Introduction

Checking Knowledge and evaluation

- Third part of USI-Algo-Prog teaching unit (UE) - approx. 3ECTs
- 1 DS and 1 exam - 1 Test not planned in advance.
- Theorical point of view of programming activities.
- Applications and implementations in Python

History

Before availability of modern computer machine :

- Coming from works of a mathematician : Al Khwarismi (780-850).
- Algebraic calculus roots.
- **1834** : Charles Babbage publish the first description of a programmable machine.
- **1815-1852** : the first informatic program is written by Ada Lovelace.
- **1934** : Turing machine (`http://zanotti.univ-tln.fr/turing/`)

Algorithms

- Donald Knuth, fundation of programmation *“The Art of Computer Programming”*.
- Author of TeX and Metafont.
- Algorithm of *Knuth-Morris-Pratt* (1977): pattern research in string - used in genome sequence processing.
- New architectures: recursive, parallel or quantum algorithmics.

Computer Architecture

- At its core, two components :
  1. **C**entral **P**rocessing **U**nit
  2. Memory storage
- CPU allows you to read, to write and to modify the memory space.
- Data are 0 or 1 value - discrete/boolean.

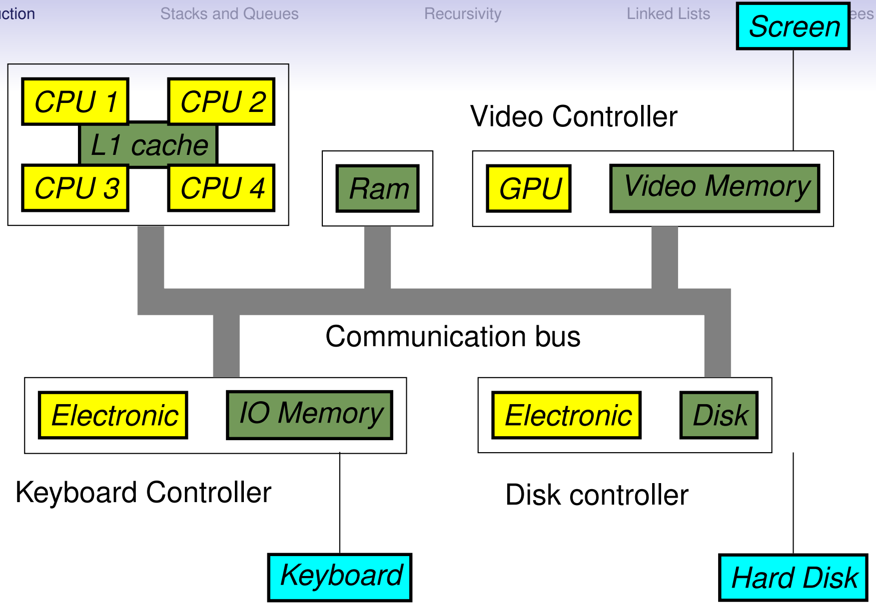


Figure : Computer Architecture

Algorithm Principles

- An algorithm is step-by-step set of operations, or instructions, which solve a problem.
- Algorithmics is a set of rules and methods used to define and to design algorithms.

Complexity

- Efficiency of algorithms are measured by their complexity, that is to say the memory and time cost.
- For an algorithm which processes data of size `n`, the complexity is given :
  - **in time** : by the number of elementary operations used to process the data
  - **in size** : by the necessary memory space required to store the data

Complexity (2/2)

- Algorithm study is concerned with the **order of magnitude** of the complexity
- Elementary operations such as affectations, addition, multiplications, and conditional statements are considered to have an execution time of 1, denoted by $O(1)$ or $\Omega(1)$.
- Three classes of complexity : best-, worst- and average-case complexities.

Abstract Data Types

- An abstract data type is a data type along with all the operations which are allowed on the data of this type
- In other words, an abstract type is defined by:
  1. a name,
  2. some **data**, *i.e.* a set of values,
  3. a set of functions, named **primitives**, operating on this data.

