner's Method

- We want to convert a number stored as string into an integer.
- Naive approach: We may start from the least important digit, keep track of an exponent by 10 and sum up.
- ... or we use Horner's method and start with the most important digit.

Number $a_n a_{n-1} a_{n-2} \dots a_0$ in decimal (position) system means: $a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0$. It holds:

$$a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0 = (\dots((a_n * 10) + a_{n-1} * 10) + \dots + a_1) * 10 + a_0$$

In the same way we may evaluate numbers in other position systems (binary, ternary, quaternary, decimal, hexadecimal...).

Horner's Method

- We want to convert a number stored as string into an integer.
- Naive approach: We may start from the least important digit, keep track of an exponent by 10 and sum up.
- ... or we use Horner's method and start with the most important digit.
- We find its value and proceed (inductively):
  Multiply so far obtained result by 10 and add (sum up with) the newly loaded digit.

Number $a_n a_{n-1} a_{n-2} \dots a_0$ in decimal (position) system means: $a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0$. It holds:

$$a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_0 = (\dots((a_n * 10) + a_{n-1} * 10) + \dots + a_1) * 10 + a_0$$

In the same way we may evaluate numbers in other position systems (binary, ternary, quaternary, decimal, hexadecimal...).

Example

```
program x;
var a:string;
    i,value:longint;
begin
     readln(a); i:=1; value:=0;
     while i<=length(a) do
     begin
          value:=10*value+ord(a[i])-ord('0');
          i:=i+1;
     end;
     writeln(value);
end.
```

Evaluating a polynomial

- Consider a polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$.

Evaluating a polynomial

- Consider a polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$.
- We want to evaluate it, i.e., find its value for some value of x.

Evaluating a polynomial

- Consider a polynomial $a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$.
- We want to evaluate it, i.e., find its value for some value of x.
- Possibilities?

Evaluating a polynomial

- Consider a polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$.
- We want to evaluate it, i.e., find its value for some value of x.
- Possibilities?
- Brute force (estimate $a_n x^n$, $a_{n-1} x^{n-1}$,... and sum it up)

Evaluating a polynomial

- Consider a polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$.
- We want to evaluate it, i.e., find its value for some value of x.
- Possibilities?
- Brute force (estimate $a_n x^n$, $a_{n-1} x^{n-1}$,... and sum it up)
- or Horner's method:

$$\sum_{i=0}^{n} a_i x^i = ((\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0).$$

Evaluating a polynomial by Horner's method

- 1: Read the coefficient of highest (so far not processed) monomial
- so far estimated value multiply with x,
- add the value of the newly read coefficient,
- GOTO 1;

Example

```
program nothing;
var i,a,sum,degree,x:integer;
{Evaluate a polynomial for a value x, use variable a
to read the coefficients}
begin
      readln(degree); readln(x);
      sum:=0;
      for i:=0 to degree do
      begin sum:=sum*x;
            readln(a);
            sum:=sum+a;
      end;
      writeln('The value is:  ',sum);
end.
```

Excursion – labels and GOTO

- It is possible to perform (loosely controlled) skips across the program in Pascal.

Excursion – labels and GOTO

- It is possible to perform (loosely controlled) skips across the program in Pascal.
- After defining the global variables (section `var`) we define a section `label`. There we list the used labels.

Excursion – labels and GOTO

- It is possible to perform (loosely controlled) skips across the program in Pascal.
- After defining the global variables (section `var`) we define a section `label`. There we list the used labels.
- Then we may use these labels in the program

Excursion – labels and GOTO

- It is possible to perform (loosely controlled) skips across the program in Pascal.
- After defining the global variables (section `var`) we define a section `label`. There we list the used labels.
- Then we may use these labels in the program
- and by `goto label;` perform a skip there.

Excursion – labels and GOTO

- It is possible to perform (loosely controlled) skips across the program in Pascal.
- After defining the global variables (section `var`) we define a section `label`. There we list the used labels.
- Then we may use these labels in the program
- and by `goto label;` perform a skip there.
- Never use GOTO (in structured programming). I am using it in pseudocode in order to postpone the cycling condition after the kernel of the algorithm.

Defining functions and procedures

- It happens that several (nontrivial) operations get performed many times (and it is embarassing to write them more than once).

Defining functions and procedures

- It happens that several (nontrivial) operations get performed many times (and it is embarassing to write them more than once).
- Procedures and functions provide us with a possibility of defining once and using (calling) many times.

Defining functions and procedures

- It happens that several (nontrivial) operations get performed many times (and it is embarassing to write them more than once).
- Procedures and functions provide us with a possibility of defining once and using (calling) many times.
- Procedure is a part of a program. Procedure is able to process given parameters.

Defining functions and procedures

- It happens that several (nontrivial) operations get performed many times (and it is embarassing to write them more than once).
- Procedures and functions provide us with a possibility of defining once and using (calling) many times.
- Procedure is a part of a program. Procedure is able to process given parameters.
- Function is a part of a program. It is able to process given parameters and to return a result.

