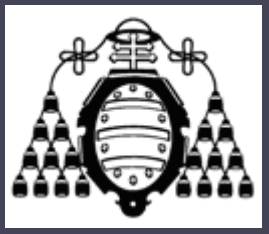


Algorithmics

Principles of algorithmics

Vicente García Díaz – garciavicente@uniovi.es

University of Oviedo, 2016

Table of contents

Principles of algorithmics

Basic concepts

Definition of algorithm (I)

- According to the RAE:
  - Ordered and finite set of operations, which allows us to find the solution of a problem

- Most accepted definition:
  - Finite sequence of steps or instructions to reach the solution of a given problem in a finite time, for any set of input values

Definition of algorithm (II)

- Is that an algorithm?

$n! = 1 * 2 * 3 \ldots * n$

$n!$ = n * (n-1) * (n-2) * (n-3) * … * 1

```
public static int factorial(int n){
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

- And that?

```
public static int factorial(int n){
    if (n < 0)
        return -1;
    else if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

Algorithms and computer science

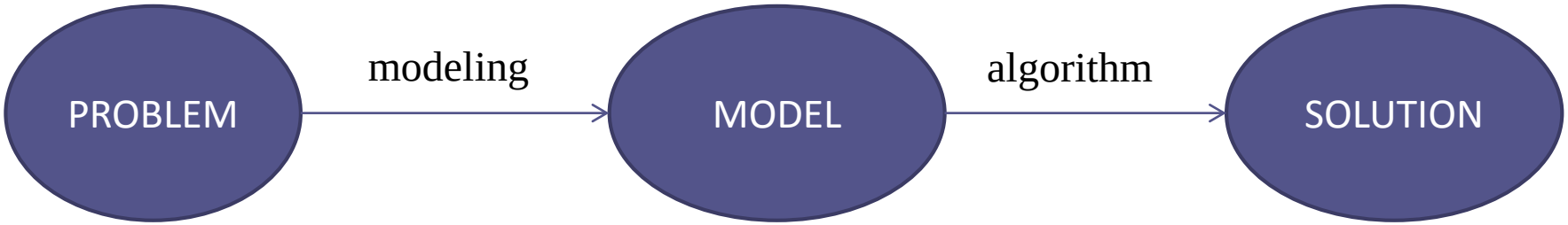
- The primary purpose of most programs is to store and retrieve information
  - We need data structures and algorithms
- This course is about:

helping you to understand how to structure information to support efficient processing to solve problems

Approaches for solving a problem

- Computer program design:
  - To design an algorithm that is easy to understand, code and debug
  - To design al algorithm that makes efficient use of the computer's resources
- What is the ideal?
  - The first one goal is related to other courses in the degree
  - **In this course we will focus on the base, the second objective**

Design of algorithms

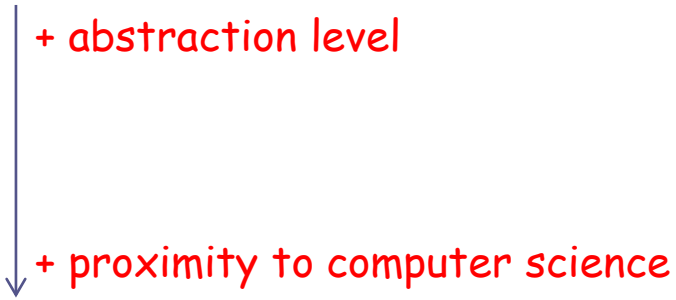


- Examples
  - Problem 1: Order a set of numbers
  - Problem 2: Set of cities connected and we want to know the Km between each pair of cities….

Formulation of algorithm (I)

- Once we have an appropriate mathematical model, we can formulate an algorithm

- How can we formulate an algorithm?
  - Natural language
  - Pseudocode
  - Programming language



Formulation of algorithm (II)

- Problem: Obtain the maximum of a set of numbers
- Model: List in which each element is a number
- Solution:
  - Natural language
    - Traverse in sequence each number in the list
    - When a number is greater than all other numbers previously traversed, mark it as the maximum

Formulation of algorithm (III)

- Problem: Obtain the maximum of a set of numbers
- Model: List in which each element is a number
- Solution:
  - Pseudocode

```
While there are unvisited numbers on the list do
  Check if the current number is the biggest visited
  number
    If it is the biggest, mark it as the maximum
EndWhile
```

  - Pseudocode (Refinement)

```
max = none
For i=1 to n do:
  If element[i] > max
    Max = element[i]
EndFor
Return max
```

Formulation of algorithm (IV)

- Problem: Obtain the maximum of a set of numbers
- Model: List in which each element is a number
- Solution:
  - Programming language

```
public static int max(int[] list){
    int max = -1;
    for (int i = 0; i < list.length; i++){
        if (list[i] > max)
            max = list[i];
    }
    return max;
}
```

PROGRAM = DATA STRUCTURES + ALGORITHM

The need for data structures

- Processor speed and memory size still continue improving
- Won't any efficiency problem we might have today be solved by tomorrow's hardware?
- Are data structures really so important?
- As we develop more powerful computers, the aim so far has always been to use that additional computing power to tackle more complex problems
  - Bigger problems sizes
  - New problems previously deemed computationally infeasable

Cost and efficiency of solutions

- Efficient algorithms
  - A solution is said to be efficient if it solves the problem within the required resource constraints
- Cost of a solution
  - The amount of resources that the solution consumes
- Resources?

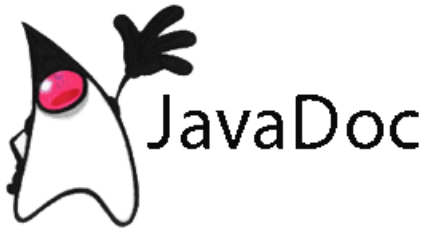
How do you solve complex problems?

- They are divided into subproblems, so that:
  - Each subproblem is well detailed
  - Each subproblem can be solved independently
  - The solutions of the subproblems can be combined

- At first, no matter how the subproblems are solved, we only need to know that they can be solved
  - Concept of abstraction

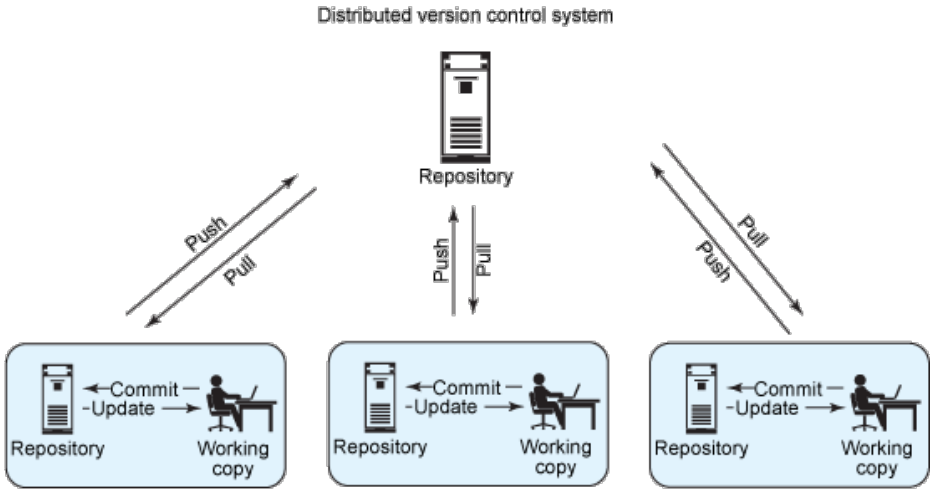
Basic tools we will use

Basic tools we will use



Git

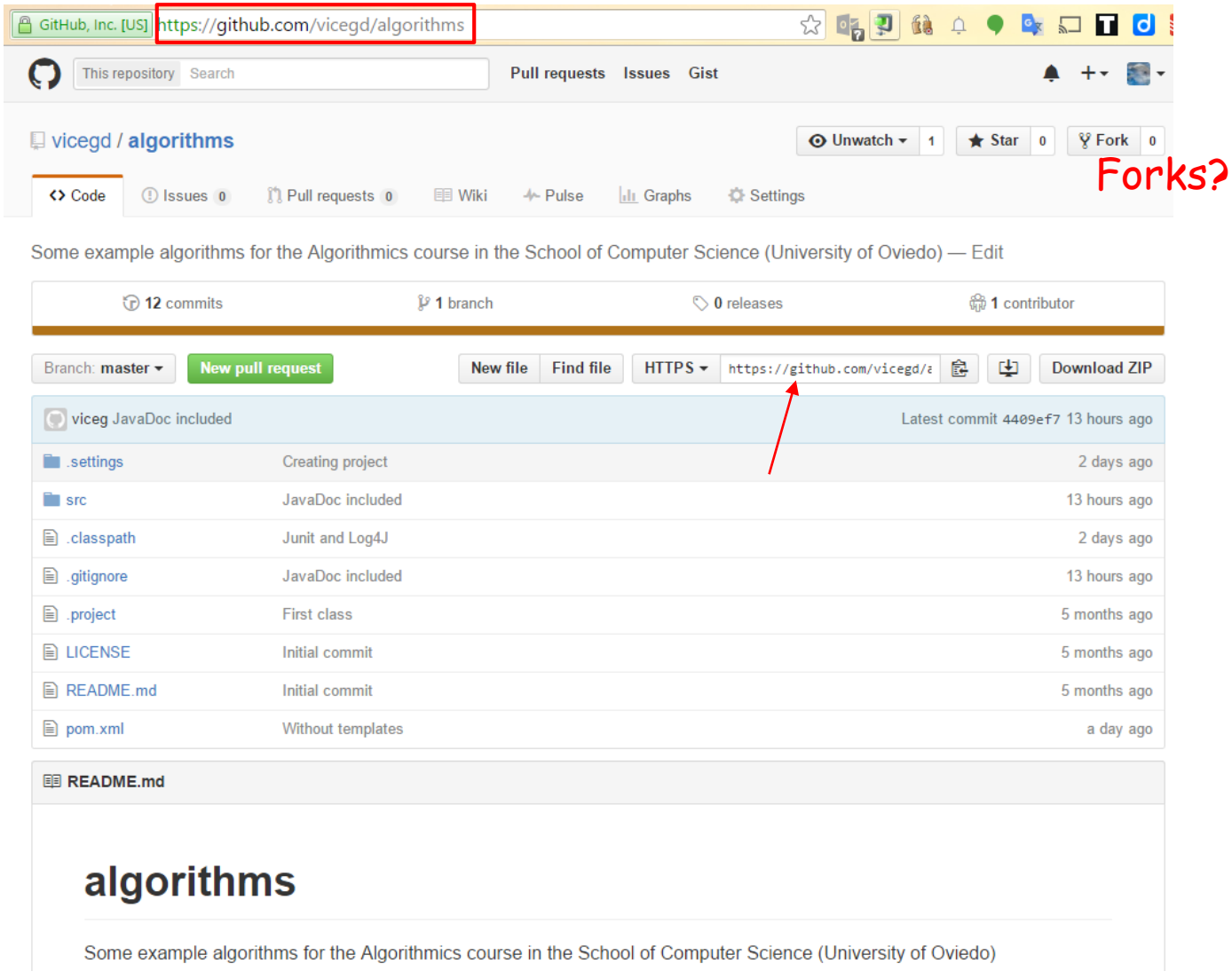
- Free and Open Source distributed version control system
  - https://git-scm.com/
- To share and keep safe the code…
- There are others such as Subversion, CVS, Perforce, ClearCase, Mercurial

