ISTITUTO ITALIANO DI TECNOLOGIA
DATA SCIENCE AND COMPUTATION FACILITY

Introduction to Programming Languagues and Computing

Sergio Decherchi
Data Science and Computation Facility,
Fondazione Istituto Italiano di Tecnologia, Genova, Italy

Programming languages

Programming languages

- Programming languages are a set of non-ambiguous instructions used to communicate with a computer.
- Non-ambiguous = deterministic, not human language
- They are used to create software, applications, and other computer programs.
- Monolotic programs, API, WebService, Drivers.

Low-Level Programming Languages

- Low-level programming languages are closer to the machine code and are harder to read and write.

- They are often used to write operating systems, device drivers, and other system-level software.

- Examples of low-level programming languages include assembly language and machine language.

Machine language

- Machine language is the lowest-level programming language and is specific to a particular computer.
- It is written in binary code and is difficult for humans to read and write.
- Machine language is the language that the computer's processor understands directly.

Assembly Language

- Assembly language is a low-level programming language that is specific to a particular computer architecture (e.g. ARM, x86).
- It is often used to write operating systems and device drivers and is more efficient than high-level languages.
- Assembly language instructions correspond directly to machine instructions, but they are easier for humans to understand and write than machine code.

```
MOV eax, 3
MOV ebx, 4
ADD eax, ebx, ecx
```

High-Level Programming Languages

- High-level programming languages are designed to be easy to read and write for humans.
- They are often used to develop software applications and can be divided into several sub-categories, such as object-oriented, scripting, and functional languages.

Scripting Languages

- Scripting languages are used to automate tasks and to write small programs quickly.
- They are often used for web development, such as JavaScript, and for system administration tasks, such as shell scripting.
- Examples of scripting languages include JavaScript, Python, and Ruby.

Functional Programming Languages

- Functional programming languages are based on the concept of functions.
- They are often used for scientific and mathematical applications and can be used to write highly parallelized programs.
- Examples of functional programming languages include Haskell, Lisp, Julia.

Not just a language aspect, but also a programming style (JAX). We have functions in all programming languages.

Object-Oriented Programming Languages

- Object-oriented programming languages are based on the concept of objects and classes.

- They allow developers to create reusable code and to write programs that are easy to modify and maintain.

- Examples of object-oriented programming languages include Java, Python, and C++.

Compiled vs Interpreted Languages

- Programming languages can be divided into two categories based on how they are processed: compiled and interpreted.

- ➢ Compiled languages are translated into machine code before being run.
- ➢ Interpreted languages are executed line by line and interpreted line by line.

Each language has its own advantages and disadvantages, and the choice often depends on the needs of the project.

Compiled Languages

- Compiled languages are those that are translated into machine code before being executed.
- The source code is compiled into an executable program that is then run by the computer.
- This compilation process takes time and resources.
- Fast when executed, tailored for the specific machine.
- The compiler can detect syntactic errors during compilation.
- More difficult to debug
- Less portable, less flexible not good for fast prototipyng.

Interpreted Languages

- Interpreted languages are those that are executed line by line by the computer.
- The source code is read and interpreted by the computer and online translated into machine code.
- One avoids compilation, but the program is slower when executed.
- Interpreted languages can also be more prone to errors, as the interpreter may not catch all errors before the program is run.
- Interpreted languages can also be less secure, as the source code is often distributed with the program and can be read and modified by anyone.

The virtual machine

•A virtual machine is a software implementation of a physical machine that provides an environment for executing programs or running applications.

•It acts as an intermediate layer between the program and the underlying operating system, providing a platform-independent execution environment.

• VM enables "write once, run anywhere" functionality. Programs compiled to bytecode can be executed on any system with a compatible VM, regardless of the underlying hardware or operating system.

•The VM manages memory allocation and deallocation, including automatic garbage collection. It abstracts the complexities of memory management, making it easier for developers to focus on application logic.

•Performance Optimization: the JVM for example includes various optimization techniques, such as just-in-time (JIT) compilation, which dynamically compiles frequently executed bytecode into machine code for improved performance.

Programming statements

- A program is a text file in which at every line there is a <u>statement</u> or part of it

- It is composed of:
    - Variables (base types)
    - Data Structures
    - Control instructions
    - Functions
    - Modules

    ….

Variables

A variable is a named container that holds a datum or a reference to a datum .

1.Data Storage: variables hold different types of data, such as numbers, text, or complex structures like arrays or objects. They provide a way to store and retrieve information as the program runs.

2.Assignment and Manipulation: variables are created by assigning a value to them using the assignment operator (=). The value can be changed and manipulated throughout the program execution, allowing for dynamic data handling.

3.Data Types: variables have specific data types, such as integers, floating-point numbers, strings, or boolean values. The data type determines the kind of data the variable can hold and the operations that can be performed on it.

4.Basic types : integer, floating point, character, string.

5.Access: access by reference (pointers) or by copy.

6.Naming Conventions: variables have names assigned by the programmer, following certain naming conventions (e.g., avoiding reserved keywords, using descriptive names). The name represents an identifier to access the stored data.

Data Structures

A data structure is a way of organizing and storing data in a computer program. It provides a systematic and efficient approach to represent, access, and manipulate data.

1.Types of Data Structures: there are various types of data structures, including:

- **Arrays**: Collection of elements accessed by their index.
- **Lists**: Dynamic structures that allow adding or removing elements.
- **Stacks**: LIFO (Last-In-First-Out) structure for managing function calls or undo/redo operations.
- **Queues**: FIFO (First-In-First-Out) structure for managing waiting lists or process scheduling.
- **Trees**: Hierarchical structures with parent-child relationships.
- **Graphs**: Networks of interconnected nodes.
- **Hashtables**: Key-value pairs for efficient data retrieval.

2.Operations: data structures support different operations, such as insertion, deletion, searching, sorting, and traversal. The choice of data structure depends on the specific requirements of the program or problem at hand.

3.Algorithms and Data Structures: data structures often work in conjunction with algorithms. Algorithms define the steps and logic to perform specific tasks, while data structures provide the underlying organization and access methods for the data being manipulated.

Arrays

An array is a collection of elements of the same data type, grouped together under a single name. It provides a way to store and organize multiple values in a contiguous block of memory.