Stacks and Queues

Stacks and Queues

Outline :

1. Definitions.
2. Stack primitives and examples.
3. Queue primitives and examples.
4. Implementation of stacks.
5. Implementation of queues.

Stack Definition

- Stacks and queues are containers where only one object is accessible at each time.
- **Stack**
    - In a stack, the accessible object is the last inserted one.
    - LIFO: last in - first out.
    - Example: a stack of plates.
- **Queue**
    - In a queue, the accessible object is the first inserted one.
    - FIFO: first in - first out.
    - Example: a queue to buy a cinema ticket.

Primitives of Stack

- Two operations are defined as stack **primitives**:
  - `push(P,e)` to add an element `e` in `P`.
  - `pop(P)` to suppress an element in `P`.
- *Property*: if an element `f` is pushed after an element `e` then the element `f` is accessible before `e`.

Stack Implementation

```
def create_stack( ):
    return []

def push( theStack, element ):
    theStack.append( element )

def pop( theStack ):
    return theStack.pop()
```

- From `wiki.python.org/moin/TimeComplexity`, the only one operation which is not in $O(1)$ is `pop()`.
- Complexities of `create_stack()` and `push(theStack, element)` are in $O(1)$.
- Complexity of `pop(theStack)` is in $O(n)$, where n is the number of element into the stack.

Stack Implementation

To implement a stack with all operations in $O(1)$, it is necessary to use the class `deque` belonging to the module `collections`:

```
import collections

def create_stack( ):
    return collections.deque()

def push( theStack, element ):
    theStack.append( element )

def pop( theStack ):
    return theStack.pop()
```

Queue

- A Queue *F* is a data structures containing elements and having 2 operations :
  - `enqueue(F,e)` which adds the element *e* into *F*;
  - `dequeue(F)` which suppresses an element from *F*;
- *Property*: if an element `f` is added after an element `e`, then the element `e` is accessible before `f`.

Queue Implementation

```
def create_queue( ):
    return []

def enqueue( theQueue, element ):
    theQueue.append( element )

def dequeue(theQueue ):
    return theQueue.pop(0)
```

- From `wiki.python.org/moin/TimeComplexity`, `pop()` is the only one operation which is not in $O(1)$.
- Complexities of `create_queue()` and `enqueue(theQueue, element)` are in $O(1)$
- `dequeue(theQueue)` is in $O(n)$, where n is the number of elements into the queue.

Queue Implementation

- If we want to implement a queue with all operations are assumed as in $O(1)$, we have to use the module *deque*:

```
import collections

def create_queue( ):
    return collections.deque()

def enqueue(theQueue, element ):
    theQueue.append( element )

def dequeue( theQueue ):
    return theQueue.popleft()
```

Recursivity

Recursivity and everyday-life

Something that repeats itself within itself.

- In different forms of Art : *mise en abyme*, fractals :

(a)

(b)

Figure

- In computer science : when a function calls itself.

Call stack

- Call stack:
  - The sequence of instructions in a function are stored in a specific memory place called the **call stack**.
  - The stack is dedicated to a program.
  - When a function is called, all the variables, instructions and return values are pushed in the stack.
  - At the end of the function, only the return value is kept on the stack, all the other elements are popped from the stack.
  - The program resumes from the last instruction in the main program (or function).

Call stack

Consider the following program :

```
def g( a );
    print("g(" + str(a) + ")")
    return 1000

def h( a );
    print("h(" + str(a) + ")")
    return 2000

def f( a ):
    print("f(" + str(a) + ")")
    v = g(a+1)
    print( v )
    v = h(a+2)
    print( v )

f( 1 )
print("fin")
```

The following result is displayed on the terminal :

```
f(1)
g(2)
1000
h(3)
2000
fin
```

Your turn : Draw each step in the call stack resulting from the call to `f(1)`.