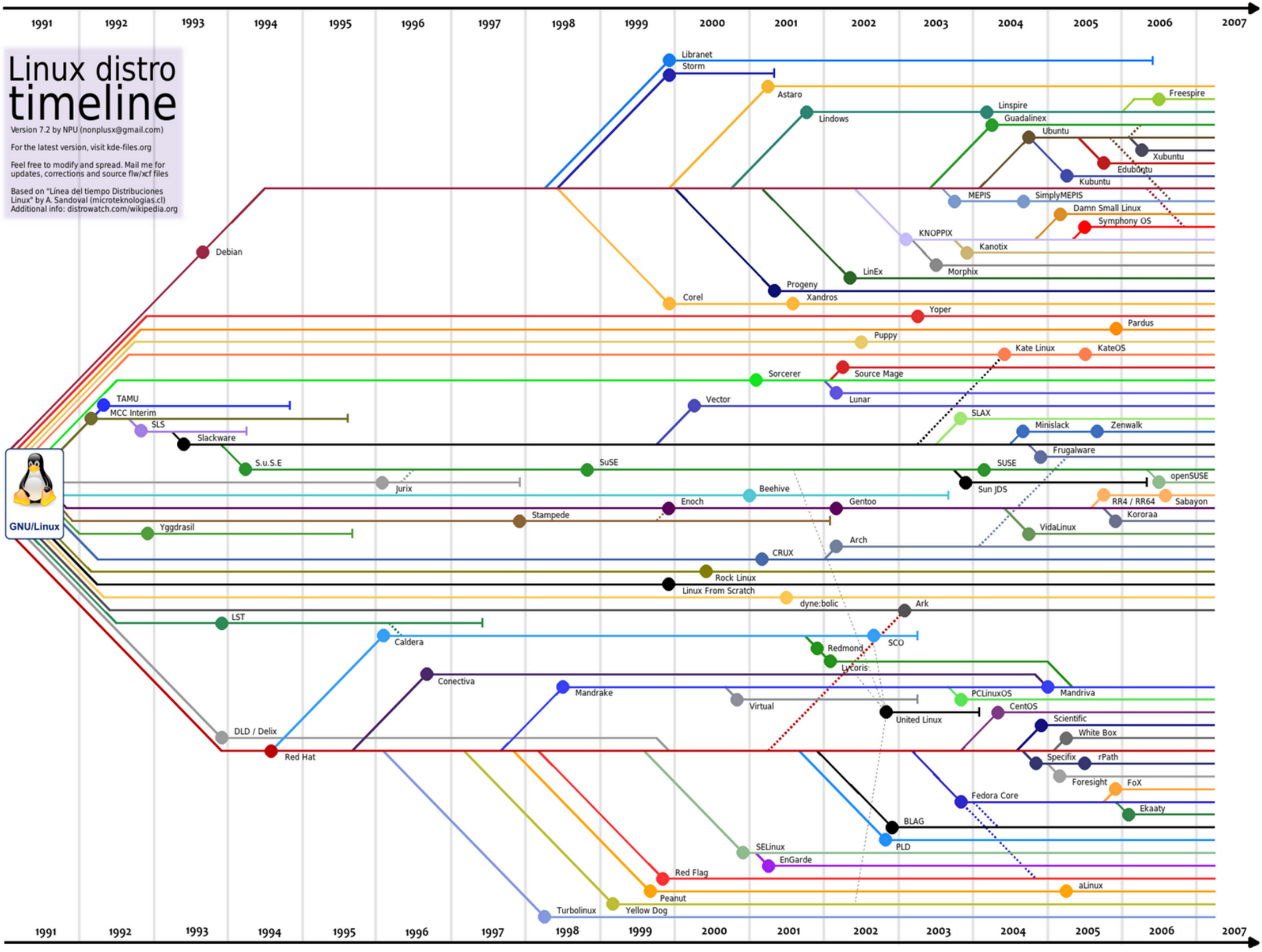


http://gotgroove.com/wp-content/uploads/2014/06/distributed-version-control-system.png

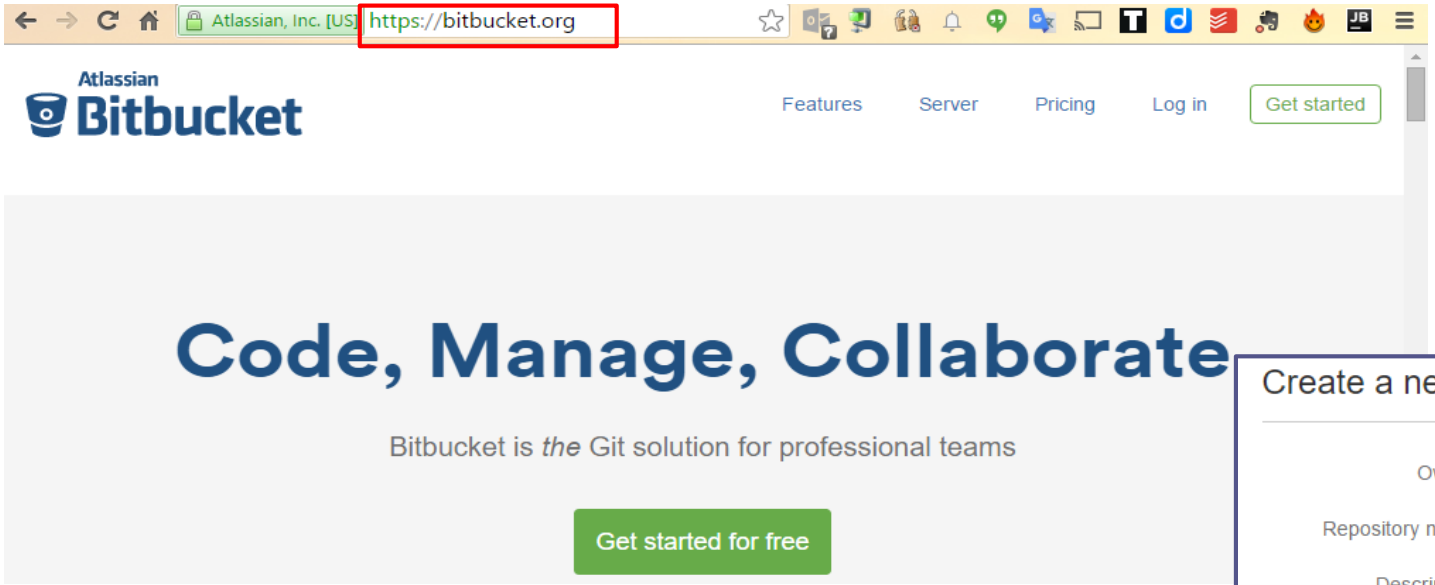
Basic tools we will use

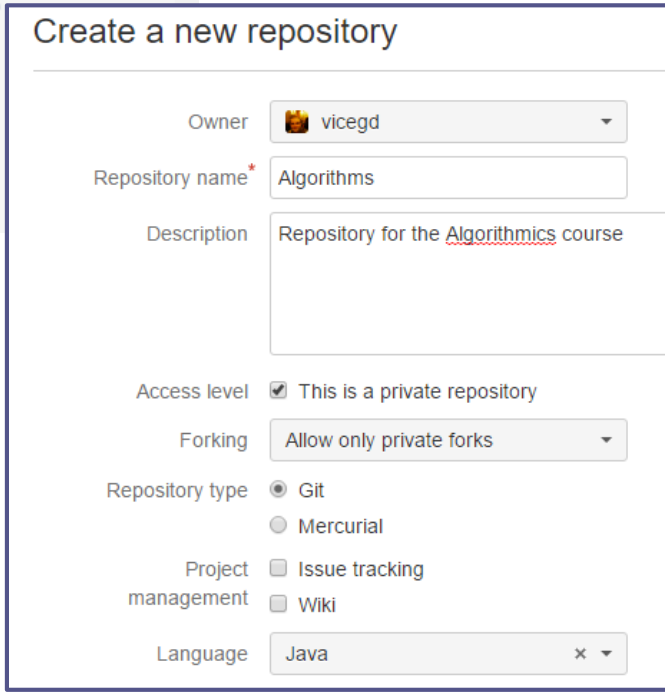
GitHub




Linux distro timeline
Version 7.2 by NPU (nonplusx@gmail.com)
For the latest version, visit kde-files.org
Feel free to modify and spread. Mail me for updates, corrections and source flw/xcf files
Based on "Línea del tiempo Distribuciones Linux" by A. Sandoval (microteknologias.cl)
Additional info: distrowatch.com/wikipedia.org
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
GNU/Linux
Debian
TAMU
MCC Interim
SLS
Slackware
S.u.S.E
Jurix
SuSE
Stampede
Yggdrasil
Enoch
Beehive
Gentoo
Vector
Sorcerer
Source Mage
Lunar
Libranet
Storm
Astaro
Lindows
Linspire
Guadalinex
Freespire
Ubuntu
Xubuntu
Edubuntu
Kubuntu
MEPIS
SimplyMEPIS
Damn Small Linux
Symphony OS
KNOPPIX
Kanotix
Morphix
LinEx
Progeny
Xandros
Corel
Yoper
Pardus
Puppy
Kate Linux
KateOS
SLAX
Minislack
Zenwalk
Frugalware
SUSE
Sun JDS
openSUSE
RR4 / RR64
Sabayon
Kororaa
VidaLinux
Arch
CRUX
Rock Linux
Linux From Scratch
dyne:bolic
Ark
LST
Caldera
Redmond
SCO
LyCoris
Conectiva
Mandrake
Virtual
PCLinuxOS
Mandriva
United Linux
CentOS
Scientific
White Box
Specifix
rPath
Foresight
FoX
Ekaaty
DLD / Delix
Red Hat
Fedora Core
BLAG
PLD
SELinux
EnGarde
Red Flag
Peanut
Yellow Dog
aLinux
Turbolinux