1.Indexing: each element in an array is assigned a unique index, starting from 0 (or 1) and incrementing by 1 for each subsequent element. Indexing allows us to access individual elements based on their position in the array.

2.Size and Length: arrays have a fixed size defined at the time of creation. The length or size of an array represents the number of elements it can hold. It's important to consider the size when working with arrays to prevent access errors.

3.Declaration and Initialization: arrays are declared by specifying the data type and the name of the array. They can be initialized with initial values during declaration or assigned values later in the program.

4.Accessing Elements: elements in an array can be accessed by their index using square brackets notation. For example, array_name[index] retrieves the value at the specified index.

5.Manipulating Array Elements: array elements can be modified by assigning new values to them. They can also be used in expressions and manipulated using various operations provided by the programming language.

Lists

A list is an ordered collection of elements, where each element has a specific position or index within the list. Lists can hold elements of different types, such as numbers, strings, or even other data structures.

1.Dynamic Size: lists can dynamically grow or shrink in size to accommodate varying numbers of elements. They provide flexibility when dealing with collections of unknown or changing lengths.

2.Element Access: elements in a list can be accessed using their index value. Indexing starts from 0 (or 1) for the first element and increments by 1 for each subsequent element.

3.Operations:

- **Adding Elements**: elements can be added to a list using methods like append(), insert(), or extend(). New elements can be added at the end, specific positions, or by combining multiple lists.
- **Removing Elements**: elements can be removed from a list using methods like remove() or pop(). Removal can be based on the element's value or index.
- **Modifying Elements**: elements in a list can be modified by directly assigning new values to specific indices.
- **Traversing Elements**: lists support iteration and looping, allowing easy access and manipulation of each element.

4.List Functions and Methods: programming languages often provide a set of built-in functions and methods specifically designed for list manipulation, such as sorting, counting, slicing, and searching.

Stack

A stack is an abstract data type that stores a collection of elements with two main operations: push and pop. It operates on the LIFO principle, where the last element inserted is the first one to be removed.

1.Stack Operations:

- Push: Inserts an element onto the top of the stack.
- Pop: Removes the topmost element from the stack.
- Peek/Top: Retrieves the topmost element without removing it.
- isEmpty: Checks if the stack is empty.
- Size: Returns the number of elements in the stack.

2.Visual Representation: Stacks can be visualized as a vertical structure, resembling a stack of plates. New elements are added to the top, and removal also occurs from the top.

3.Common Applications:

- Function Calls: Stacks are used to manage function calls in programming. Each function call gets added to the stack, and when a function completes, it is removed from the stack.
- Undo/Redo Operations: Stacks can be used to implement undo and redo functionality, where operations are pushed onto the stack and can be reversed by popping them.

4.Stack Implementation: Stacks can be implemented using arrays or linked lists. Arrays offer direct access to elements, while linked lists provide dynamic resizing.

Queue

A queue is an abstract data type that stores a collection of elements with two primary operations: enqueue and dequeue. It operates on the FIFO principle, where the first element inserted is the first one to be removed.

1.Queue Operations:

- Enqueue: Adds an element to the end of the queue.
- Dequeue: Removes the element from the front of the queue.
- Front/Peek: Retrieves the element at the front without removing it.
- isEmpty: Checks if the queue is empty.
- Size: Returns the number of elements in the queue.

2.Visual Representation: Queues can be visualized as a horizontal structure, resembling a line of people waiting. New elements are added to the rear, and removal occurs from the front.

3.Applications:

- Process Scheduling: Queues are used to manage processes or tasks in an operating system, ensuring fairness and order.
- Print Queue: Queues are used to manage print jobs in systems, processing them in the order they are received.
- Breadth-First Search: Queues are utilized in graph algorithms like breadth-first search to explore nodes in a level-by-level manner.

4.Queue Implementation: Queues can be implemented using arrays or linked lists. Arrays offer direct access to elements, while linked lists provide dynamic resizing.

Hashtable

A hashtable, also known as a hash map, is an abstract data type that stores elements as key-value pairs. It uses a hash function to map keys to indexes in an underlying array, enabling fast access and retrieval.

1.Key-Value Pairs: Each element in a hashtable consists of a unique key and its associated value. Keys are used to identify and locate specific values within the hashtable.

2.Hashing: Hashing is a process that converts keys into a unique numeric value, known as a hash code. The hash code is used as an index to store and retrieve values in the hashtable.

3.Collision Resolution: Collisions can occur when different keys produce the same hash code, resulting in overlapping indexes. Various collision resolution techniques are employed to handle collisions and ensure proper storage and retrieval of values.

4.Benefits of Hashtables:

- Fast Access: Hashtables provide fast retrieval of values based on their keys, typically with constant time complexity (O(1)).
- Efficient Storage: Hashtables optimize memory usage by dynamically adjusting their size to accommodate the number of stored elements.
- Key-Based Lookup: Hashtables allow direct access to values based on their keys, making them suitable for scenarios requiring efficient data retrieval.

5.Common Applications:

1. Caching: Hashtables are used for caching frequently accessed data to improve performance (LRU).
2. Dictionaries: Hashtables are employed to implement dictionary-like data structures for efficient lookup and retrieval of information.
3. Symbol Tables: Hashtables are utilized in compilers and interpreters to store variable names and their corresponding values.

Control instructions

Control instructions enable programmers to control the order and conditions under which code is executed. They allow for decision-making and looping, providing flexibility and control over program execution.

1.Conditional Control:

- **If Statement**: The if statement allows the program to execute a block of code based on a specified condition. It provides branching based on whether the condition evaluates to true or false.
- **If-Else Statement**: The if-else statement expands upon the if statement by providing an alternative block of code to execute when the condition is false.
- **Switch Statement:** The switch statement allows for multi-way branching based on the value of a variable or expression. It provides an alternative to multiple if-else statements.

2.Looping Control:

- For Loop: The for loop executes a block of code a specified number of times, iterating over a range of values or elements.
- While Loop: The while loop repeatedly executes a block of code as long as a given condition remains true.
- Do-While Loop: The do-while loop is similar to the while loop, but it executes the code block at least once before evaluating the condition.

1.Control Flow: Control instructions control the flow of program execution, determining which statements are executed and in what order. They enable the program to make decisions, repeat actions, or choose among different alternatives.