Visualizing the stack in Python

Try this Python program

```
import inspect

def f( a, b ):
    g(a+1, b+1)

def g( c, d ):
    print( "the stack is : " )
    print( inspect.stack() )
    print( "" )

    print( "Data of the function situated at the top of the stack are :  " )
    print( inspect.stack()[0] )
    print( "" )

    print( "unction variables at the top of the stack are :  " )
    print( inspect.getargvalues( inspect.stack()[0][0] ) )

f( 1, 2 )
```

Visualizing the stack in Python

The following result is obtained when you run the program :

```
the stack is  :
[
  (<frame object at 0x7f59363a6938>, 'prog.py', 8, 'g',['␣␣␣␣print(␣inspect.stack()␣)\n'], 0)
  (<frame object at 0x7f59363a93f0>, 'prog.py', 4, 'f',['␣␣␣␣g(a+1,␣b+1)\n'], 0),
  (<frame object at 0x7f59364f6050>, 'prog.py', 18, '<module>',['f(␣1,␣2␣)\n'], 0)
]

Data of the function situated at the top of the stack are :
(<frame object at 0x7f59363a6938>, 'prog.py', 12, 'g',['␣␣␣␣print(␣inspect.stack()[0]␣)\n'],

Function variables at the top of the stack are :
ArgInfo(args=['c', 'd'], varargs=None, keywords=None,locals={'c': 2, 'd': 3})
```

Recursivity

- **Definition**
  - If an algorithm - a function - calls itself, it is called **recursive**.
  - In other words, a function is recursive if it is found **at least twice** in the call stack during its execution.
- **Complexity**
  - A recursive algorithm needs to store the calling context of each call.
  - Recursivity can lead to a memory complexity bigger than iterative algorithm.

Recursivity : example

- It is possible to write a recursive power function : $pow(x, n) = x^n$.
- In Python :

```
def power(x, n):
    if (n == 0):
        return 1
    if (n == 1):
        return x
    return power(x, n-1) * x
```

Linked Lists

Linked Lists

Description

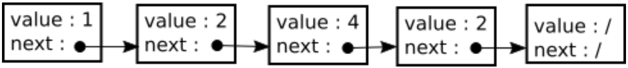
- Data structures which contain elements - could be identical. Each element is nested into a `cell`.
- The `cell` contains the value and the way to access to the next one. This is called a **linked** list.

Definition

- In the list `l` the element sequence $l_1, l_2, ..., l_n$ are nested into $c_1, c_2, ..., c_n$ cells.
- One more `cell`, c_{n+1}, is used to specify the end of the list.
- Even two elements are similar, the cells nesting them are different.
- For each `cell`,c_i, we have a value and a reference to the `next` cell if $i \leq n$ or `None` else.

Linked Lists

- The list is empty if it contains only the end of list `cell`.
- This is the schema of the list [1,2,4,2]



Primitives of Linked Lists

- The linked list type provides the following functions.

```
create_list(l)
push_front(l, e)
first( l )
end( l )
next( l, cell )
value( cell )
```

where *l* is a list, *e* an element of the list and *cell* a cell.

Primitives of Linked Lists

- *create_liste* create an empty list with an end list cell.;
- *push_front* add 1 to the size of the list and add the element e at the beginning of the list;
- Functions *first*, *end* and *next* return cells;
- Function $end(l)$ returns the cell at the end of the list c_{n+1};
- Function $first(l)$ returns the first cell c_1 belonging to the list;
- For each integer $i \in [1, n]$ the function $next(l, c_i)$ returns the successor of c_i which is c_{i+1} and None else;
- For each integer $i \in [1, n]$, the function $value(c_i)$ returns the value of c_i which is l_i;

Primitives of Linked Lists

- In Python

```
def create_list():
    return {'first':None, 'end': None }
def add_to_beginning(l , e):
    l['first'] = { 'value':e, 'next':l['first'] }
def next(l, e):
    return e['next']
def first(l):
    return l['first']
def end(l):
    return l['end']
def value(l, e):
    return e['value']
```