Bitbucket

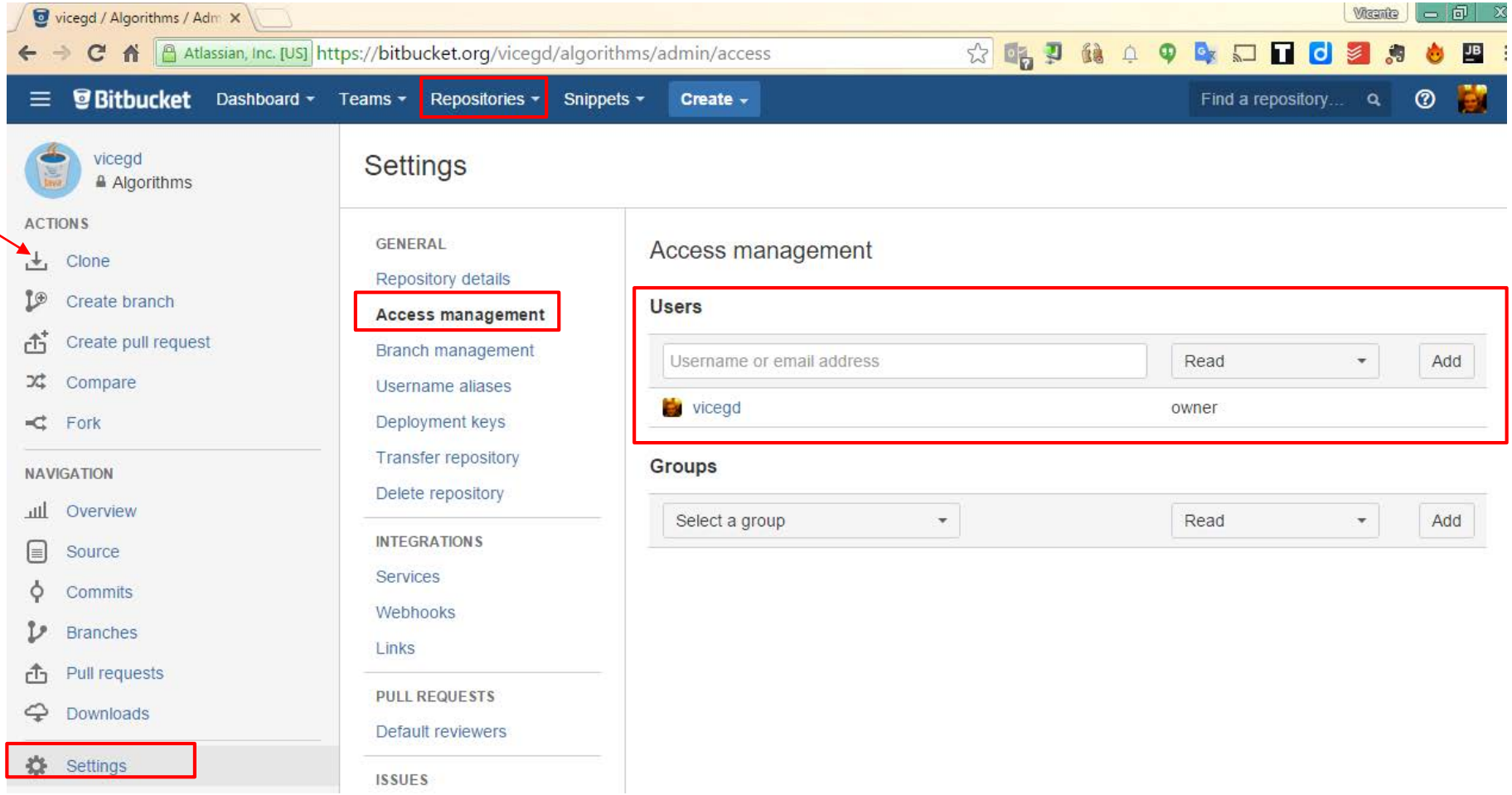




- Private projects
- **Create a new repository using your email account**

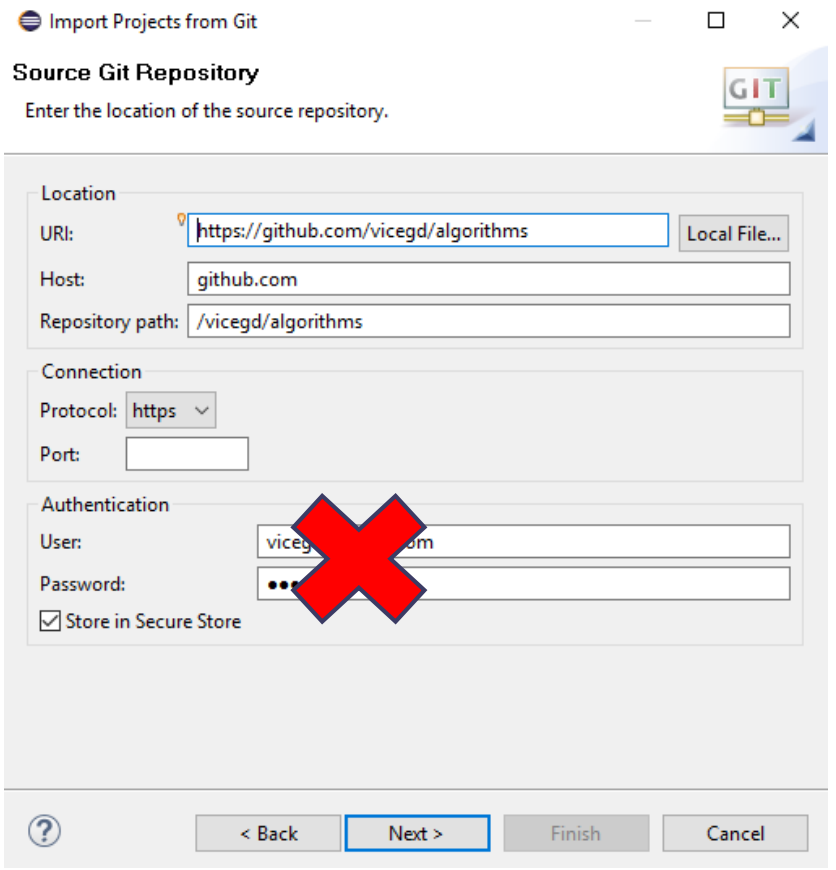
Basic tools we will use

Bitbucket (II)



Importing projects in Eclipse

- **File → Import → Git → Projects from Git → Clone URI**
  - We should include the URI of the repository
  - If we want to make changes we need user/password
  - We also should indicate the location for the local repository



