European Funds
Knowledge Education Development
Wrocław University
of Science and Technology
European Union
European Social Fund
ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Data Structures and Algorithms – W01

Iterators – part 1

Complexity - part 1

Contents

- General remarks about the color scheme
- Iteration
- Recursion
- Iterators
  - Interface `Iterable<X>`
  - Iterator on the array
  - `FilterIterator`, `Predicate`
  - Iterator and foreach loop
- Computational complexity part 1
  - The concept of the algorithm
  - Notation and terminology
  - An example of improving the algorithm - exponentiation

General remarks

- Colors:
  - No thick frame - content directly related to the course
  - Blue thick frame - programming information related to practice or programming environments.
  - Violet thick frame - other information.
- Program codes:
  - Full versions can be found on ePortal
  - During the lecture, the presented codes will be formatted to take up less space and some methods may be omitted.

Iteration

- **Iteration** - repeatedly repeating the same block of actions.
- The **number of repetitions** can be specified **explicitly** (known a priori)

```
i=0; s=0;
do{
  s+=a[i]; i++;
} while (i<n);
```

- Or as a result during the processing of a specific situation (fulfilling / not fulfilling a given condition)

Recursion

- **Recursion** - calling the method **by itself** (directly or indirectly). This means implementing the idea of dividing the problem into subproblems of the same nature, but of smaller sizes

```
public static void drawNumberSequence(int k){
  if (k==0) System.out.println(k);
  else {
    System.out.print(k+" ");
    drawNumberSequence(k-1);
  }
}
public static void drawPyramid(int n){
  if (n==0) System.out.println(n);
  else {
    drawNumberSequence(n);
    drawPyramid (n-1);
  }
}
public static void main(String[] args) {
 drawPyramid(7);
}
```

```
7 6 5 4 3 2 1 0
6 5 4 3 2 1 0
5 4 3 2 1 0
4 3 2 1 0
3 2 1 0
2 1 0
1 0
0
```

aisd.W01a.RekurencjaDemo

Iterative array processing 1/2

- Array processing often boils down to using an indexed variable whose value changes in a loop in an explicitly specified way.

```
public class Student {
  int indexNo;
  double scholarship;
   public Student(int no, double amount){
     indexNo=no;
     scholarship=amount;
   }
   public void increaseScholarship(double amount){
     scholarship+=amount;
   }
   public void showData(){
     System.out.printf("%6d %8.2f\n",indexNo,scholarship);
   }
}
```

aisd.W01a.Student

Iterative array processing 2/2

- Iterative operations on the student board can be implemented as follows:

```
// assume we have following list of students:
Student [] s = new Student[5];
s[0]=new Student(1,500);
s[1]=new Student(2,400);
s[2]=new Student(3,0);
s[3]=new Student(4,500);
s[4]=new Student(5,700);
// Increase the scholarship for all students
for (int i = 0; i< s.length; i++)
  s[i].increaseScholarship(50);
// Displaying the list of scholarships:
for (int i = 0; i< s.length; i++)
  s[i].showData();
```

```
1   550,0
2   450,0
3    50,0
4   550,0
5   750,0
```

aisd.W01a.StudentDemo

Iterators 1/2

- An **iterator** is a class **object** that is used to **navigate elements of a collection** in an orderly manner so that each element can be **visited exactly once**.
- This means that we see the collection as a **linear structure**.
- This is a *design pattern*.
- Its aim is to hide the internal structure of the collection.
- Using iterators (instead of accessing elements directly to the collection with methods specific to this collection) allows you in future to convert the collection to another without changing the code using iterators.
- In more complex collections (trees, graphs, etc.), finding the "next" element can be difficult for the user of the collection, hence provide the collection's iterator the use of such a collection
- There may be many ways to go through the collection, then you must create the appropriate iterator for each method.