Primitives of Linked Lists

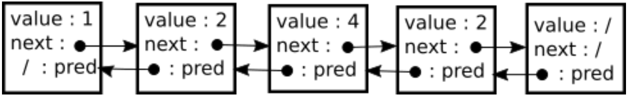
- List can be used as it :

```
l = create_list()
add_to_beginning(l, 3)
add_to_beginning(l, 5)
add_to_beginning(l, 2)

it = first(l)
while( it != end(l) ):
    print( value(l, it) )
    it = next(l, it)
```

Double Linked List

- A double linked list is a linked list with an additional information : the `precursor`.
- `Precursor` is a reference to c_{i-1} if $2 \geq i \geq n+1$, or to the end list cell c_{n+1} if $i = 1$ or to *None* else.
- The liste $[1, 2, 4, 2]$ could be display as :



Double Linked List

- Two primitives functions are defined for the list *l* and the cell *cell*:

```
push_back(l, e)
prev( l, cell )
```

- *push_back* add 1 to the list size and the element *e* at the end of the list.
- For all integers $i \in [2, n+1]$ the function $prev(l, c_i)$ returns the precursor of c_i which is c_{i-1}.
- If $i = 1$, then $prev(l, c_i)$ returns c_{n+1};
- For all integers $i \in [1, n]$, the function $value(c_i)$ returns the value of c_i which is l_i;

Complexity

- Evaluation of time and memory space to execute a program.
- Mathematic formula to obtain it from initial data and executing time (and memory).
- Caution : difficult to obtain in practice.
- Suppose the time of each operation is constant and counting how many operations are done for each program.
- Notation : complexity of the function *fct* is noticed $\mathcal{C}(fct)$

Complexity - Example

- The following program :

```
def fct( n ):             # initialisation de n : 1 opération
    r = 0                 # 1 opération
    for i in range(n):    # n*( incrément + corps de boucle ) = n*( 1 + 2 )
        r = r + 1         # | 2 operations
    return r              # 1 opération
```

realize $3 + 3n$ operations, its complexity is $\mathcal{C}(fct) = 3 + 3n$.

Complexity - Example

```
def max( T ):                  # init. : 1 opération
    res = T[0]                 # 1 opération
    for i in range( len(T) ):  # C1
        if res < T[i] :        # | C2
            res = T[i]         # | | 2 operations
    return res                 # 1 opération
```

- Let n the size of the array T.
- Complexity $C2$ of the conditional jump `if` is equal to 3 operations plus the operations of the `if` core.
- The core of `if` adds 2 or 0 supplementary operations depending on the test value.
- Result : $3 \leq C2 \leq 5$.

Complexity - Example

- Complexity of $C1$ - `for` loop - is

$$n \times (\text{ incrément } + \text{loop core })$$

Then :

$$C1 = \begin{cases} n(1+3) & \text{if false;} \\ n(1+5) & \text{if true.} \end{cases}$$

Program complexity is :

$$3 + 4n \leq \mathcal{C}(max) \leq 3 + 6n.$$

- The lower bound is reached if the element is at the first position in the array.
- The upper bound is reached if all element are distint and sorted by ascendant order in the array.

Complexity - Example

- Another stupid implementation of the program :

```
def max2( T ):                         # init. : 1 opération — on pose n= len(T)
    T = list( T )                      # = + Copie de tableau : 1+n opérations
    for i in range( len(T) ):          # C1
        for j in range( len(T) ):      # | C2
            if T[i] < T[j]:            # | | C3
                T[i] = T[j]            # | | | 3 opérations
    return T[ len(T)-1 ]               # 3 opérations
```

By analogy to the previous one, one can compute :

$$
\begin{gathered}
3 \leq C3 \leq 6, \\
n(1+3) \leq C2 \leq n(1+6), \\
n(1+4n) \leq C1 \leq n(1+7n).
\end{gathered}
$$

The complexity is :

$$5 + 2n + 4n^2 \leq \mathcal{C}(max2) \leq 5 + 2n + 7n^2.$$

Trees

Trees

- A tree is a data structure *A* containing elements named `vertex`.
- Primitive functions are :
  - `Child(A,p)` return the list of vertex accessible from `p`.
  - `parent(A, f)` return the vertex of `A` which is the ascendant of `f` or None.
  - `root(A)` return the vertex of `A` which is the root of the tree.