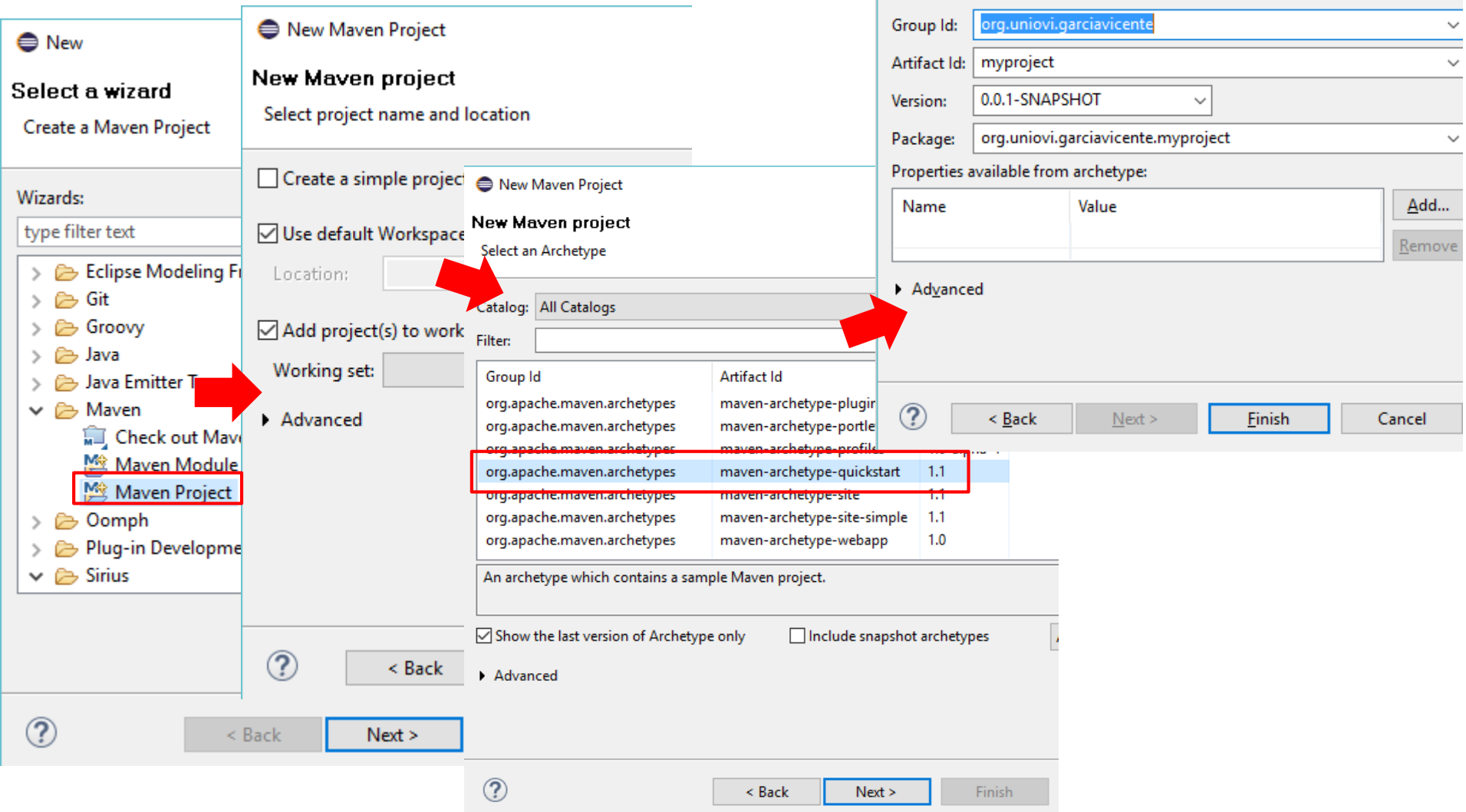
Apache Maven

- It is a tool based on the Project Object Model (POM)
- It can manage different steps in the Project such as: builds, reports, testings, etc.
- We want to use it to manage the Project dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  <modelVersion>4.0.0</modelVersion>
  <groupId>algorithms</groupId>
  <artifactId>algorithms</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.13</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.13</version>
    </dependency>
  </dependencies>
</project>
```

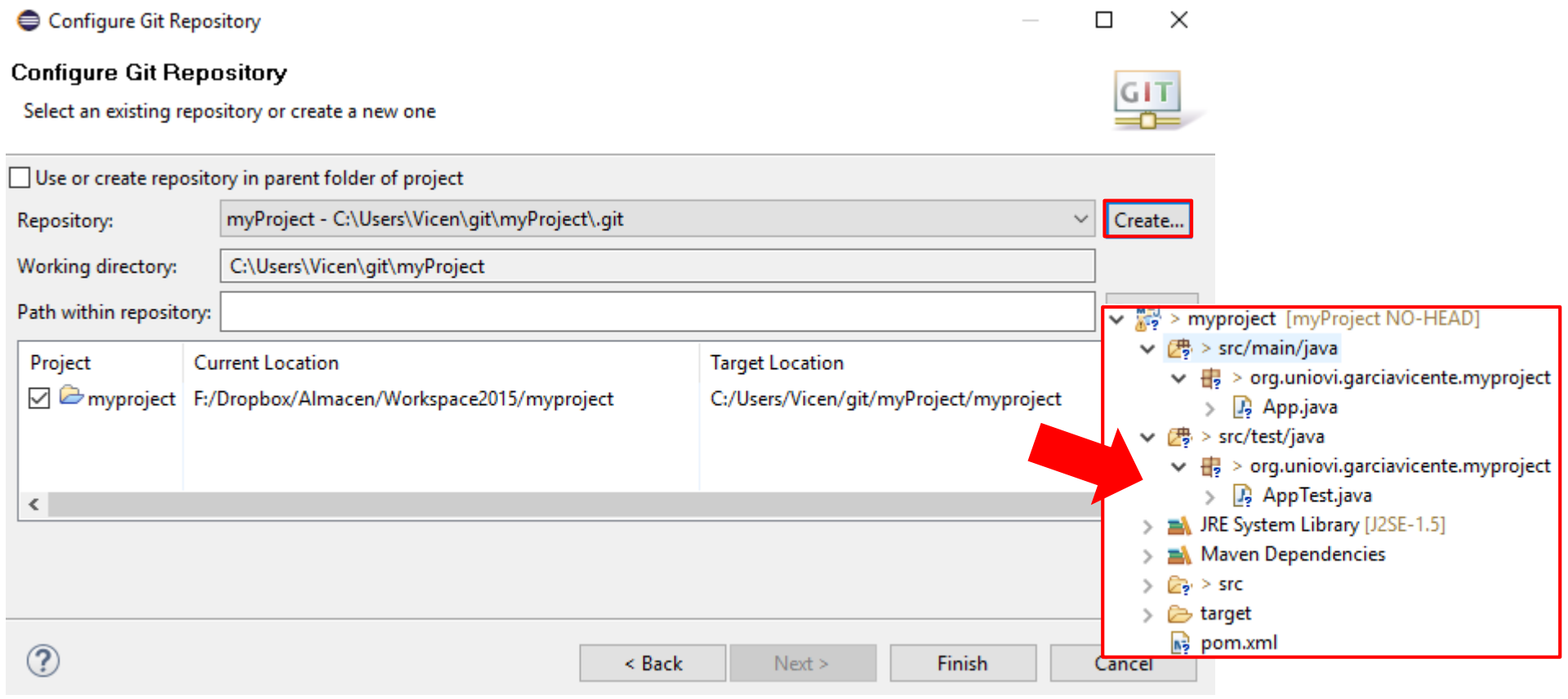
Basic tools we will use

Apache Maven



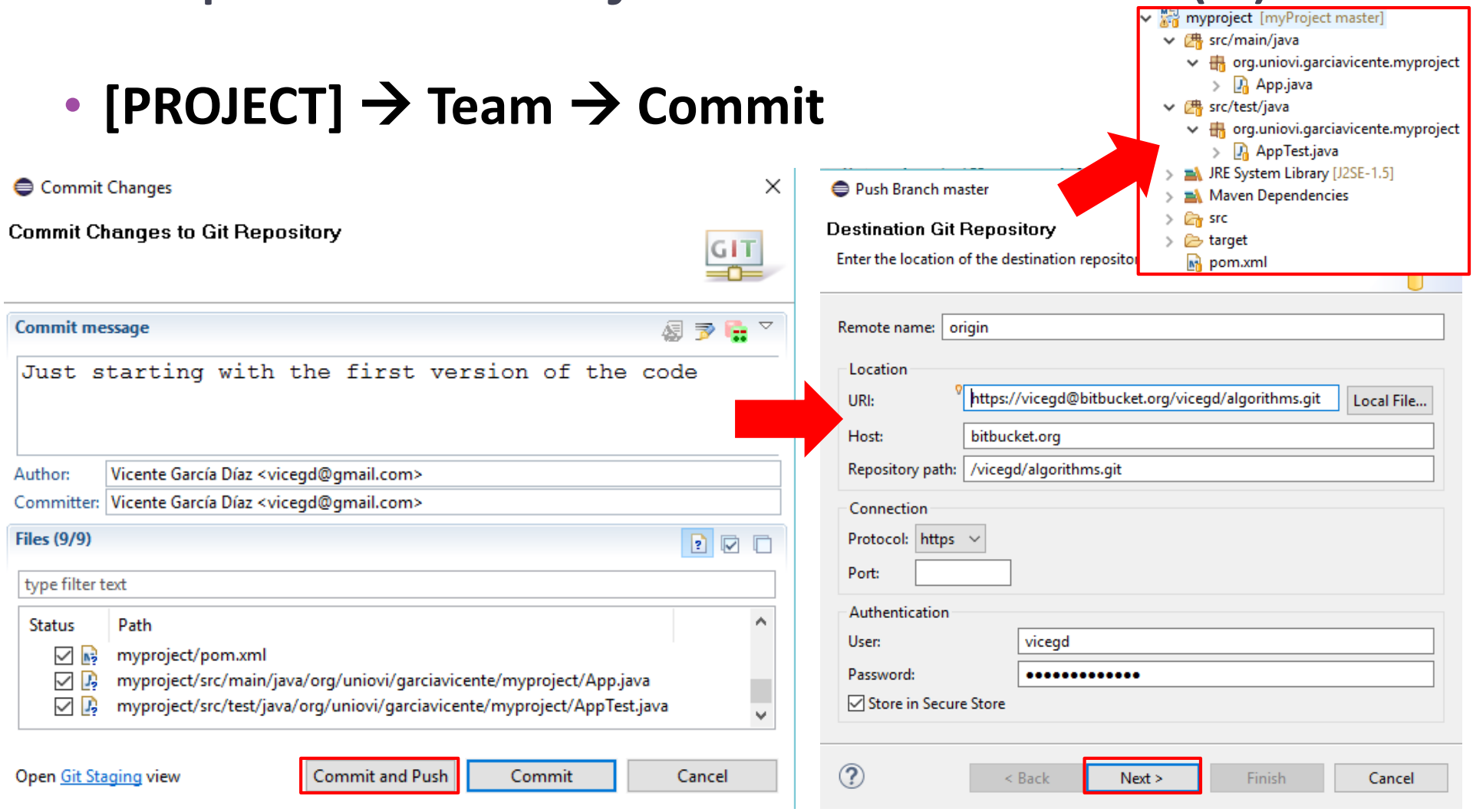
Prepare the Project to be shared

- **[PROJECT] → Team → Share Project → Git**



Prepare the Project to be shared (II)

- **[PROJECT] → Team → Commit**



JUnit

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

- JUnit is a framework to write repeteable tests
  - http://junit.org/
- It is used to create unit tests to check if software is working
- **It is mandatory to test all the code in a different class (not the class in which we write the algorithm!)**

SLF4J and LOG4J


