INF421, Lecture 1
Lists and Complexity

Leo Liberti

LIX, École Polytechnique, France

Course

- **Objective**: to teach you some data structures and associated algorithms
- **Evaluation**: TP noté en salle info le 16 septembre, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 26/8, 2/9, 9/9, 16/9, 23/9, 30/9, 7/10, 14/10, 21/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI31,32,33,34)
- **Books**:
  1. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
  2. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  3. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  4. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

Lecture summary

- Reminders
- Complexity
- Lists

Reminders

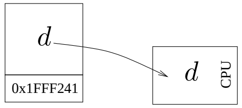
Memory

Memory cell

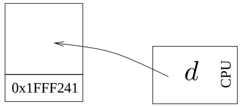
- has an address
- stores a datum d

Two operations

- Move datum from cell to CPU (read)



- Move datum from CPU to cell (write)



Representation of memory: *a sequence of cells*

d_0	d_1	d_2	d_3	d_4	d_5
0x0	0x1	0x2	0x3	0x4	0x5

A function $D : \mathbb{A} \to \mathbb{D}$

$\mathbb{A}$: set of addresses

$\mathbb{D}$: set of data elements

Assumptions

- For theoretical purposes, assume memory is infinite

$\to$ In practice it is finite

- Each datum can be stored in a single cell

$\to$ Different data elements might have different sizes

Naming memory

A program variable is just a name for a chunk of memory

X denotes:

0x4	0x5	0x6	0x7

- We simply associate a name to the starting address
- The size of the chunk is given by the name's **type**

- **Basic types**: `int`, `long`, `char`, `float`, `double`
- **Composite types**: Cartesian products of basic types

  if `y.a` $\in$ `int` and `y.b` $\in$ `float` then `y` $\in$ `int` $\times$ `float`

Basic operations

- **Assignment**: write value in memory cell(s) named by variable (i.e. "variable=value")
- **Arithmetic**: $+, -, \times, \div$ for integer and floating point numbers
- **Test**: evaluate a logical condition: if true, change address of next instruction to be executed
- **Loop**: instead of performing next instruction in memory, jump to an instruction at a given address (more like a "go to")

WARNING! *In these slides, I use "$=$" to mean two different things*:

1. in assignments, "variable $=$ value" means "put value in the cell whose address is named by variable"
2. in tests, "variable $=$ value" is TRUE if the cell whose address is named by variable contains value, and FALSE otherwise

in C/C++/Java "$=$" is used for assignments, and "$==$" for tests

Composite operations: programs

Programs are built recursively from basic operations

- If A, B are ops, then concatenation "`A;B`" is an op

  Semantics: execute A, then execute B

- If A, B are ops and T is a test, "`if (T) A else B`" is an op

  Semantics: if T is true execute A, else B

- If A is an op and T is a test, "`while (T) A`" is an op

  Semantics: `1:(if (T) A else (go to 2)) (go to 1) 2:`

Complexity

Complexity

- Several different programs can yield the same result: which is best?
- Evaluate their time (and/or space) complexity
  - **time complexity**: how many "basic operations"
  - **space complexity**: how much memory

  used by the program during execution
- *Worst case*: max values during execution
- *Best case*: min values during execution
- *Average case*: average values during execution

P: a program
t_P: number of basic operations performed by P

Time complexity (worst case)

- $\forall P \in \{\texttt{assignment}, \texttt{arithmetic}, \texttt{test}\}$:

  $$\boxed{t_P = 1}$$

- **Concatenation**: for P, Q programs:

  $$\boxed{t_{P;Q} = t_P + t_Q}$$

- **Test**: for P, Q programs and R a test:

  $$\boxed{t_{\texttt{if } (T)\ P\ \texttt{else}\ Q} = t_T + \max(t_P, t_Q)}$$

  $\max$: worst-case policy

- **Loop**: it's complicated

  (depends on how and when loop terminates)