Iterators 2/2

- In the theory of Gramma and others design patterns, it was proposed that the iterator would allow:
  - Going to **the beginning of** the collection (first)
  - **Reading** the **current** collection element (current)
  - **Move forward** to the next element (next)
  - Transition **to the end** of the collection (last)
  - **Go back** to the previous item (previous)
  - Check if the iterator went **beyond** the collection (isDone)
- Extensive iterators can also, for example, remove the element they point to
- In practice, it turns out that the vast majority of use of iterators consists in a single pass after the collection from the first element to the last one without the need to re-read the same element repeatedly.
- In addition, for certain structures, going backwards, starting from the end, or again from the very beginning is very difficult and inefficient.
- The Java developers have therefore created **one-way one-use** iterators as a **basic** iterator. It has the following features:
  - Once created, the iterator is set at the beginning of the collection
  - We only move forward
  - We can only read the current position once
  - We can not go back to the beginning
  - When we reach the end of the collection, the iterator is basically useless.

Iterators in Java

- In Java there is an `Iterator` interface in two variants:
  - As a general interface
  - As a generic interface
- Their definitions are in the `java.util` package
- It has only three methods and the assumption that the **constructor** for a particular collection should set it **BEFORE** the first element.
- The `hasNext()` method returns information about whether another element exists.
- The `next()` method "skips" over the element, returns it and becomes BEFORE the next element. If the "current" element is missing, it ends with the exception of `java.util.NoSuchElementException`.
- The `remove()` method removes the newly jumped item. Reusing without running `next ()` ends with the exception of `java.lang.IllegalStateException`.
- If the `remove()` method can not be executed (for example, the collection is unmodifiable) it ends with the exception of `java.lang.UnsupportedOperationException`

```
interface Iterator{
  boolean hasNext();
  Object next();
  void remove();
}
```

```
interface Iterator<T>{
  boolean hasNext();
  T next();
  void remove();
}
```

Interface `Iterable<X>`

- To make the most of the iterator mechanism, collections must provide their correct iterators. For this purpose, the collection class must provide the implementation of the `Iterable<X>` interface.
- It is in the package `java.util`
- Is a general or generic interface
- It has only one method:
  - `Iterator iterator();`
    or
  - `Iterator <X> iterator();`
- Thanks to the implementation of this interface, the collection will also be able to be used in the new form of the `FOR` loop (see a few more slides)

Diagram of operations using iterators

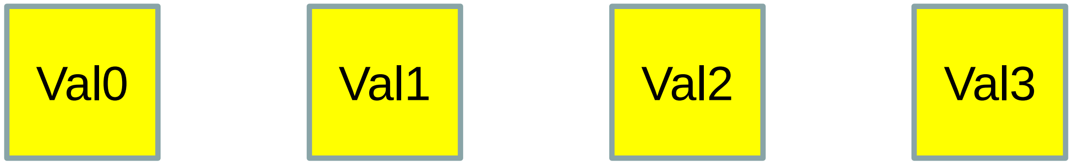
- An example of an iterator action for a four-element linear collection and the following instruction sequence:

```
SomeCollection<X> col=…; // the collection contains values
                           // Val0,Val1,Val2,Val3
Iterator iter=col.iterator();// download the iterator
                           // from the collection
iter.hasNext();     //true
X value=iter.next();       // value=Val0
iter.hasNext();     //true
X value=iter.next();              // value=Val1
iter.remove();             // Val1 removed
iter.hasNext();     //true
X value=iter.next();              // value=Val2
iter.hasNext();     //true
X value=iter.next();              // value=Val3
iter.hasNext();     //false
```