Tree Properties

- A root has no parent : $parent(A, root(A)) = None$
- All vertex, except the root, have a parent (and only one) :

$$\forall s \in A \subset \{root(A)\},\ parent(A, s) \in A$$

- From each vertex, root is accessible by using parent relationship.

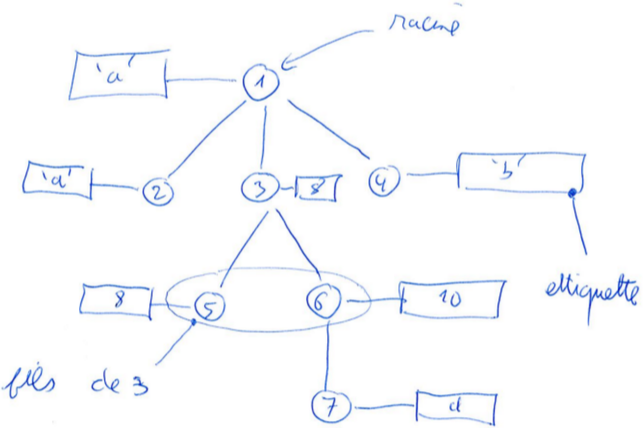
$$\forall s \in A, \exists k \in \mathbb{N}, parent^k(A, s) = None;$$

 where $parent^k$ is recursively defined by : $parent^0(A, s) = s$ and $parent^{k+1}(A, s) = parent(A, parent^k(A, s))$ for $k \geq 1$.
- The childs of a vertex s have as s as parent:

$$\forall s \in A,\ \forall f \in child(A, s),\ parent(A, f) = s.$$

Labeled Trees

- A labeled tree is a tree A with one operation $label(A, s)$ which returns an element named label of s for each vertex.
- It is possible to draw a tree like that :



Binary Trees

- A binary tree is a data structure A containing elements named *vertex*.
- The primitive functions are :
  - $left\_child(A, p)$ takes as parameters a tree A, a vertex p and returns a vertex of A named left child of p or *None* if p has no left child.
  - $right\_child(A, p)$ takes as parameters a tree A, a vertex p and returns a vertex of A named right child of p or *None* if p has no right child.
  - $parent(A, f)$ takes as parameter a tree A , a vertex f and returns a vertex of A or *None*; this vertex is called the parent of f;
  - $root(A)$ takes as paremeter a tree A and returns a vertex of A named the root of the tree.

Primitives of Binary Trees - Properties

- a root has no parent : $parent(A, root(A)) = None$;
- From any vertex, it is possible to reach the root by using parent relationship :

$$\forall s \in A, \exists k \in \mathbb{N}, root(parent^k(A, s));$$

 where $parent^k$ is recursively defined by : $parent^0(A, s) = s$ and $parent^{k+1}(A, s) = parent(A, parent^k(A, s))$ for $k \geq 1$.
- The left child of a vertex s have as parent s :

$$\forall s \in A, \text{if } left\_child(A, s) \neq None \text{ then } parent(A, left\_right(A, s)) = s;$$

- The right child of a vertex $s$have as parent s :

$$\forall s \in A, \text{if } right\_child(A, s) \neq None \text{ then } parent(A, right\_child(A, s)) = s$$

Labeled Binary Trees

- A labeled binary tree is a tree A with one operation $label(A, s)$ which returns an element named label of s for each vertex.
- It is possible to draw a tree like that :