Loop complexity example

The complete loop

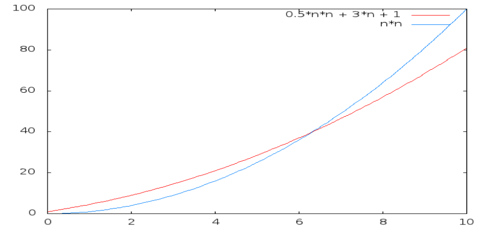
Let P be the following program:

```
i = 0;
while (i < n) do
  A;
  i = i + 1;
end while
```

- Assume A does not change the value of i
- Body of loop executed n times
- $t_P(n) = 1 + n(t_A + 3)$
- *Why the '3'? Well,* $t_{(i<n)} = 1$, $t_{(i+1)} = 1$, $t_{(i=\cdot)} = 1$

Orders of complexity

- In the above program, suppose $t_A = \frac{1}{2}n$
- Then $t_P = \frac{1}{2}n^2 + 3n + 1$
- No one really cares about the constants $2, 3, 1$: all that matters is that t_P "behaves no worse than" the fn. n^2



$\frac{1}{2}n^2 + 3n + 1$ is $O(n^2)$

- A function $f(n)$ is *order of* $g(n)$ (notation: $O(g(n))$) if:

$$\exists c > 0 \; \exists n_0 \in \mathbb{N} \; \forall n > n_0 \; (f(n) \leq cg(n)) \tag{1}$$

- For $f(n) = \frac{1}{2}n^2 + 3n + 1$ and $g(n) = n^2$, $c = 1$ and $n_0 = 6$

Some examples

Functions	*Order*
$an + b$ with a, b constants	$O(n)$
polynomial of degree d' in n	$O(n^d)$ with $d \geq d'$
$n + \log n$	$O(n)$
$n + \sqrt{n}$	$O(n)$
$\log n + \sqrt{n}$	$O(\sqrt{n})$
$n \log n^3$	$O(n \log n)$
$\frac{an+b}{cn+d}$, a, b, c, d constants	$O(1)$

- Make an effort to find the best (most slowly increasing) function $g(n)$ when saying "$f(n)$ is $O(g(n))$"
- E.g. no one would say that $2n + 1$ is $O(n^4)$ (although it's technically true) — rather say $2n + 1$ is $O(n)$

Remark

- The complexity order is an asymptotic description of $t_P(n)$
- If $t_P(n)$ does not depend on n, its order of complexity is $O(1)$ (i.e. constant)
- **Example**: looping 10^{1000} times over an $O(1)$ code still yields an $O(1)$ program
- In other words, n must appear as a parameter of the program for the complexity order to be anything other than constant

Complexity of easy loops

```
input n;
int s = 0;
int i = 1;
while (i ≤ n) do
  s = s + i;
  i = i + 1;
end while
output s;
```

- $t(n) = 3 + 5n + 1 = 5n + 4$
- $\Rightarrow t(n)$ is $O(n)$

```
for i = 0; i < n − 1; i = i + 1 do
  for j = i + 1; j < n; j = j + 1 do
    print i, j;
  end for
end for
```

- $t(n) = 1 + \underbrace{(5(n-1)+6)+\ldots+(5+6)}_{n-1}$ $= 1 + 5((n-1)+\ldots+1) + 6(n-1) = \frac{5}{2}n(n-1) + 6n - 5$ $= \frac{5}{2}n^2 + \frac{7}{2}n - 5$
- $t(n)$ is $O(n^2)$