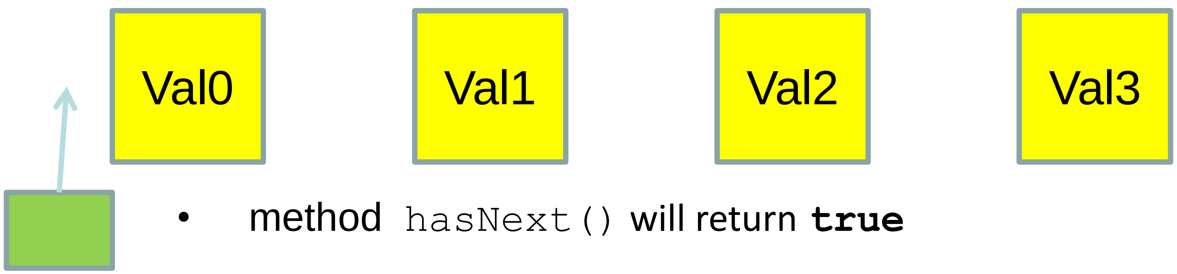
aisd.W01a.IteratorDemo

Operation of iterators 1/3

- The state of the abstract linear collection

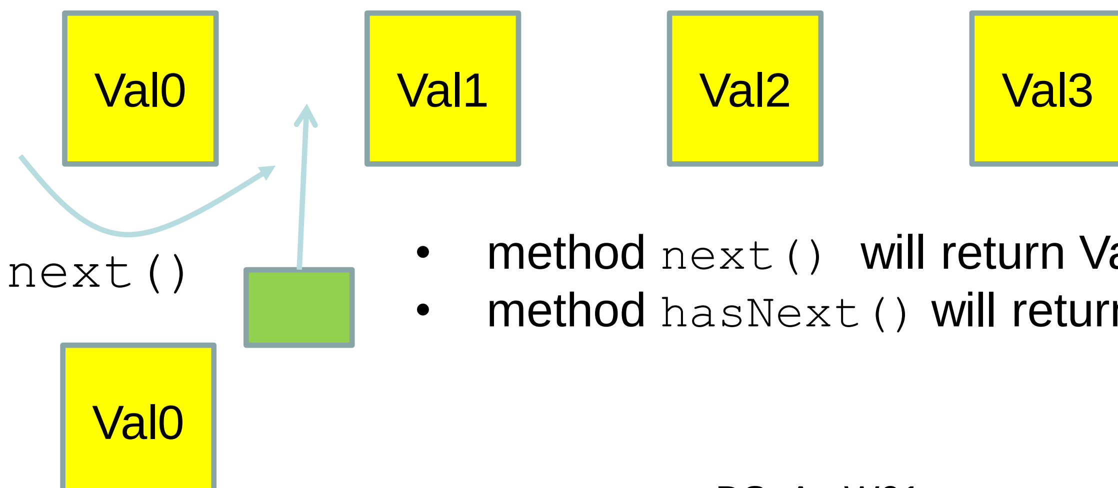


- Iterator state after construction



  - method `hasNext()` will return **`true`**

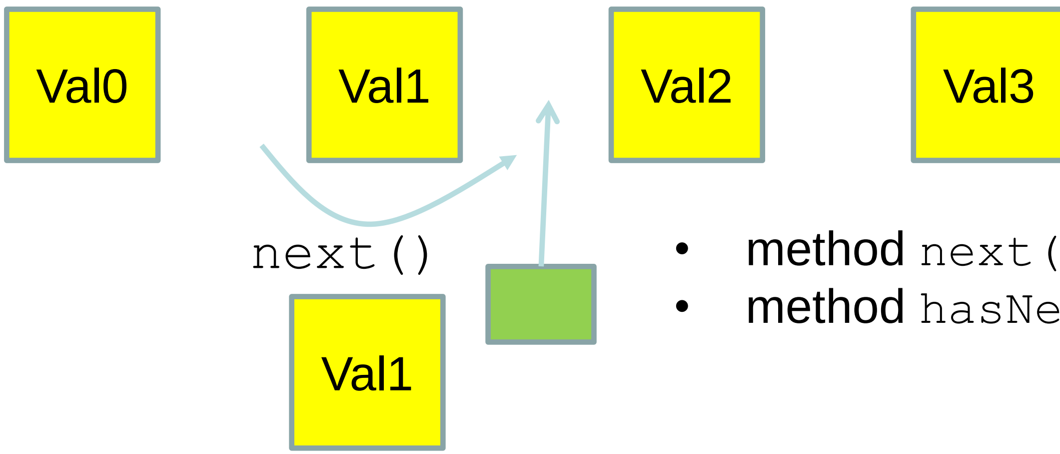
- Iterator state after execution `next()`