```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.13</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.13</version>
</dependency>
```

- SLF4J (Simple Logging Facade for Java)
  - Serves as a facade for various logging frameworks (java.util.logging, logback, log4J, etc.
- Apache Log4J
  - http://logging.apache.org/log4j/2.x/
  - Java-basic logging tool with lots of possiblities

All the messages should be printed using a logging tool, not print messages!

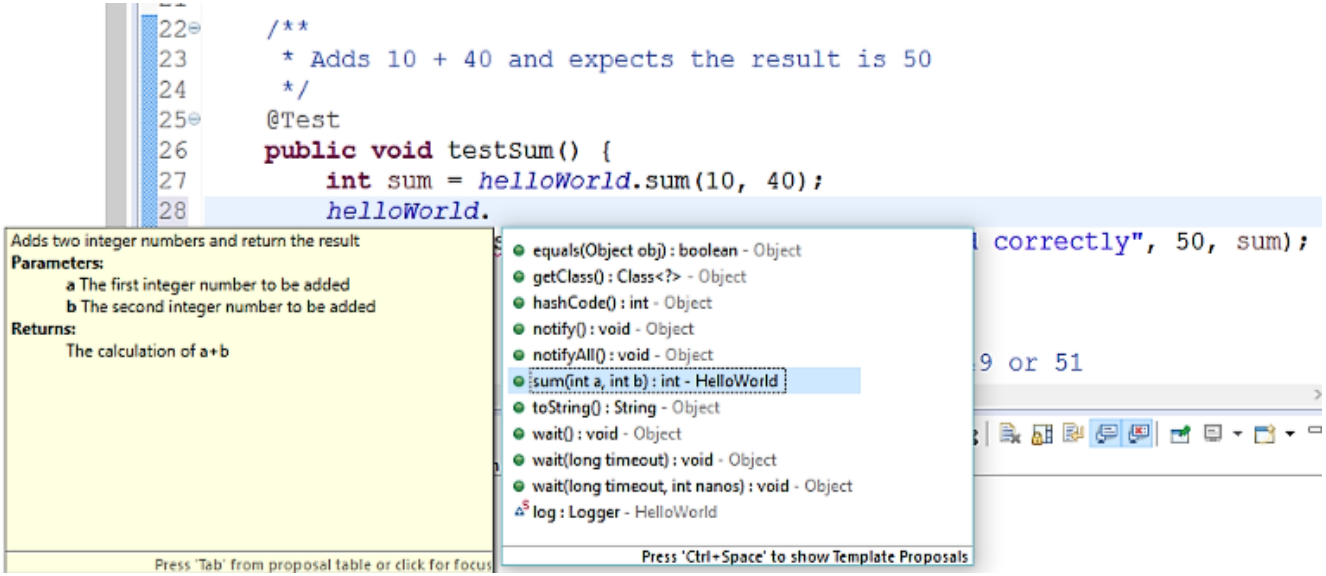
Level	Description
OFF	The highest possible rank and is intended to turn off logging.
FATAL	Severe errors that cause premature termination. Expect these to be immediately visible on a status console.
ERROR	Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console.
WARN	Use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a status console.
INFO	Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum.
DEBUG	Detailed information on the flow through the system. Expect these to be written to logs only.
TRACE	Most detailed information. Expect these to be written to logs only. Since version 1.2.12.[5]

Javadoc

- It is a documentation generator based on comments included in the Java code
  - It uses HTML as output
  - It is included in the Java SDK

All the public elements should be commented with Javadoc!

- It is integrated in several IDEs such as Eclipse
  - **Project → Generate Javadoc**



Data structures

Need for data structures

- Algorithms require an adequate data representation to achieve efficient
- Such representations, together with the allowed operations on them in a computer environment, result in the so-called **data structures**
- Most current languages work with a set of very similar data structures

Basic data structures in Java

Type	Category	Size	Interval
`boolean`	Logic	1 bit	true, false
`char`	Char	2 bytes	Unicode chars
`byte`	Integer	1 byte	[-128, 127]
`short`	Integer	2 bytes	[-32768, 32767]
`int`	Integer	4 bytes	[-2.14*E8, 2.14*E8]
`long`	Integer	8 bytes	[-9.22*E18, 9.22*E18]
`float`	Real	4 bytes	[-3.4*E38, 3.4*E38]
`double`	Real	8 bytes	[-1.7*E308, 1.7*E308]

Arrays

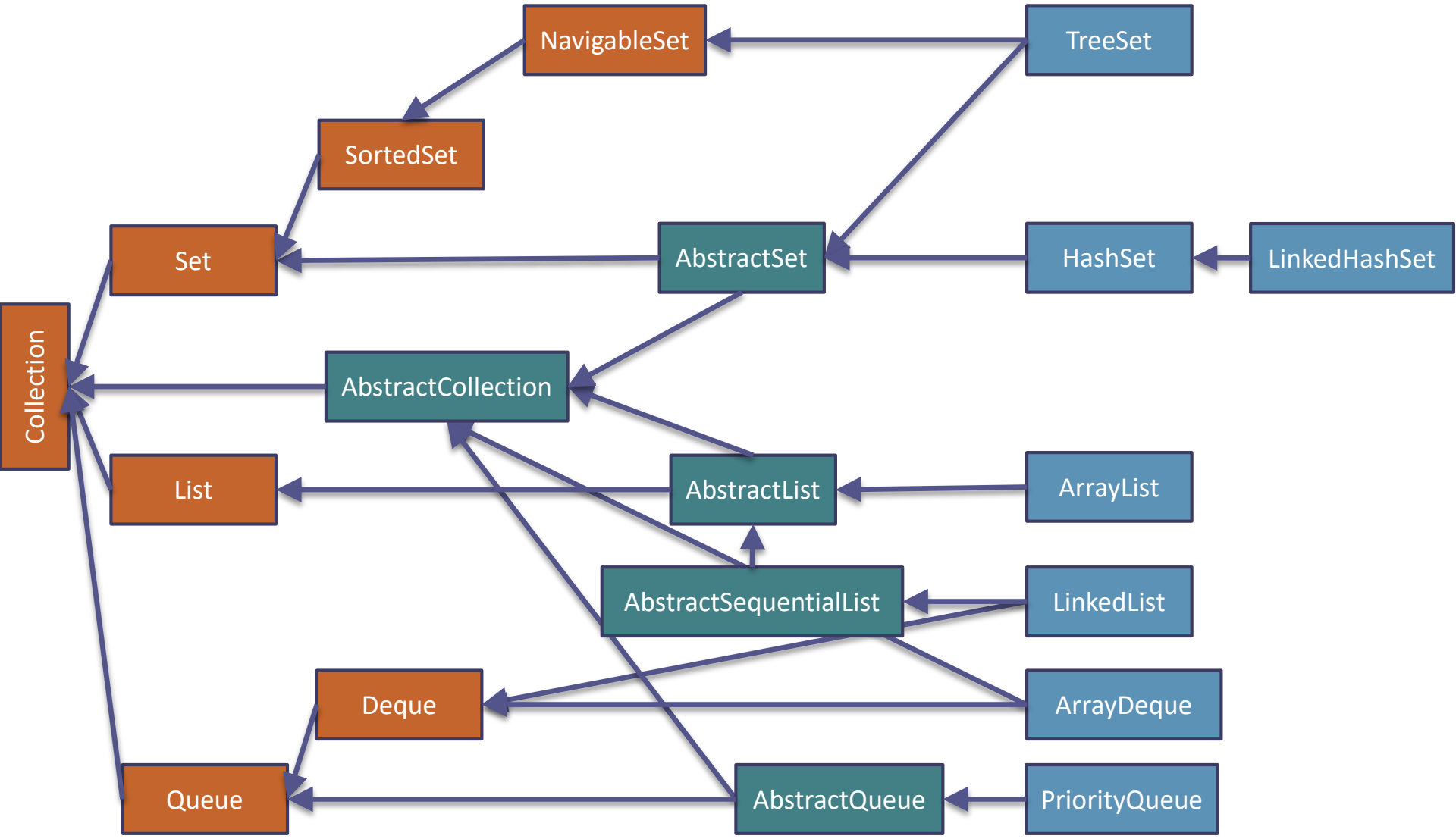
- Useful data structure
- They are not very versatile, but exist in virtually all programming languages
- Some disadvantages:
  - Size?
    - You cannot dynamically resize the array. Fixed number of elements
  - Types?
    - You cannot introduce elements of different types. Elements of the same type
  - Access?
    - The elements are only accessed by their position
  - They do not have certain operations:
    - Sets of items
    - Stacks of items
    - Priority queues

Data collections in Java

- Matrices or arrays:
  - They have a fixed size
  - They are not flexible
- …but Java has an API for working with collections:
  - It offers a great power
  - Facilitates programming tasks
    - Quality, speed, ...
  - It is based on a set of interfaces and implementations
  - It is called **Java Collection** Framework
    - Part of **java.util** package

Data structures

Java Collections



Interfaces

Interface	Description
`Collection`	Enables to work with groups of objects
`List`	Extends `Collection` to handle sequences (list of objets)
`Queue`	Extends `Collection` to handle lists in which elements are retrieved from the head (often FIFO)
`Deque`	Extends `Queue` to handle a double-ended queue (often FIFO & LIFO)
`Set`	Extends `Collection` to handle sets, which must contains unique elements
`SortedSet`	Extends `Set` to handle sets sorted in ascending order
`NavigableSet`	Extends `Set` to handle retrieval of elements based on closest-match searches

Abstract implementations

Interface	Description
`AbtractCollection`	Implements most of the `Collection` interface
`AbstractList`	Extends `AbstractCollection` and implements most of the `List` interface
`AbstractSequentialList`	Extends `AbstractList` to support sequential rather than random access of the elements
`AbstractQueue`	Extends `AbstractCollection` and implements most of the `Queue` interface
`AbstractSet`	Extends `AbstractCollection` and implements most of the `Set` interface

Implementations

Interface	Description
`ArrayList`	Extends `AbstractList` and implements a dynamic array
`LinkedList`	Extends `AbstractSequentialList` and implements a linked list
`PriorityQueue`	Extends `AbstractQueue` and implements a priority-based queue
`ArrayDeque`	Extends `Deque` and implements a double-ended queue
`HashSet`	Extends `AbstractSet` and implements a set stored in a Hash table
`LinkedHashSet`	Extends `HashSet` and mantains a linked list of the entries in the set
`TreeSet`	Extends `AbstractSet` and implements a set stored in a tree

ArrayList

Output?

```
public static void main(String[] args) {
    ArrayList<String> collection = new ArrayList<String>();
    System.out.println("Initial size of the collection: " + collection.size());

    //Adding elements
    collection.add("A");
    collection.add("B");
    collection.add("C");
    collection.add("D");
    collection.add(1, "A2");
    System.out.println("Size after additions: " + collection.size());
    System.out.println("Contents: " + collection);

    //Removing elements
    collection.remove("A2");
    collection.remove(3);
    System.out.println("Size after deletions: " + collection.size());
    System.out.println("Contents: " + collection);
}
```

LinkedList

Output?

```
public static void main(String[] args) {
    LinkedList<String> collection = new LinkedList<String>();
    System.out.println("Initial size of the collection: " + collection.size());

    //Adding elements
    collection.add("B");
    collection.add("C");
    collection.add("D");
    collection.addLast("E");
    collection.addFirst("A");
    collection.add(1, "A2");
    System.out.println("Size after additions: " + collection.size());
    System.out.println("Contents: " + collection);

    //Removing elements
    collection.remove("A2");
    collection.remove(3);
    collection.removeFirst();
    collection.removeLast();
    String firstElement = collection.peekFirst();
    System.out.println("Size after deletions: " + collection.size());
    System.out.println("Contents: " + collection);
}
```

PriorityQueue

Output?

```
public static void main(String[] args) {
    PriorityQueue<String> collection = new PriorityQueue<String>();
    System.out.println("Initial size of the collection: " + collection.size());

    //Adding elements
    collection.add("B");
    collection.add("A");
    collection.add("C");
    collection.add("D");
    collection.add("A");
    //Removing elements
    collection.remove("A");

    System.out.println("Size after operations: " + collection.size());
    while (collection.peek() != null) {
        System.out.print(collection.poll());
    }
    System.out.println();
}
```

ArrayDeque

Output?

```
public static void main(String[] args) {
    ArrayDeque<String> collection = new ArrayDeque<String>();

    collection.add("B");
    collection.add("A");
    collection.add("C");
    collection.add("D");
    collection.add("A");
    while (collection.peek() != null) {
        System.out.print(collection.poll());
    }
    System.out.println();

    collection.push("B");
    collection.push("A");
    collection.push("C");
    collection.push("D");
    collection.push("A");
    collection.add("Z");
    collection.push("J");
    while (collection.peek() != null) {
        System.out.print(collection.pop());
    }
    System.out.println();
}
```

HashSet

Output?