2.Code Blocks: Control instructions are typically used to define code blocks, which group multiple statements together. Code blocks are enclosed, for instance, within curly braces {} and provide a scope for local variables.

Functions

A function is a reusable block of code that performs a specific task or calculation. It takes input parameters, performs operations, and optionally returns a result.

1.Modularity and Reusability: Functions promote modularity and reusability by breaking down complex tasks into smaller, manageable pieces. They can be called from different parts of a program, eliminating code duplication and improving maintainability.

2.Function Signature: A function has a unique name and a signature that specifies its parameters and return type (if any). The signature defines how the function can be invoked and what it produces.

3.Function Components:

- Parameters: Functions can accept zero or more input parameters that provide values or data for the function to work with.
- Body: The body of the function contains the set of instructions to be executed when the function is called. It defines the logic and operations to perform the desired task.
- Return Statement: Functions can optionally return a value or result using the return statement. The returned value can be used in further computations or assigned to a variable.

4.Function Invocation: Functions are called or invoked by their name, providing the necessary arguments (if any). The function executes its code, potentially modifying data or producing a result.

5.Standard Library Functions: Programming languages provide a standard library that includes pre-defined functions for common tasks, such as mathematical calculations, string manipulation, input/output operations, and more

6.User-Defined Functions: Programmers can create their own custom functions to solve specific problems or encapsulate frequently used code. User-defined functions enhance code organization and readability.

Modules

A module is a self-contained unit of code that encapsulates functions, and variables. It provides a way to organize and group code logically.

1.Code Organization: Modules help in organizing code by grouping related functionality together. They allow developers to break down large programs into smaller, manageable units.

2.Encapsulation: Modules encapsulate code, hiding implementation details and exposing only necessary interfaces or functions. This promotes information hiding and improves code maintainability.

3.Reusability: Modules facilitate code reuse by allowing functions, classes, or variables to be imported and used in other parts of a program. They promote modular programming and reduce code duplication.

4.Namespace: Modules create a separate namespace, preventing naming conflicts between different modules. This enables programmers to use similar names for variables or functions without collision.

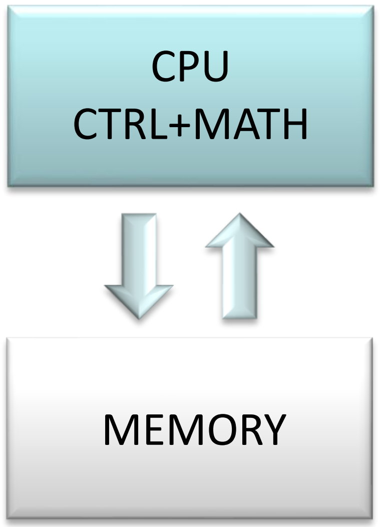
5.Standard Library Modules: Programming languages provide a standard library with a collection of pre-built modules for various common tasks, such as file handling, networking, mathematics, and more.

6.Custom Modules: Programmers can create their own custom modules to encapsulate and reuse code specific to their applications. These modules can be shared among projects or with other developers.

7.Module Import: Modules are imported into programs using import statements. Once imported, their functions, classes, and variables can be accessed and utilized.

How a computer works

The Von Neumann architecture and ISA



I.S.A **: Instruction set architecture** (CISC,RISC)

x86, ARM, EPIC (Itanium), Z80

x86 extended sets: MMX, SSE, SSE2, SSE4.2, AVX, AVX2, AVX512 to accelerate integer and floating point operations.

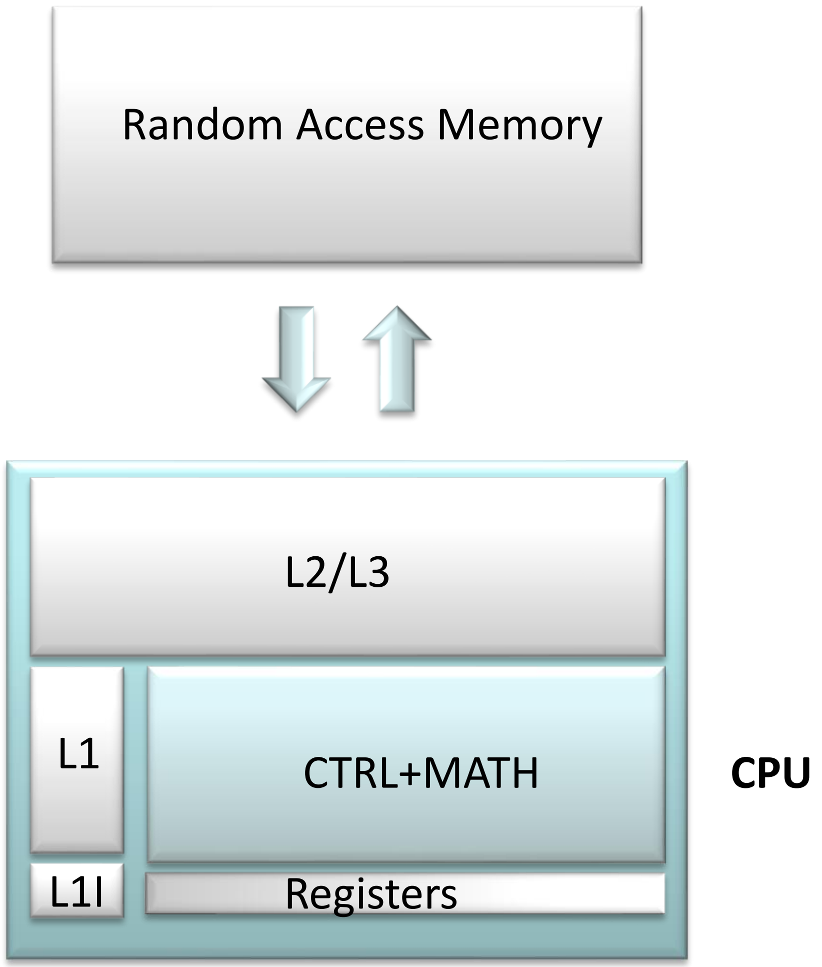
Python/C instructions are split in simpler ones. Or sometimes grouped for performance reasons.