  - method `next()` will return Val0
  - method `hasNext()` will return **`true`**

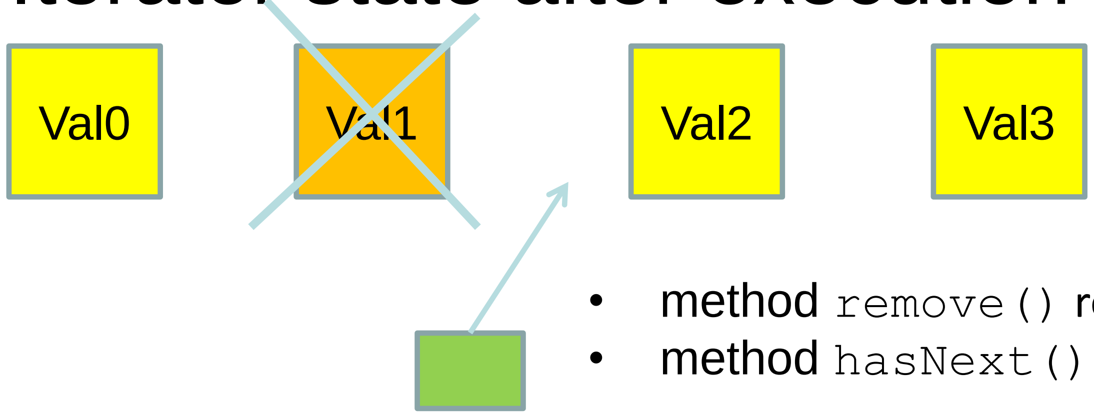
Operation of iterators 2/3

- Iterator state after next execution `next()`



- method `next()` will return Val1
- method `hasNext()` will return **`true`**

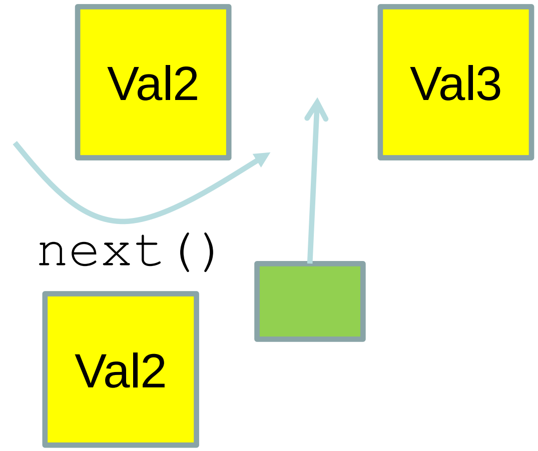
- Iterator state after execution `remove()`



- method `remove()` remove Val1 from collection
- method `hasNext()` will return **`true`**