```
public static void main(String[] args) {
    HashSet<String> collection = new HashSet<String>();
    System.out.println("Initial size of the collection: " + collection.size());

    //Adding elements
    collection.add("A");
    collection.add("B");
    collection.add("C");
    collection.add("D");
    System.out.println("Size after additions: " + collection.size());
    System.out.println("Contents: " + collection);

    //Removing elements
    collection.remove("A");
    collection.remove(1);
    System.out.println("Size after deletions: " + collection.size());
    System.out.println("Contents: " + collection);
}
```

LinkedHashSet

Output?

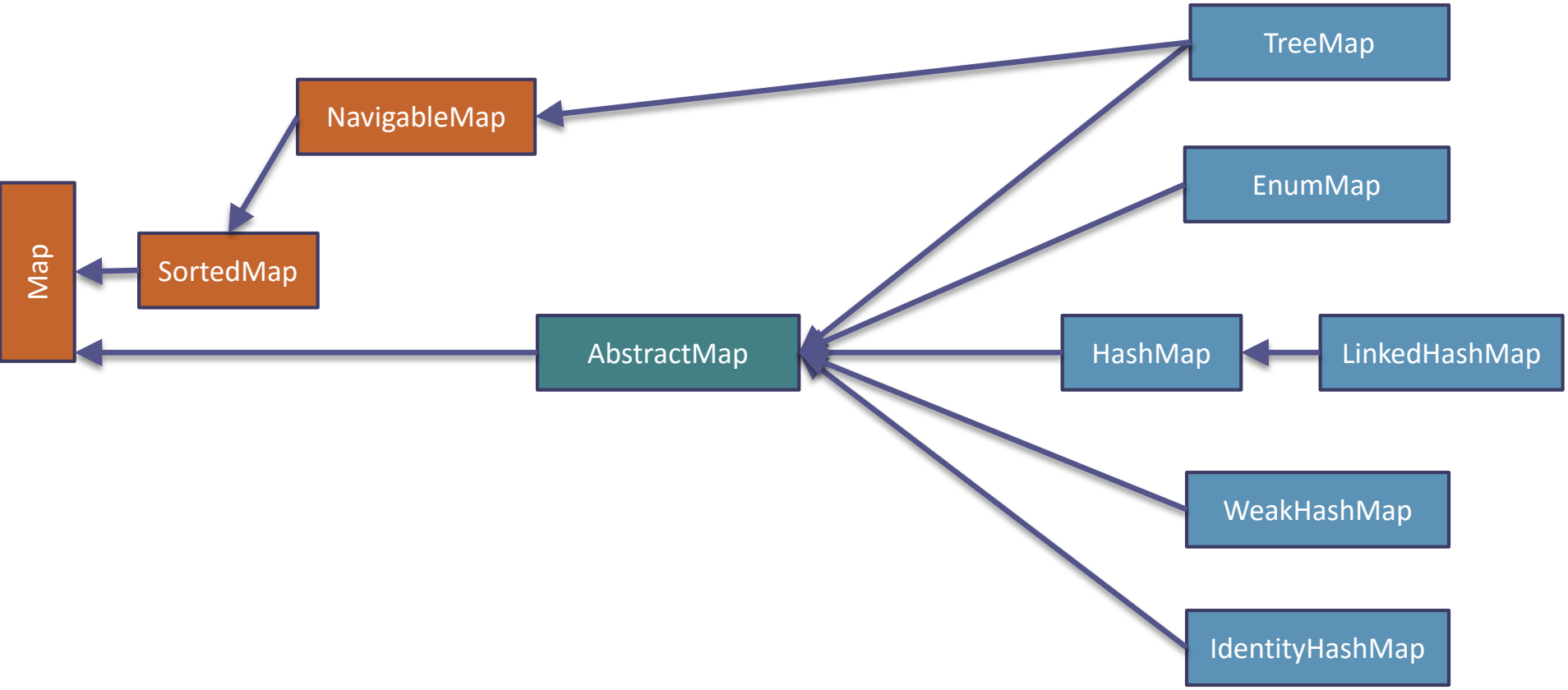
```
public static void main(String[] args) {
    LinkedHashSet<String> collection = new LinkedHashSet<String>();
    System.out.println("Initial size of the collection: " + collection.size());

    //Adding elements
    collection.add("B");
    collection.add("A");
    collection.add("C");
    collection.add("D");
    System.out.println("Size after additions: " + collection.size());
    System.out.println("Contents: " + collection);

    //Removing elements
    collection.remove("A");
    collection.remove(1);
    System.out.println("Size after deletions: " + collection.size());
    System.out.println("Contents: " + collection);
}
```

TreeSet

Output?

```
public static void main(String[] args) {
    TreeSet<String> collection = new TreeSet<String>();
    System.out.println("Initial size of the collection: " + collection.size());

    //Adding elements
    collection.add("D");
    collection.add("A");
    collection.add("B");
    collection.add("C");
    collection.add("E");
    System.out.println("Size after additions: " + collection.size());
    System.out.println("Contents: " + collection);

    //Removing elements
    collection.remove("A");
    System.out.println("Size after deletions: " + collection.size());
    System.out.println("Contents: " + collection);