Arrays

Like a vector in maths

- A vector $x \in \mathbb{Q}^n$ is an n-tuple $(x_1, \ldots, x_n)$ for some $n \in \mathbb{N}$
- In computers: x is the name for a memory address with n successive cells
- Indexing starts from 0 (last cell is called x_{n-1})

x :	x_0	x_1	x_2	x_3	x_4

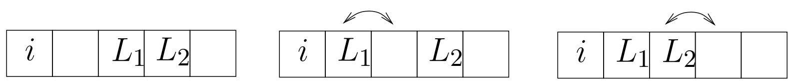
- An array is **allocated** when the memory is reserved
- The size of the array, n, is decided at allocation time
- Usually, the size of the array does not change
- When the array is no longer useful, the reserved memory can be **deallocated** or **freed**

Array operations

For an array of size n:

Operations	*Complexity*
Read value of i-th component	$O(1)$
Write value in i-th component	$O(1)$
Size	$O(1)$
Remove element (cell)	*forget it**
Insert element (cell)	*forget it**
Move subsequence to position i	$O(n)$

Moving subsequence L to position i:
extract (contiguous) subsequence L from the array, and re-insert it after position i and before position $i+1$



*: can simulate these operations using pointers, or de-realloc

Norm of a vector in $\mathbb{R}^5$

input $x \in \mathbb{Q}^5$;
`int` $i = 0$;
`float` $a = 0$;
while $(i < 5)$ **do**
  $a = a + x_i \times x_i$;
end while
$a = \mathtt{sqrt}(a)$;

- Computes $\sqrt{\sum_{i=0}^{4} x_i^2}$
- Complexity: $O(1)$ (why?)

Incomplete loop

input $x \in \{0,1\}^n$;
`int` $i = 0$;
while $(i < n \wedge x_i = 1)$ **do**
  $x_i = 0$;
  $i = i + 1$;
end while
if $(i < n)$ **then**
  $x_i = 1$;
end if
output x;

Input	*Output*
(0,0,0,0)	(1,0,0,0)
(1,1,0,0)	(0,0,1,0)
(0,1,1,0)	(1,1,1,0)
(1,1,1,1)	(0,0,0,0)

- Components of x can only be 0 or 1
- Loop continues over all components as long as their value is 1; at the first 0 component, it stops
- Complexity?

Worst case complexity of incomplete loop

- Among all inputs of the algorithm, find those yielding the worst complexity
- In the case above, $x = (1, 1, \ldots, 1)$ always makes the loop continue to the end, i.e. for n iterations

Thm.

$(1, 1 \ldots, 1)$ is the input yielding worst complexity

Proof

Suppose false, then there is a vector $x \neq (1, \ldots, 1)$ yielding a complexity $t(n) > n$. Since $x \neq (1, \ldots, 1)$, x contains at least one 0 component. Let $j < n$ be the smallest index such that $x_j = 0$: at iteration j the loop breaks, and the complexity is $t(n) = j$, which is smaller than n: contradiction.

- Since the other operations are $O(1)$, get $O(n)$

Potential difficulty of this approach: identifying the worst-case inputs and *proving* no other input is worse

Average case complexity of incomplete loop (1/2)

- Average case analysis needs a probability space:
  - assume the event $x_i = b$ is independent of the events $x_j = b$ for all $i \neq j$
  - assume each cell x_i of the array contains 0 or 1 with equal probability $\frac{1}{2}$
- For any vector having first $k + 1$ components $(\underbrace{1, \ldots, 1}_{k}, 0)$, the loop is executed k times (for all $0 \leq k < n$)

Event of a binary $(k+1)$-vector having given components has probability $\left(\frac{1}{2}\right)^{k+1}$

- If the vector is $(\underbrace{1, \ldots, 1}_{n})$ the loop is executed n times

Event of a binary n-vector having given components has probability $\left(\frac{1}{2}\right)^{n}$

Average case complexity of incomplete loop (2/2)

- The loop is executed k times with probability $\left(\frac{1}{2}\right)^{k+1}$, for $k < n$
- The loop is executed n times with probability $\left(\frac{1}{2}\right)^{n}$
- Average number of executions:

$$\sum_{k=0}^{n-1} k2^{-(k+1)} + n2^{-n} \le \sum_{k=0}^{n-1} k2^{-k} + n2^{-n} = \sum_{k=0}^{n} k2^{-k}$$

Thm.