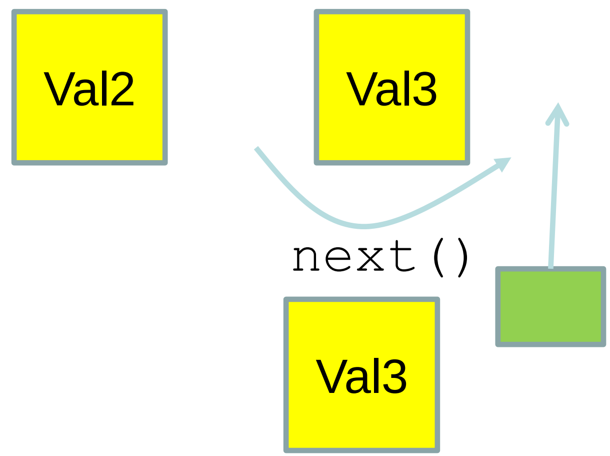
Operation of iterators 3/3

- Iterator state after next execution `next()`



- method `next()` will return Val2
- method `hasNext()` will return **`true`**

- Iterator state after next execution `next()`



- method `next()` will return Val3
- method `hasNext()` will return **`false`**

Iterator for an ordinary array

- There are **no iterators** for an ordinary array.
- How could the implementation of such an iterator look like?

```
public class ArrayIterator<T> implements Iterator<T> {
  private T array[];
  private int pos = 0;

  public ArrayIterator(T anArray[]) {
    array = anArray;
  }
  public boolean hasNext() {
    return pos < array.length;
  }
  public T next() throws NoSuchElementException {
    if (hasNext())
      return array[pos++];
    else
      throw new NoSuchElementException();
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

aisd.util.ArrayIterator

Reverse iterator

- In the Gramm idea, the iterator was bidirectional, and had symmetrical methods for reverse operation. Thanks to this, it was easy to write a general reverse iterator.
- In the case of Java library iterator it is impossible, but part of the collection may be returned by the reverse iterator as a regular iterator.
- For an array, it could look like this:

```
public class ArrayReverseIterator<T> implements Iterator<T> {
  private T array[];
  private int pos;

  public ArrayIterator(T anArray[]) {
    array = anArray;
    pos = array.length;
  }
  public boolean hasNext() {
    return pos > 0;
  }
  public T next() throws NoSuchElementException {
    if (hasNext())
      return array[--pos];
    else
      throw new NoSuchElementException();
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

aisd.util.ArrayReverseIterator

FilterIterator, Predicate

- When you want to go through a collection and perform certain operations for selected (by certain logical conditions) elements, you should use a **filter**.
- The filter must have **a method** to check whether the next element meets our condition (**predicate**).
- In Java, there is no such an iterator implemented as standard. The implementation could look like this:

aisd.util.Predicate

```
public interface Predicate<T>{
  boolean accept(T arg);
}

public final class FilterIterator<T> implements Iterator<T>{

  private Iterator<T> iterator;
  private Predicate<T> predicate;

  private T elemNext=null;
  private boolean bHasNext=true;

  public FilterIterator(Iterator<T> iterator, Predicate<T> predicate) {
    super();
    this.iterator = iterator; // cannot be null
    this.predicate = predicate; // cannot be null
    findNextValid();
  }
```

aisd.util.FilterIterator

FilterIterator 2/2

```
  private void findNextValid() {
    while (iterator.hasNext()) {
      elemNext = iterator.next();
      if (predicate.accept(elemNext)) {
        return;
      }
    }
    bHasNext=false;
    elemNext=null;
  }
  @Override
  public boolean hasNext() {
    return bHasNext;
  }

  @Override
  public T next() {
    T nextValue = elemNext;
    findNextValid();
    return nextValue;
  }
}
```

aisd.util.FilterIterator

Loop foreach

- From Java 5.0 there is a so-called foreach loop:

```
for({variable_loop_declaration}:{collection or array})
  {loop body}
```

- The execution of this loop means assigning a loop variable of subsequent values from the collection (array) and executing a loop body for it
- To do this, however, the collection must be returned by the iterator, i.e. it must provide the `Iterable <X>` interface.
  - Exception - an ordinary array has no iterator, but the foreach loop also works for it
- This is not a new mechanism, but an abbrevation notation for a particular form of the `for` loop:

```
for(TypX x:collection)
  {
  loop body
  }
```

```
for(Iterator<TypX> iter=collection.iterator(); iter.hasNext();)
  {
  TypX x=iter.next();
  loop body
  }
```

Iterator and collection

- What happens when you modify the collection during the transition after the iterator?
  - We can delete the position in front of which the iterator stands
  - We can delete the position behind the iterator
  - We can add an item, before or after the iterator
  - Add an element at the end when the iterator has reached the end
  - A few of the above modifications!
  - In concurrent programming, you can do a modification in another process!
  - etc. etc.
- The modification can be made by the methods for this collection or by the another iterator’s method.
- It was have chosen the simplest solution: if the collection structure is modified, attempting to use the iterator will result in the `ConcurrentModificationException` exception.
- Of course, this does not apply to the call to the remove () function of the considered iterator.
- As part of this lecture, this aspect of the operation of iterators will be omitted so as not to complicate the examples.