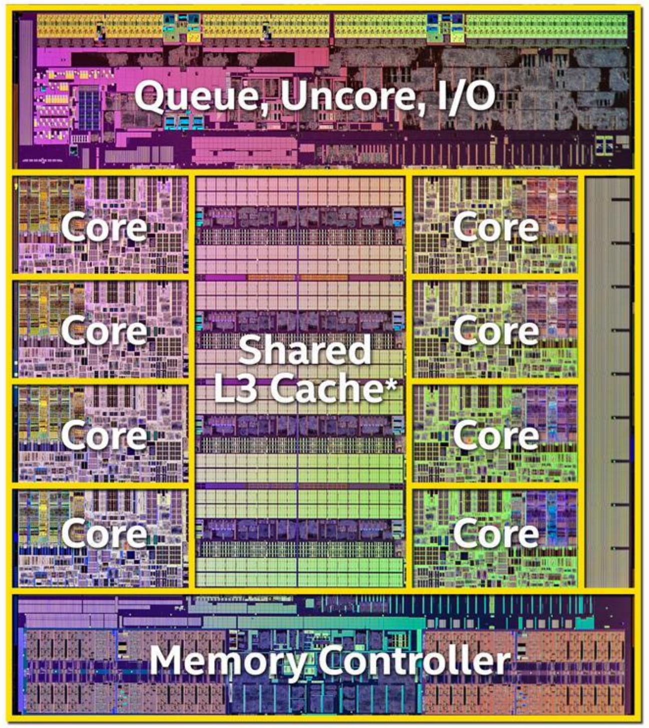
Easy examples are: load number, load@address, jump at memory location ....

Complex example MMX instruction for adding packed unsigned byte integers and saturate.

PADDUSB mm, mm/m64

Memory Hierarchy





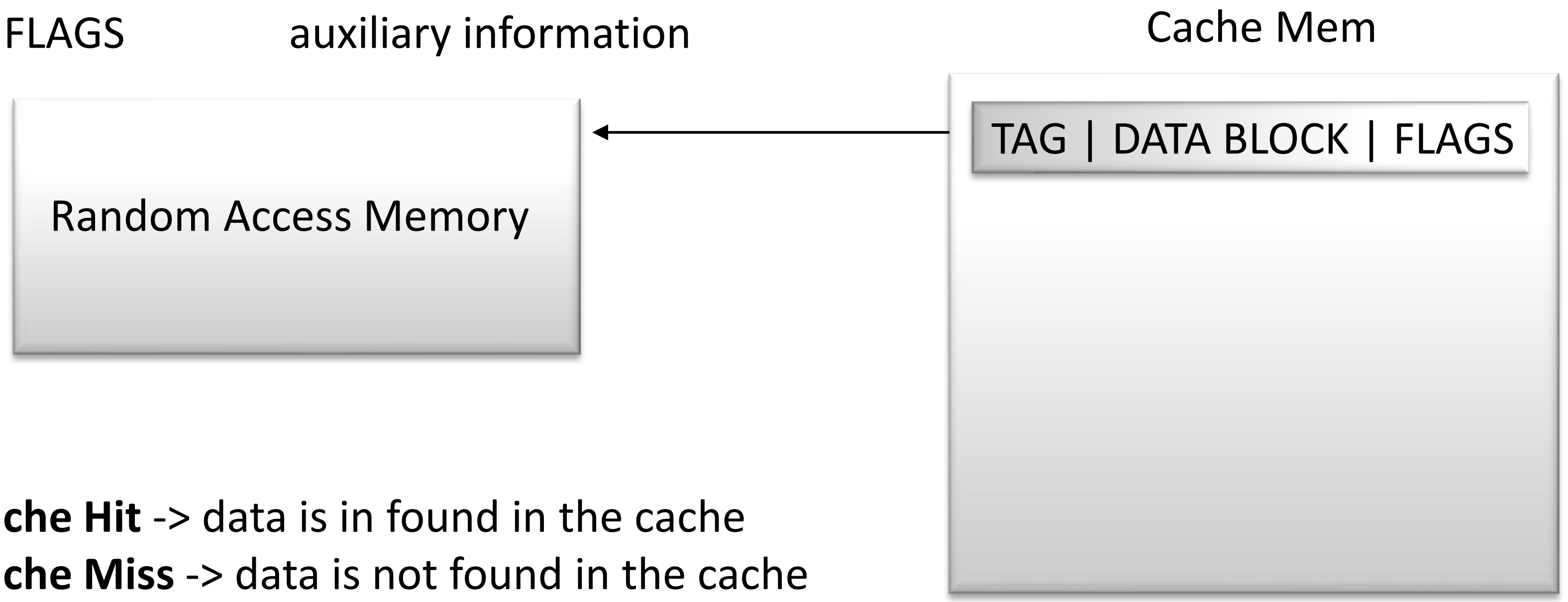
Read/Write Speed for i9-9900K		**Size**
RAM	47.6 Gb/s	16 Gb
L3	367 Gb/s	16 Mb
L2	890 Gb/s	256Kb (core)
L1	2344 Tb/s	64Kb (core)
Registry	Usually 3x L1	few Kb

Inside the CPU: the cache

It leverages the **temporal and physical locality of data**

- It moves data in cache **blocks** (spatial locality)
- **LRU**: Least Recently Used discard policy (temporal locality)

TAG	points to the index in RAM
DATA BLOCK	the actual data
FLAGS	auxiliary information