    //Obtaining subset
    System.out.println(collection.subSet("B", "E"));
}
```

Working with maps



Interfaces

Abstract classes

Concrete classes

.NET equivalent to Java Collections (generics)

Java Class	.NET Class
ArrayList	List
LinkedList	LinkedList
PriorityQueue	Queue
ArrayDeque	Stack
TreeSet	SortedSet
HashMap	Dictionary
LinkedHashMap	SortedDictionary

Graphs

- A graph is a set of nodes connected by a set of lines
  - Directed graph → edges indicate the direction
  - Undirected graph → edges do not indicate direction
- A graph is a pair G = <N,A> where N is a set of nodes and A is a set of edges
- The sequences of edges can form paths and cycles

Some algorithms for graphs

- Search algorithms for the shortest path
  - Algorithm of Breadth first search
  - Algorithm of Dijkstra
  - Algorithm of Bellman-Ford
  - Algorithm of Floyd-Warshall
- Search algorithms for the minimum spanning tree
  - Algorithm of Prim

Trees

- A tree consists of a **set of nodes** and a **set of edges** connecting pairs of nodes that form a hierarchical structure
- Features:
  - It has a **root node**
  - Every node c (except the root) is connected by an edge to only a node p, where p is the father of c and c is one of the children of p
  - There is only one path from the root to each node
    - The number of edges that must be traversed is the length of the path

Some types of trees

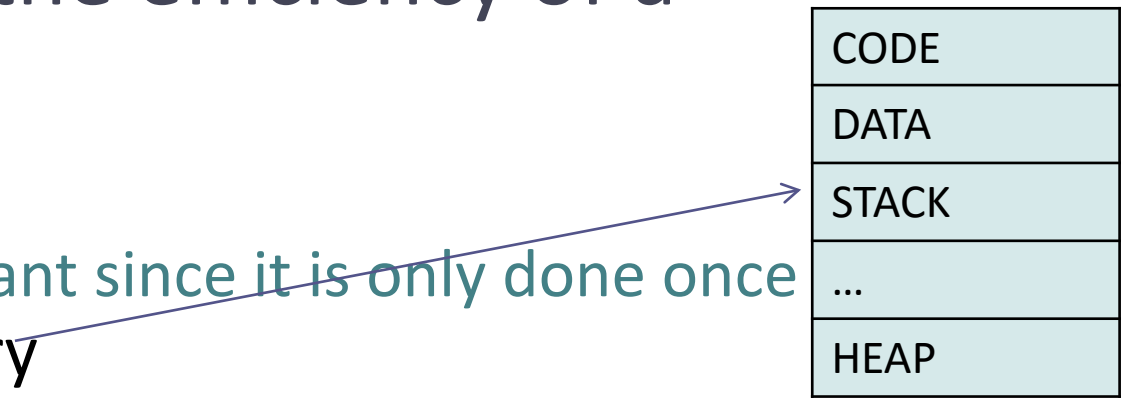
- Binary trees
  - Perfectly balanced trees
  - Search trees
    - Simply balanced trees
    - Optimal trees
    - Splay trees
  - Expression trees
- Multipath trees
  - B-trees

Algorithm analysis

How to compare two algorithms?

- To compare two algorithms in terms of efficiency we could implement both algorithms as computer programs and then run them
  - We need to use a suitable range of inputs and measure how much resources are used
- It makes no sense because:
  1. To much effort required
  2. Different programs can be better written
  3. Empirical test cases can favor one algorithm
  4. It is possible that the need of looking for an algorithm continues

How to measure the efficiency of a program?



- Compilation time
  - It is not often important since it is only done once
- Occupation of memory
  - It is sometimes a very critical parameter
  - *It depends on*: computer speed, compiler, code size, …
- Execution time
  - We always want to solve problems in the shortest possible time
  - That often run counter to the memory occupation
    - We must find a compromise solution
  - *It depends on:* computer speed, quality of compiler, problem size, number of input/output operations and **quality of the algorithm used**

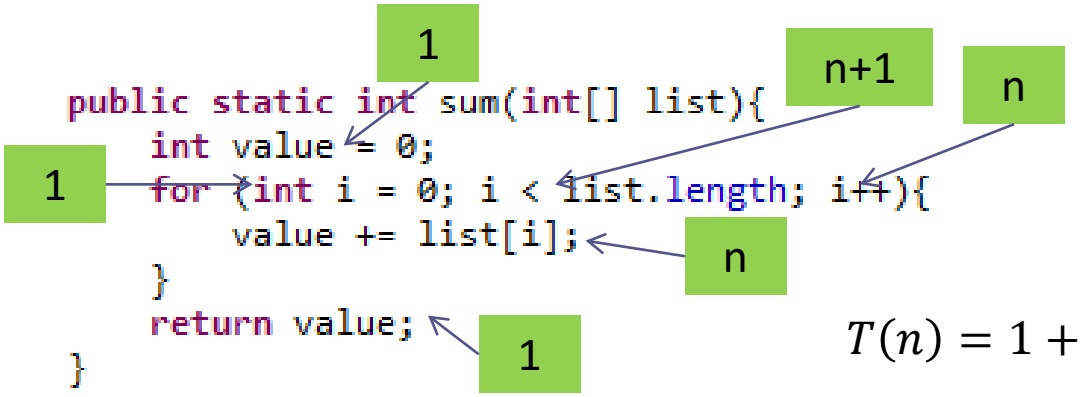
Analytical study of the runtime

- We call $T(n)$ to the execution time of a program with an input n
- We cannot use standard units of time as seconds because the execution time depends on many factors
- You can only say things like '***run time is proportional to n***', without specifying a constant because we are considering an idealized computer

What is n?

- The exact notion of what measures n depends on the nature of the problem:
  - Vector
    - The vector length
  - Matrix
    - Number of elements that compose it
  - Graph
    - Number of nodes
  - Table of a database
    - Number of records
  - …

Best, worst and average case (I)



$$T(n) = 1 + 1 + \sum_{i=1}^{n}(1 + 1 + 1) + 1 + 1 = 3n + 4$$

Best, worst and average case (II)

- T(n) best
  - It is the time in the best case scenario of the algorithm
  - Some "best case" may be difficult to appear

- T(n) worst
  - It is the time in the worst case scenario

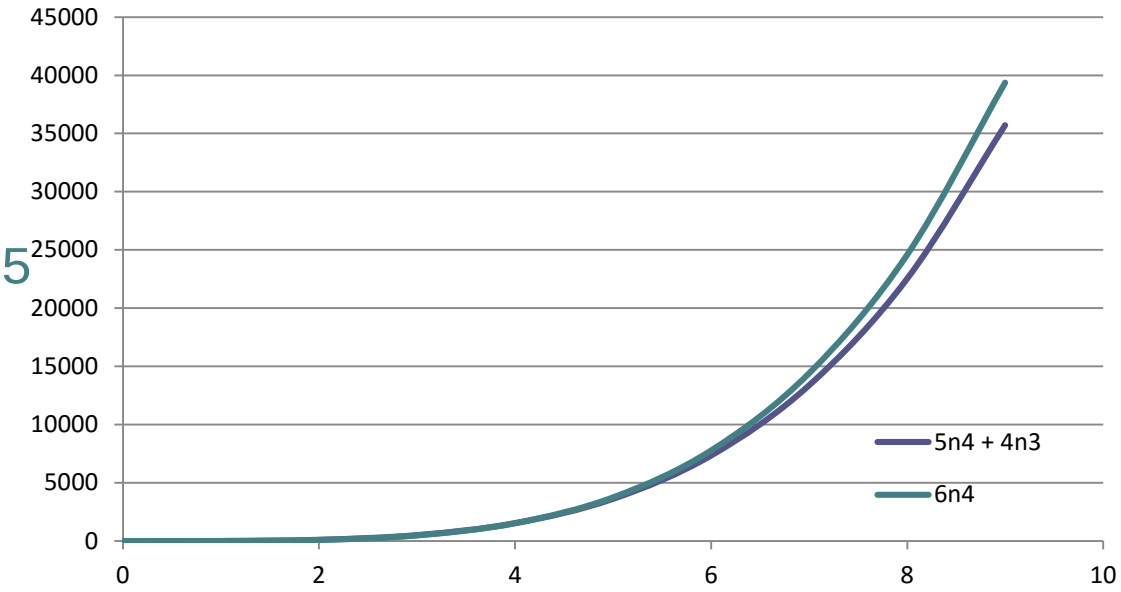
- T(n) average ➔ $T(n)_{average} = \sum_{i=1}^{N_{cases}} P_i * Ti(n)$
  - It is the average execution time for inputs of size n
  - Apparently it seems the most reasonable measure

Asymptotic notation (I)

- Refers to the speed of growth of the values of a function f(n) → O(f(n))
- T(n) is O(f(n)) if there are two positive constants c and n_0 such that $T(n) \leq c * f(n) \quad \forall n \geq n_0$
- If T(n) is O(f(n)) ➔ T(n) has a growth rate of f(n)

- Example:
  - $T(n) = 5n^4+4n^3$

  with the values c=6 and n_0=5
  - $5n^4+4n^3 <= 6n^4$



Asymptotic notation (II)

- You can evaluate a program by checking their growth rate, ignoring the constants (e.g., a program $O(n)$ will be better than a program $O(n^2)$
- However, there are cases (values of n) in which for example $O(100*n)$ behave worse than $O(n^2)$
  - … although it would only be for certain values of a small size of n
- To facilitate the analysis the values of the constants are often removed

Properties of the asymptotic notation

1. $T(n)$ is $O(T(n))$
2. $O(c*f(n)) = O(f(n))$
3. $T(n) = T_1(n)+T_2(n)+\ldots+T_m(n)$
   - ➔ $O(f(n))= \max(O(T_1(n))+O(T_2(n))+\ldots+O(T_m(n)))$
4. $T(n) = T_1(n)*T_2(n)*\ldots*T_m(n)$
   - ➔ $O(f(n))= O(T_1(n)*T_2(n)*\ldots*T_m(n))$
5. $O(\log_a(n)) = O(\log_b(n))= O(\log(n))$
6. $O(\log(n^k)) = O(K*\log(n))= O(\log(n))$
7. $O(\log^k(n)) \neq O(\log(n))$

Asymptotic analysis

- It allows you:
  - To measure the inherent difficulty of a problem
  - To estimate the time cost and space required for every algorithm presented
  - To see how each algorithm compares to other algorithms for solving the same problem in terms of its efficiency

Basic types of complexities

POLYNOMIAL

- $O(1)$ It is optimal but almost always impossible
- $O(\log n)$ It is optimal
- $O(n)$ It is a good and quite usual complexity
- $O(n \log n)$ It is a good complexity (recursive problems)
- $O(n^2)$ Appears in nested loops and similar cases
- $O(n^3)$ For large values it grows excessively
- $O(n^k)$ General case of polynomial complexity

NOT POLYNOMIAL

- $O(2^n)$ Exponential complexity. If possible, avoid it
- $O(2^k)$ General case of exponential complexity
- $O(n!)$ It is a bad complexity. It should be avoided
- $O(n^n)$ It is a bad complexity. It should be avoided

Comparison of growth

n	O	O(logn)	O(n)	O(nlog)	$O(n^2)$	$O(n^3)$	$O(2^n)$	O(n!)	$O(n^n)$
1	1	0	1	0	1	1	2	1	1
5	1	2,3219281	5	11,60964	25	125	32	120	3125
10	1	3,3219281	10	33,219281	100	1000	1024	3628800	1E+10
20	1	4,3219281	20	86,438562	400	8000	1048576	2,4329E+18	1,049E+26
50	1	5,6438562	50	282,19281	2500	125000	1,126E+15	3,04141E+64	8,882E+84
100	1	6,6438562	100	664,38562	10000	1000000	1,268E+30	9,3326E+157	1E+200
200	1	7,6438562	200	1528,7712	40000	8000000	1,607E+60	*TOO LONG*	*TOO LONG*
500	1	8,9657843	500	4482,8921	250000	1,25E+08	3,27E+150	*TOO LONG*	*TOO LONG*
1000	1	9,9657843	1000	9965,7843	1000000	1E+09	1,07E+301	*TOO LONG*	*TOO LONG*
10000	1	13,287712	10000	132877,12	1E+08	1E+12	*TOO LONG*	*TOO LONG*	*TOO LONG*

Examples of asymptotic notation (I)

- $T(n) = \frac{4}{5} * n^3 + 100$ — $O(n^3)$
- $T(n) = 10n * log(n)$ — $O(n*log(n))$
- $T(n) = 100n + \frac{1}{10000} 3n * log(n)$ — $O(n*log(n))$
- $T(n) = + \ n^2 \ + \ 50n$ — $O(n^2)$
- $T(n) = n \ log \ n + \ 2 \ n \ + \ 13$ — $O(n^*log(n))$
- $T(n) = n^2 \ + \ 4(n \ + \ log(n))^2 \ * \ n$ — $O(n^3)$
- $T(n) = \frac{4n^2 + 2log(n)}{n+1}$ — $O(n)$
- $T(n) = 16 \ log(n) \ + \ n^{0'4}$ — $O(n^{0,4})$

Examples of asymptotic notation (II)

- $T(n) = n^3 + 100^5 n$ — $O(n^3)$
- $T(n) = 3n^2 + 59n$ — $O(n^2)$
- $T(n) = \frac{1}{10} * 2^n + 4n^2$ — $O(2^n)$
- $T(n) = 5n^{1.6} + 10n*\log(n)$ — $O(n^{1.6})$
- $T(n) = n \log n + 10^4 n + 10^{100}$ — $O(n*\log(n))$
- $T(n) = 2^n + 4(n + \log(n))^2 * n$ — $O(2^n)$

Calculating execution times (I)

- A program with a O(f(n)) time complexity takes t1 sec. for a problem size of n1, how long would it take for a larger size of n2?
  - n2 = k*n1 => **k =** $\frac{n2}{n1}$
  - If for f(n1) it takes t1 and for f(n2) it takes t2 => **t2 =** $\frac{f(n2)}{f(n1)}$ *** t1**
  - EXAMPLES:
    - f(n) = O(n) ???
    - f(n) = O(n^c) ???
    - f(n) = O(log n) ???
    - f(n) = O(nlog n) ???
    - f(n) = O($n^{c1}\log^{c2}n$) ???
    - f(n) = O(c^n) ???
    - f(n) = O(n!) ???

Calculating execution times (II)

- Table of example for `n1 = 1000 y t1 = 1 u.t.` → **t2?**

Complexity	n1 = 1000	n2 = 10000	n2 = 10^6
O(1)	1 u.t.	1 u.t.	1 u.t.
O($\log_2 n$)	1 u.t.	~1.3 u.t.	~2 u.t.
O(n)	1 u.t.	10 u.t.	1000 u.t.
O(nlogn)	1 u.t.	~13 u.t.	~2000 u.t.
O(n^2)	1 u.t.	100 u.t.	10^6 u.t.
O(n^2logn)	1 u.t.	~130 u.t.	~2*10^6 u.t.
O(n^3)	1 u.t.	1000 u.t.	10^9 u.t.
O(2^n)	1 u.t.	2^{9000} u.t.	2^{999000} u.t.
O(n!)	1 u.t.	$\prod_{1001}^{10000} u.t.$	$\prod_{1001}^{1000000} u.t.$

http://www.rapidtables.com/calc/math/Log_Calculator.htm

Calculating problem sizes

- A program with a O(f(n))time complexity takes t1 sec. for a problem size of n1, what problem size would be solved for a larger time of t2?
  - t2 = k*t1 => **k =** $\frac{t2}{t1}$
  - If for f(n1) it takes t1 and for f(n2)it takes t2 => f(n2) = $\frac{t2}{t1}$ * f(n1) = k * f(n1) => **n2 = f^{-1}(K * f(n1))**
  - EXAMPLES:
    - f(n) = O(n) ???
    - f(n) = O(n^c) ???
    - f(n) = O(logn) ???
    - f(n) = O(c^n) ???

Calculating problem sizes (II)

- Table of example for `n1 = 10 y t1 = 1 u.t.` → **n2?**

Complexity	t1 = 1 u.t.	t2 = 1000	t2 = 10^6
O(1)	10	…	…
$O(\log_2 n)$	10	~2^{3000}	~$2^{3000000}$
O(n)	10	10^4	10^7
$O(n^2)$	10	~300	10^4
$O(n^3)$	10	100	10^3
$O(2^n)$	10	~20	~30

http://www.rapidtables.com/calc/math/Log_Calculator.htm

Invariance principle

Given an algorithm of $O(f(n))$ complexity, it will have always that complexity, regardless of the language in which it is programmed, the quality of the compiler used, the speed of the computer, etc… That is, these factors affect the constant of implementation, but never its complexity $O(f(n))$

Some complexitites (I)

- Sequential $\{s_1, s_2, \ldots s_k\} \rightarrow O(s_1) + O(s_2) + \ldots + O(s_k)$
- for (i = 1; i <= n; i++) { O(1) }
  - T(n) = n → O(n)
- for (i = n; i > 0; i--) { O(1) }
  - T(n) = n → O(n)
- for (i = 1; i <= n; i += 2) { O(1) }
  - T(n) = n/2 → O(n)
- for (i = 1; i <= 3*n; i += 2) { O(1) }
  - T(n) = (3*n)/2 → O(n)

Some complexitites (II)

- for (i = 1; i <= c1*n; i +=c2) { O(1) }
  - T(n) = (c1*n)/c2 → O(n)
- for (i = 1; i <= n; i *=2) { O(1) }
  - $T(n) = \log_2(n)$ → O(logn)
- for (i = n; i >= 1; i /=2) { O(1) }
  - $T(n) = \log_2(n)$ → O(logn)
- for (i = 1; i <= n; i *=c) { O(1) }
  - $T(n) = \log_c(n)$ → O(logn)
- for (i = 1; i <= n*n; i *=2) { O(1) }
  - $T(n) = \log_2(n^2)$ → O(logn)
- for (i = 1; i <= n*n*n/2; i *=c) { O(1) }
  - $T(n) = \log_c(n^3/2)$ → O(logn)

Some complexitites (III)

- for (i = 2*n; i >= 0; i -= 3)
  - for (j = 1; j <= n*n*n; j *= 2) {O(1)}
    - T(n) = … → O(n logn)
- for (i = n; i >= 1; i /=2)
  - for (j = 1; j <= n/3; j++) {O(1)}
    - T(n) = … → O(n logn)
- for (i = 1; i <=n; i *=3)
  - for (j = 1; j <= n*n*n; j*=2) {O(1)}
    - T(n) = … → $O(\log^2 n)$
- for (i = n/2; i <=n; i++)
  - for (j = 1; j <= n/3; j+=2) {O(1)}
    - T(n) = … → $O(n^2)$

Some complexitites (IV)

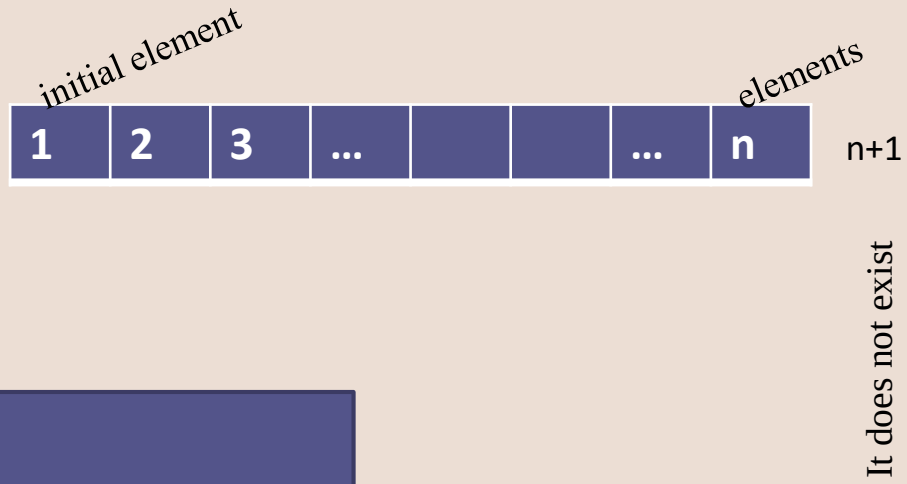
- for (i = 1; i <= n; i++)
  - for (j = i+1; j <= n; j++) {O(1)}
    - T(n) = … → $O(n^2)$
- for (i = 1; i <= n; i++)
  - for (j = n; j >=1; j--)
    - for (k = 1; k <= j; k++) {O(1)}
      - T(n) = … → $O(n^3)$
- for (i = 1; i <= n; i++)
  - for (j <=i; j <=n; j++)
    - for (k <=j; k <= n; k++)
      - for (p <=k; p <= n; p++) {O(1)}
        - T(n) = … → $O(n^4)$

Sequential search (I)