What should return the operation `next()`

- The question is: if the `next()` operation returns a reference to an object, is this:
  - the original object from the collection (that is, we receive a copy of the REFERENCE)
  - a copy of the object from the collection?
- Answer: **a copy** of the reference, that is, we have **direct access** to the element
- The same question for a collection of simple types? - answer: a copy of the value
- In the case of objects in ORDERED collections, there is the danger of destroying the order of elements.

Iterator for `Student` - example

- An array of class `Student` objects – iterators and **foreach** loop

```
// Suppose we have a list of students from previous slides:
Student[] s = new Student[5];
...
// Increase the scholarship for all students by the amount 50:
Iterator<Student> iterStud=new ArrayIterator<Student>(s);
while(iterStud.hasNext())
  iterStud.next().increaseScholarship(50);
// Show scholarship list:
iterStud=new ArrayIterator<Student>(s);
while(iterStud.hasNext())
  iterStud.next().showData();
```

aisd.W01a.StudentDemo

```
// Suppose we have a list of students from previous slides:
Student[] s = new Student[5];
...
// Increase the scholarship for all students by the amount 50:
for (Student student:s)
  student.increaseScholarship(50);
// Show scholarship list:
for (Student student:s)
  student.showData();
```

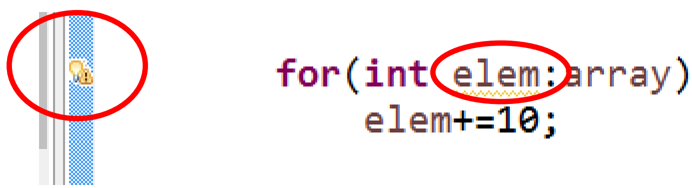
Modification in an array of `int`?

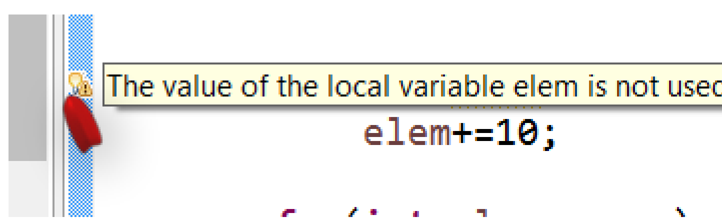
- An array of `int`
- An attempt to modify elements in a loop
- The value of the local copy is modified! - the compiler warns about it!
- Comparison with the page 20 "Foreach loop"

```java
int[] array={1,2,3,4,5,6};
for(int elem:array)
  System.out.print(elem+",");
System.out.println();

for(int elem:array)
  elem+=10;

for(int elem:array)
  System.out.print(elem+",");
System.out.println();
```

```
1,2,3,4,5,6,
1,2,3,4,5,6,
```

aisd.W01a.ArrayIntModificationDemo





Foreach loop for `Student` - example

- Modification of the content of the Student class objects is done (example two slides earlier)
- Is it possible to "replace" the student / students with new ones?

aisd.W01a.StudentModificationDemo

```
// Suppose we have a list of students from previous slides: Student
[] s = new Student[5];
...
for (Student student:s)                    // the same warning as
  student=new Student(10,1000);            // for array of int
for (Student student:s)
  student.showData();
```

```
// Suppose we have a list of students from previous slides:
Student [] s = new Student[5];
...
Iterator<Student> iter=new ArrayIterator<Student>(s);
Student stud=iter.next();                  // here you can see directly that
stud=new Student(10,1000);                 // we operate on the local variable
for (Student student:s)
  student.showData();
```

- There will be no change in the array, because we work on **local** reference **copies**.