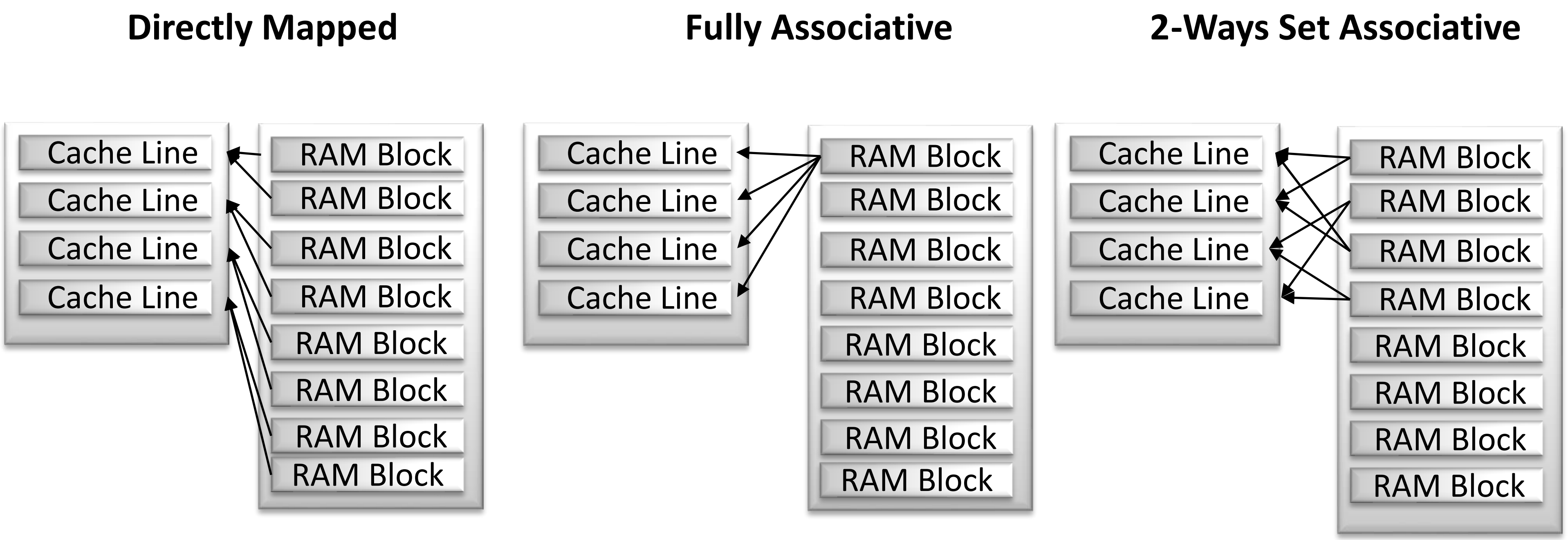


Cache Hit -> data is in found in the cache
Cache Miss -> data is not found in the cache

Code optimization hint = Recycle data as much as possible, temporalily and spatially

Inside the CPU: the cache

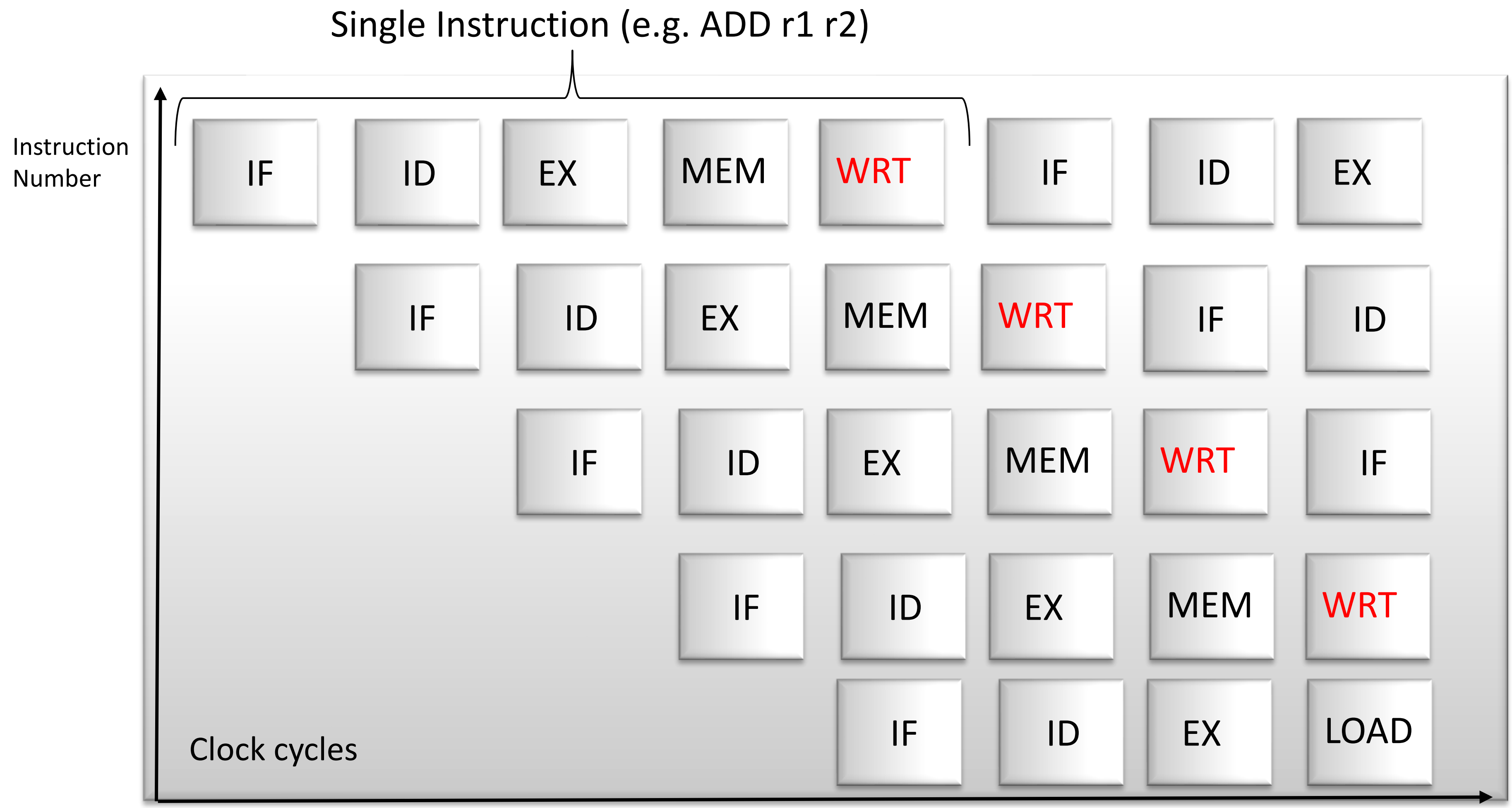
The physical memory is not accessed directly but through virtual pages (**MMU**).



Many specialized caches: Victim cache (L4 of Crystalwell), Trace Cache (P4, L1, Micro-op cache), Branch Target Cache (ARM), Scratchpad memory (Athlon 64), Translation Lookaside Buffer (TLB, part of MMU), GPU (CUDA shared, OpenCL local) on-chip memory.