```
public static boolean search(int[] list, int value){
    int i = 0;
    int n = list.length - 1;
    while ((i <= n) && (list[i] != value))
        i++;
    if (i == n+1)
        return false;
    else
        return true;
}
```

- Best case:

SENTENCE	TIMES
i = 0	
n = list.length – 1	
i <= n	
list[i] != value	
i++	
i == n+1	
return false	
return true	

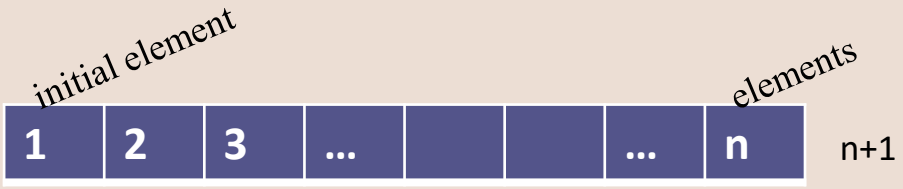


Sequential search (II)

```
public static boolean search(int[] list, int value){
    int i = 0;
    int n = list.length - 1;
    while ((i <= n) && (list[i] != value))
        i++;
    if (i == n+1)
        return false;
    else
        return true;
}
```

- Worst case:

SENTENCE	TIMES
i = 0	
n = list.length – 1	
i <= n	
list[i] != value	
i++	
i == n+1	
return false	
return true	



It does not exist

Sequential search (III)

$T(n)_{average} = \sum_{i=1}^{N_{cases}} P_i * Ti(n)$

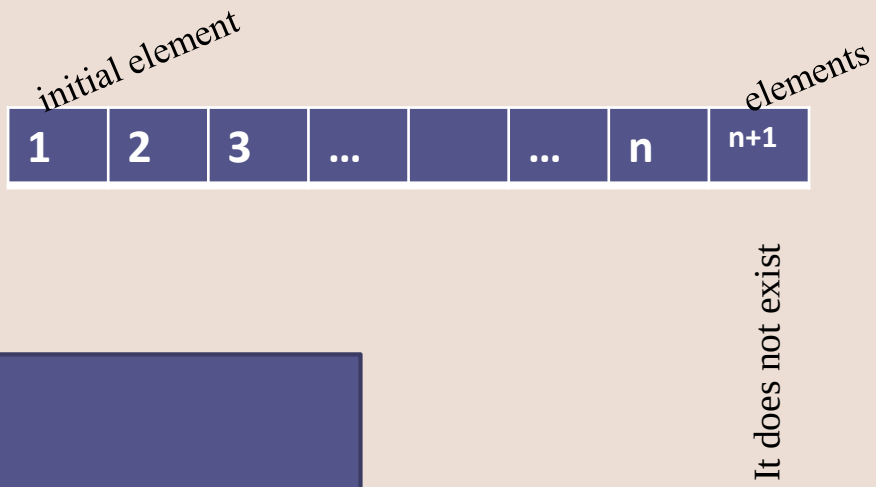
- Average case:
  - N_{cases} = n+1
  - To facilitate the calculation
    - P_{exist}= ½
    - $P_{donotexist}$ = ½
  - When the element exists, it is equally likely to appear in any position in the array:
    - $P_{element\ 1} = \frac{1}{2n}$, $P_{element\ 2} = \frac{1}{2n}$, …, $P_{element\ n} = \frac{1}{2n}$
  - $T(n) = \sum_{i=1}^{n+1} P_i T_i(n) = \frac{1}{2n}[6 + (6+3) + (6+3+3) + \cdots + (6 + 3(n -$

Sequential search with sentinel (I)

- Best case:

SENTENCE	TIMES
list.add(value)	
i = 0	
list.get(i) != value	
i++	
i==list.size()-1	
return false	
return true	

```
public static boolean search(List<Integer> list, int value){
    list.add(value); //we know now that the element is in the array
    int i = 0;
    while (list.get(i) != value)
        i++;
    if (i == list.size()-1) //element n+1
        return false;
    else
        return true;
}
```



Sequential search with sentinel (II)

```
public static boolean search(List<Integer> list, int value){
    list.add(value); //we know now that the element is in the array
    int i = 0;
    while (list.get(i) != value)
        i++;
    if (i == list.size()-1) //element n+1
        return false;
    else
        return true;
}
```

- Worst case:

SENTENCE	TIMES
list.add(value)	
i = 0	
list.get(i) != value	
i++	
i==list.size()-1	
return false	
return true	

It does not exist

Binary or dichotomous search (I)

```
public static boolean search(int[] list, int value){
    int left = 0;
    int right = list.length-1;
    int k;

    do {
        k = (left + right) / 2;
        if (value > list[k])
            left = k+1;
        else
            right = k-1;
    } while ((list[k] != value) && (left < right));

    if (list[k] != value)
        return false;
    else
        return true;
}
```

- Best case:

SENTENCE	TIMES
left = 0	
right = list.length-1	
k	
k= (left+right)/2	
value > list[k]	
left = k+1	
right = k-1	
list[k] != value	
left < right	
list[k] != value	
return false	
return true	

The array must be ordered

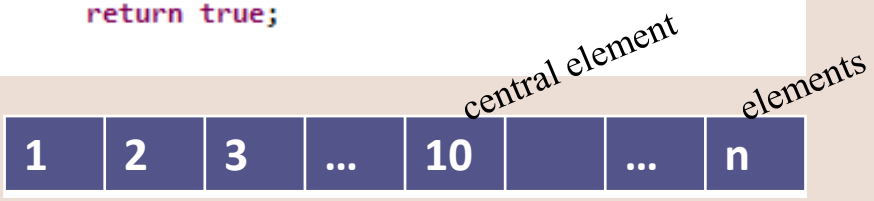
Binary or dichotomous search (II)

```
public static boolean search(int[] list, int value){
    int left = 0;
    int right = list.length-1;
    int k;

    do {
        k = (left + right) / 2;
        if (value > list[k])
            left = k+1;
        else
            right = k-1;
    } while ((list[k] != value) && (left < right));

    if (list[k] != value)
        return false;
    else
        return true;
}
```

- Worst case:

SENTENCE	TIMES
left = 0	1
right = list.length-1	1
k	1
k= (left+right)/2	$\log_2 n$
value > list[k]	$\log_2 n$
left = k+1	$\log_2 n$
right = k-1	
list[k] != value	$\log_2 n$
left < right	$\log_2 n$
list[k] != value	1
return false	1
return true	0

  - $T(n) = 5 + 5\log_2 n \rightarrow \underline{O(\log_2 n)}$
- Average case:
  - $\underline{O(\log_2 n)}$



The array must be ordered

Bibliography