COMPUTATIONAL COMPLEXITY PART 1

The concept of the algorithm

- There is no single fixed definition:
  - Classical: **unambiguous** recipe for the calculation of a certain **input** data in certain **finite** time to certain **result** data
  - PWN: a complete sequence of clearly defined actions necessary to perform a certain type of tasks. A procedure to solve the problem.
  - ...
- Features of the correct algorithm:
  - Receives **input data**
  - Generates **output**
  - **Feasibility** - the commands included in the algorithm are executable, i.e. available, and writing an algorithm is enough to use them.
  - **Unambiguity** - for the same input data we get the same output
  - **Determinism** - intermediate data depends on input data and previous steps
  - **Finite** - stops after completing the finite number of instructions
  - **Correctness** - the result is correct
  - **Generality** - it can be used for a wide range of input data

Algorithms

- Each computer program is basically an algorithm.
- However, the algorithm is also:
  - Instructions for using tools and machines
  - Recipes, medicines, etc.
  - Description of the technological process
  - The engineering calculation process
  - e.t.c.
- The algorithm can be expressed:
  - In colloquial language
  - In code in a known programming language
  - In pseudocode
  - Using diagrams
  - A mixture of the above

- Division of problems:
  - decidable, undecidable
  - decision problems, computational (function) problems

Notation and terminology

- **Step** - elementary operation (depends on the problem / algorithm):
  - Algebraic operation (addition, subtraction, ...)
  - Comparison (==, <=, ...)
  - Copying, assignment (=)
- **Problem size** (*n*) - number of input data:
  - Theory: number of bits
  - Real quantity: number of data to sort, etc.
  - Useful: size of a square matrix
  - More than one number: e.g. for graphs
- The **complexity of the problem** is expressed as a **function** of *n*, e.g. *f (n)* denoting the **number of steps** to be performed for input data **of size** *n*.

The fact

- The simplest algorithm for solving a problem is often not the most efficient. Therefore, when designing an algorithm, do **not settle** for just any algorithm that work.

How to calculate n^k ? (k in C)
Simple – from definition:

$n^0=1$
$n^k=n*n^{(k-1)}$ for k>=1

```
Power(n,k)
{
  Result=1;
  while(k>0)
  {
      Result=Result*n;
      k--;
  }
  return Result;
}
```

In the worse case:
k multiplications

What if k is of the order of 10^{100}?
That's the case with cryptography problems

k^n – better algorithm (1)

- You can use mathematical exponentiation properties:

$$k^{2n}=(k^n)^2$$
$$k^{2n+1}=k(k^n)^2$$

- E.a:

$$k^{89}=k(k^{44})^2$$
$$k^{44}=(k^{22})^2$$
$$k^{22}=(k^{11})^2$$
$$k^{11}=k(k^5)^2$$
$$k^5=k(k^2)^2$$
$$k^2=k*k$$

```
Power(k,n)
{
  if(n==0) return 1;
  if(n==1) return k;
  result=Power(k,n/2);
  if(n%2==0)
    return result*result;
  else
    return k*result*result;
}
```

In the worse case:
2*log(*n*) multiplications

Recursion !

E.g. on graphic cards
there was no recursion option

k^n – better algorithm (2)

- Putting power in a binary representation and dividing it into components with powers that are powers of two.

k	1
k^2	0
k^4	0
k^8	1
k^{16}	1
k^{32}	0
k^{64}	1

$$k^{89}=k^{64}*k^{16}*k^{8}*k^{1}$$

$$k^{89}=k^{1011001}$$

```
Power(k,n)
{
  Result=1;
  pow=k;
  while(n>0)
  {
    if((n%2)==1)
      Result*=pow;
    pow*=pow;
    n/=2;
  }
  return Result;
}
```

In the worse case:
2*log(*n*) multiplications

No recursion !