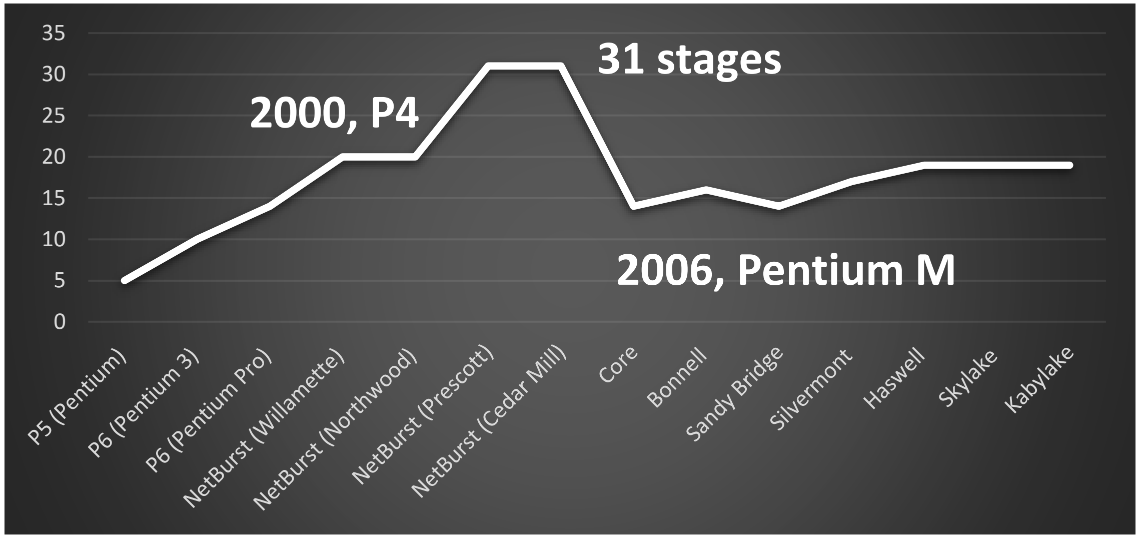
Inside the CPU: the pipeline[s]

Instruction level parallelism.
At regime, for each clock cycle one instruction is executed.



Inside the CPU: the pipeline[s]

The smaller the sub-instruction step the higher the frequency of the processor.
The pipeline mechanism assumes that the instructions are known. **Branching can destroy the pipe** and a **pipe flush** is expensive and the pipeline "stalls".



Hardware solutions:
- Try to **predict the branch result**. If you can predict the pipe does not stall
- Speculate execution: perform both **branches in parallel** than discard the wrong choice.

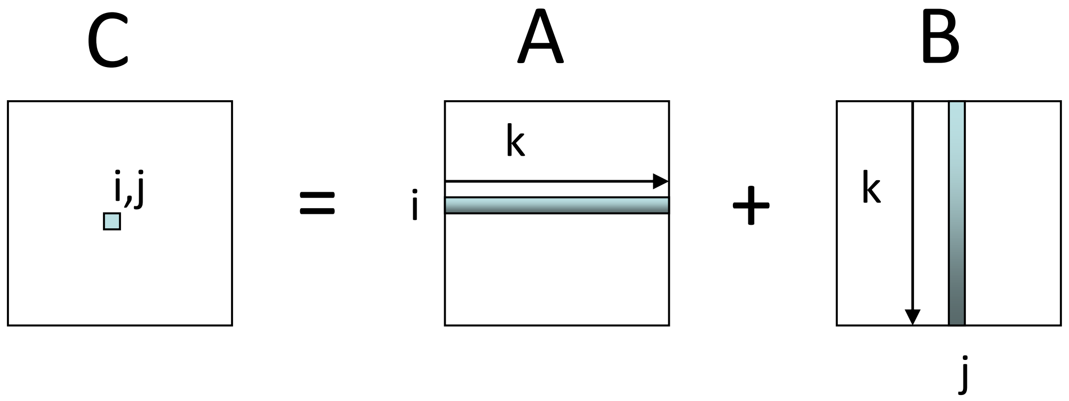
Software optimization solution: **Minimize the number of "IF" conditions**, they stall the pipeline. Use math operations to "mimick" IF conditions.

Matrix Multiplication

$C = AB$

A,B,C are n x n

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$



Space (Memory) Complexity: O(n^2)
Time Complexity : O(n^3)

Performances

Version -1 (naïve, no opt)
Exec time = **47.389** [s]

Version 0 (naïve+O3)
Exec time = **31.068** [s]

Version 1 (contiguous memory+O3)
[loop vectorizaiotn and loop unrolling takes place from here on]
Exec time = **9.374** [s]

Version 2 (cache friendly+O3)
Exec time = **1.739** [s]

Version 3 (blocks+O3)
Exec time = **1.172** [s]

Version 3.1 (block+march=native)
Exec time = **0.874** [s]

- Compile with opt report: **-fopt-info-vec-optimized, -fopt-info-loop-optimized** (g++ 7.5)

NetlibBLAS, OpenBLAS

Version 2 (cache friendly)
Exec time = **1.7** [s]

Version 3.1 (block+native opts)
Exec time = **0.874** [s]

Version 4 (NetlibBLAS, fortran, 1 thread)
Exec time = **1.7** [s]

Version 4 (OpenBLAS 1 thread)
Exec time = **0.154** [s]
Speed-up is **5-6x**. **SIMD 256/32=8x**

Version 4 (OpenBLAS 4 threads)
Exec time = **0.040** [s]
Further Speed-up is about 4. 4 Cores.

Wrt to the very naive code is **775x** faster (**201x**, 1 thread).

Algorithms

Algorithms

An algorithm is a step-by-step set of instructions or procedures used to solve a problem or perform a specific task.

1. Definition: an algorithm is a precise and unambiguous description of how to solve a problem or accomplish a task. It provides a clear sequence of steps to be followed.

2. Problem Solving: algorithms are used to solve complex problems by breaking them down into smaller, more manageable steps. They help in finding the most efficient and effective solutions.

3. Logical Sequence: algorithms are designed to be logical and sequential, ensuring that each step follows logically from the previous one. They take into account different conditions and possible outcomes.