Defining functions and procedures

- It happens that several (nontrivial) operations get performed many times (and it is embarassing to write them more than once).
- Procedures and functions provide us with a possibility of defining once and using (calling) many times.
- Procedure is a part of a program. Procedure is able to process given parameters.
- Function is a part of a program. It is able to process given parameters and to return a result.
- Examples: Cross the street; write out a message; arrive somewhere (by a train); calculate a factorial...

Defining a function

`function` name(argument :`type`;...):`type_of_result`

- Start with keyword `function` followed by name of the function.

Defining a function

<u>`function`</u> name(argument :<u>`type`</u>;...):<u>`type_of_result`</u>

- Start with keyword `function` followed by name of the function.
- arguments are listed in parentheses (as if we defined variables).

Defining a function

`function` name(argument :`type`;...):`type_of_result`

- Start with keyword `function` followed by name of the function.
- arguments are listed in parentheses (as if we defined variables).
- Inidividual arguments get separated by a semicolon (while defining).

Defining a function

<u>`function`</u> name(argument :<u>`type`</u>;...):<u>`type_of_result`</u>

- Start with keyword `function` followed by name of the function.
- arguments are listed in parentheses (as if we defined variables).
- Inidividual arguments get separated by a semicolon (while defining).
- After a colon we put the type of the result.

Defining a function

`function` name(argument :`type`;...):`type_of_result`

- Start with keyword `function` followed by name of the function.
- arguments are listed in parentheses (as if we defined variables).
- Inidividual arguments get separated by a semicolon (while defining).
- After a colon we put the type of the result.
- Value of the result gets assigned into a special variable with the same name as the function has.

Example

```
function sum_up(a:integer; b:integer):integer;
begin
        sum_up:=a+b;
end;
```

Example

```
program x;
var a:integer;


function sum_up(a:integer; b:integer):integer;
begin
        sum_up:=a+b;
end;

begin
        a:=sum_up(5,10);
        writeln(a);
end.
```

Local variables

- Each function may use special variables (its own).

Local variables

- Each function may use special variables (its own).
- These variables are called the *local* variables.

Local variables

- Each function may use special variables (its own).
- These variables are called the *local* variables.
- We define them in a normal way, just their definition appears after the header of a particular function-definition:

Local variables

- Each function may use special variables (its own).
- These variables are called the *local* variables.
- We define them in a normal way, just their definition appears after the header of a particular function-definition:
- ```
  function f(a:integer):boolean;
  var b,c:integer;...
  begin...end;
  ```
  ```

Example

```
function sum_up(a:integer; b:integer):integer;
var c:integer;
begin
        c:=a+b;
        sum_up:=c;
end;
```

Note that the variable used to define the result is *write-only*. It must **never** be read! (It could not be distinguished from calling a parameter-less function.)

Scope resolution

- Except of global variables we obtain so called *local* variables.

Scope resolution

- Except of global variables we obtain so called *local* variables.
- Local variables are visible only from the appropriate functions.

Scope resolution

- Except of global variables we obtain so called *local* variables.
- Local variables are visible only from the appropriate functions.
- A local variable may have the same name as some global one.

Scope resolution

- Except of global variables we obtain so called *local* variables.
- Local variables are visible only from the appropriate functions.
- A local variable may have the same name as some global one.
- In case of this conflict, inside the function only the local variable is visible.

Scope resolution

- Except of global variables we obtain so called *local* variables.
- Local variables are visible only from the appropriate functions.
- A local variable may have the same name as some global one.
- In case of this conflict, inside the function only the local variable is visible.
- Values of the parameters are (by default) a value-parameters, i.e., the value of an expression is copied. If the function changes this value, this change is not propagated to the caller.

Example

```
function sum_up(a:integer; b:integer):integer;
begin
        sum_up:=a+b;
        a:=0;
end;
begin
        x:=5; y:=10; c:=sum_up(x,y);
        writeln(x);
end.
```

Reference-parameters

Sometimes we want to propagate the argument-change to the caller. How can we do that?

We use a keyword var in an appropriate moment:

```
function f(var a:integer; b:integer):integer;
begin
        a:=5;
        b:=5;
end;
...
x:=0; y:=0; a:=f(x,y);
writeln(x); writeln(y);
...
```

Result: 5 and 0; if reference-parameter applied on not a variable ⇒ error!

Parameter-free functions

It makes sense to define functions without parameters (e.g., a function reading the data).
Then we omit parentheses behind the function-name (when, both, defining and calling it):

```
function x:integer;
begin
        x:=10;
end;

...
a:=x;
...
```

Procedures

'Procedures are functions that return no value.'

```
procedure name(arguments);

... name(arguments);...
```

example:

```
procedure writeit(a:integer;b:integer);
begin
    writeln(a); writeln(b);
    {We have outputted the parameters}
end;

... writeit(5,10);...
```