4. Reproducibility: algorithms are designed to be reproducible. Given the same inputs, an algorithm will always produce the same output, ensuring consistency and reliability.

Computational algorithms

- **Correctness** (physics, chemistry, engineering etc..) **[Paper, low TRL]**
- **Stability**: **[Usable, middle-TRL]**
  - It should work when the input changes in the same class of problems.
  - It should work when you change the class of problems.
  - It has to converge in a finite number of steps (**conditioning**).
  - Ideally, it should work using just **single precision**. If your algorithm needs double precision this should ring a bell. **Can one use mixed precision**?
- **Scalability**: **[Real-world, middle/high-TRL]**
  - **O**(n) -> Efficient
  - **O**(n^2) -> Good in some domains, unacceptable on others (naive Support Vector Machines Training, Coulomb, Distance)
  - **O**(n^3) -> Slow in any domain but one can use it (e.g. Gaussian Elimination, Singular Value Decomposition)
  - > **O**(n^4) -> Slow in any domain, generally not desirable (e.g. Semi-definite programming).
- **Universality, elegance***: where possible it should be conceptually customizable
- **Inherently parallel**: ***It is better to have a slower parallalizable algorithm than a super-fast inherently sequential algorithm***

*** D. Knuth, "The art of computer programming" inventor of TeX**

Pseudo-code

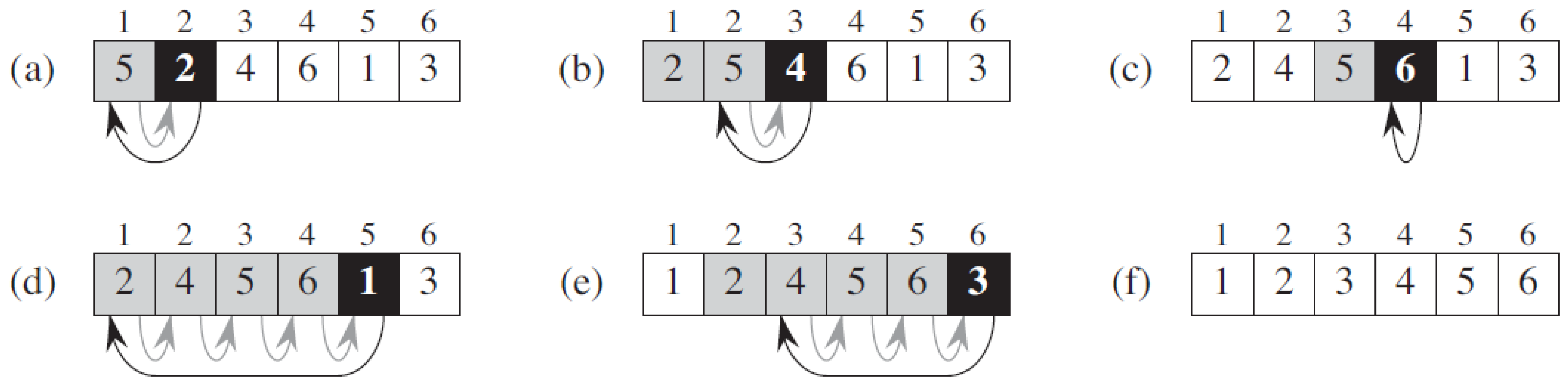
- Algorithms are universal.
- Any language can implement any algorithm (Turing complete).
- Not any language can implement cleanly and efficiently any algorithm!
- Pseudo-code as *lingua franca* for programming

```
for i = 1 to 100
    A[i]=i
    print(A[i])
end
```

Insertion sort

A first example of in-place algorithm in pseudo-code

```
INSERTION-SORT(A)
1  for $j = 2$ to $A.length$
2      $key = A[j]$
3      // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
4      $i = j - 1$
5      while $i > 0$ and $A[i] > key$
6          $A[i+1] = A[i]$
7          $i = i - 1$
8      $A[i+1] = key$
```



(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3
(e) 1 2 3 4 5 6 | 1 2 4 5 6 3
(f) 1 2 3 4 5 6 | 1 2 3 4 5 6

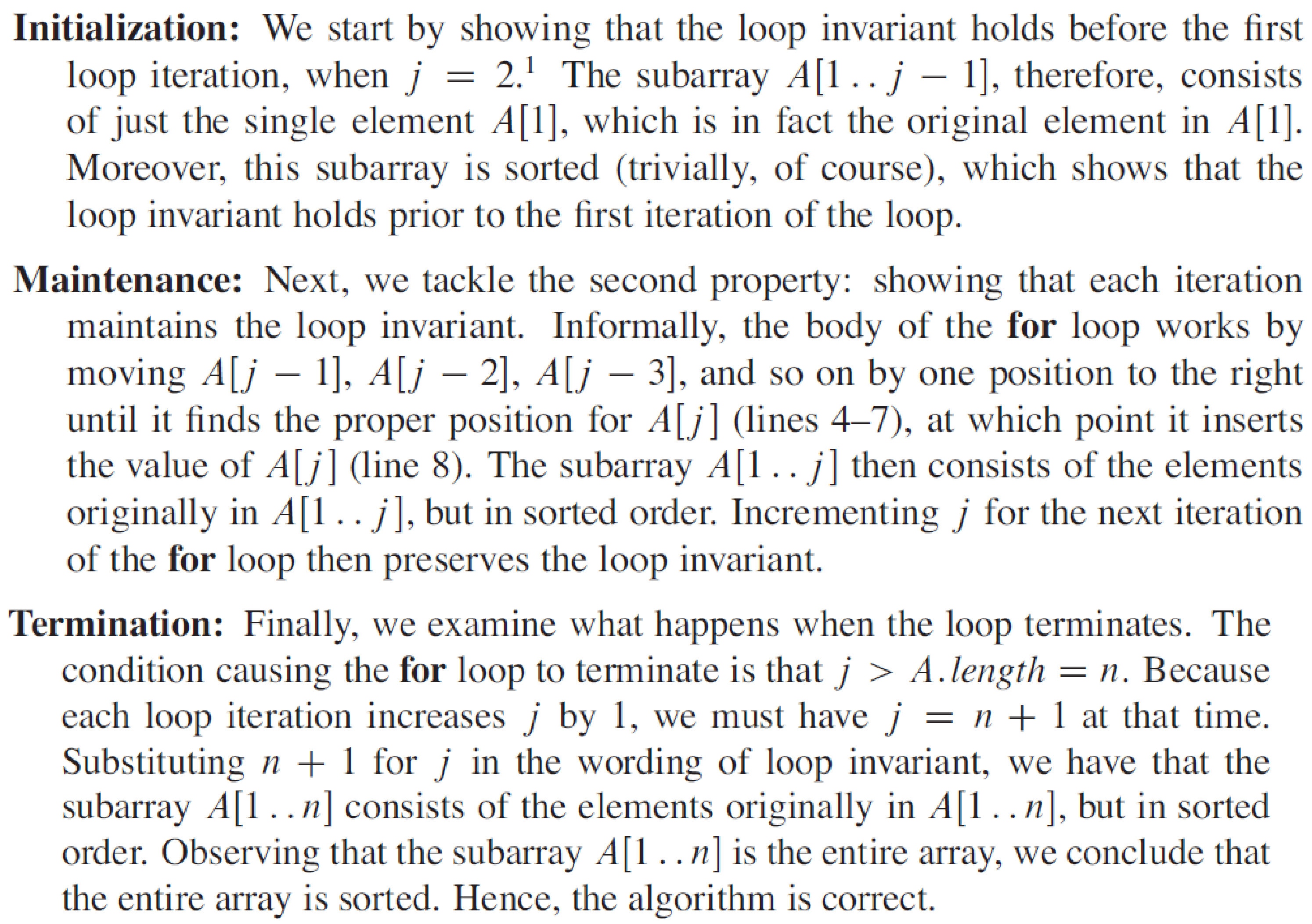
Insertion sort

Is this algorithm correct?
How to prove it ?

We can prove it through a "loop invariant". We have to check it is correct at three stages:

- Initialization
- Maintenance
- Termination

Insertion sort



Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1 \,..\, j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 \,..\, j]$ then consists of the elements originally in $A[1 \,..\, j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.\mathit{length} = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order. Observing that the subarray $A[1 \,..\, n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Insertion sort

Is this algorithm "fast"?

```
INSERTION-SORT(A)                                   cost     times
1  for $j = 2$ to $A.length$                        $c_1$    $n$
2      $key = A[j]$                                 $c_2$    $n-1$
3      // Insert $A[j]$ into the sorted
           sequence $A[1..j-1]$.                    $0$      $n-1$
4      $i = j - 1$                                  $c_4$    $n-1$
5      while $i > 0$ and $A[i] > key$               $c_5$    $\sum_{j=2}^{n} t_j$
6          $A[i+1] = A[i]$                          $c_6$    $\sum_{j=2}^{n} (t_j - 1)$
7          $i = i - 1$                              $c_7$    $\sum_{j=2}^{n} (t_j - 1)$
8      $A[i+1] = key$                               $c_8$    $n-1$
```

$$
\begin{aligned}
T(n) \quad = \quad & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \,.
\end{aligned}
$$

Insertion sort

Is this algorithm "fast"?

Best case O(n):

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned}$$

Worst case O(n^2):

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned}$$

Gradient descent

Gradient descent is an iterative optimization algorithm used to find the optimal values of the parameters in a model by minimizing the cost function.

1.Cost Function: In machine learning, a cost function measures the difference between the predicted values of the model and the actual values. The goal of gradient descent is to find the parameter values that minimize the cost function.

2.Gradient: The gradient is a vector of partial derivatives of the cost function with respect to each parameter. It represents the direction of steepest ascent. In gradient descent, the negative gradient is used to move in the direction of steepest descent.

Gradient descent

1.Iterative Process:

1. Initialization: The algorithm starts with an initial set of parameter values.
2. Compute Gradient: The gradient is computed by taking partial derivatives of the cost function with respect to each parameter.
3. Update Parameters: The parameters are updated by taking a small step in the direction opposite to the gradient.
4. Repeat: Steps 2 and 3 are repeated until convergence, where the cost function is minimized or a predefined stopping criterion is met.

2.Learning Rate: The learning rate determines the step size in each iteration. It controls how quickly or slowly the algorithm converges. A larger learning rate may converge faster but risk overshooting the optimal values, while a smaller learning rate may take longer to converge.

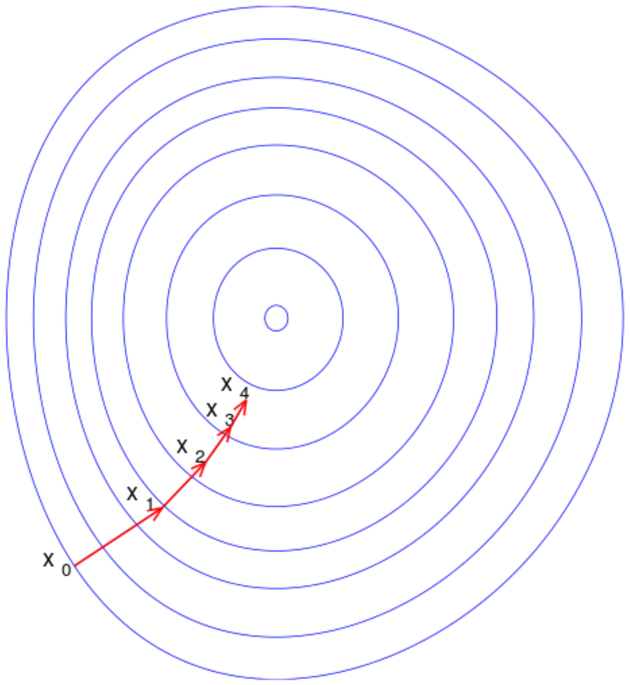
3.Variants of Gradient Descent:

- Batch Gradient Descent: Computes the gradient using the entire training dataset in each iteration.
- Stochastic Gradient Descent: Computes the gradient using only a single training example at a time, making it faster but more noisy.
- Mini-Batch Gradient Descent: Computes the gradient using a subset (mini-batch) of the training dataset, striking a balance between batch and stochastic gradient descent.

Gradient descent

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n)$$

- Convergence can be proved under mild conditions! It is a theorem.
- How many iterations??


x 0
x 1
x 2
x 3
x 4