$\lim_{n\to\infty} \sum_{k=0}^{n} k2^{-k} = 2$

Proof

Geometric series $\sum_{k\ge 0} q^k = \frac{1}{1-q}$ for $q \in [0,1)$. Differentiate w.r.t. q, get $\sum_{k\ge 0} kq^{k-1} = \frac{1}{(1-q)^2}$; multiply by q, get $\sum_{k\ge 0} kq^k = \frac{q}{(1-q)^2}$. For $q = \frac{1}{2}$, get $\sum_{k\ge 0} k2^{-k} = (1/2)/(1/4) = 2$.

Hence, the average complexity is constant $O(1)$

Jagged arrays

- **Jagged array**: a vector whose components are vectors of possibly different sizes
- E.g. $x = ((x_{00}, x_{01}), (x_{10}, x_{11}, x_{12}))$

x :

x_0 :	x_{00}	x_{01}	
x_1 :	x_{10}	x_{11}	x_{12}

- **Special case**: when all subvector sizes are the same, get a matrix: `int x[][] = new int [2][3];`

$$x = \begin{pmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \end{pmatrix}$$

Representing relations

- Jagged arrays can be used to represent a relation on a finite set
- Let $V = \{v_1 \dots, v_n\}$ and E a relation on V
  E is a set of ordered pairs (u, v)
- **Representation**:
  - array of n components
  - the i-th component is the array of v_j related to v_i
- Example: $V = \{1, 2, 3\}$,
  $E = \{(1,1), (1,2), (2,3), (3,1), (3,2), (3,3)\}$

E :

1	1	2	
2	3		
3	1	2	3

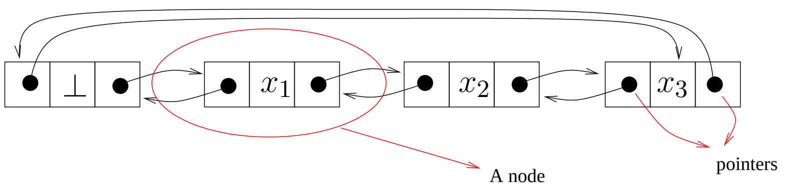
Application: Networks

Array shortcomings

- Essentially fixed size
- Size must be known in advance
- Changing relative positions of elements is inefficient

Lists

Doubly linked list



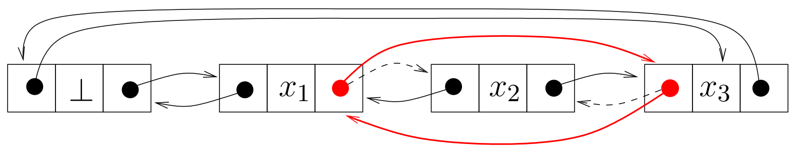
- **Node** N: a list element

N.`prev`	$=$	address of previous node in list
N.`next`	$=$	address of next node in list
N.`datum`	$=$	the data element stored in the node

- **Placeholder node** $\perp$: before the first element, after the last element, no stored data
- *Every node has two pointers, and is pointed to by two nodes*

Remove a node

Remove current node (`this`)



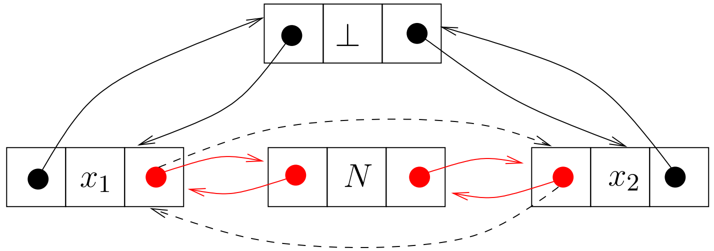
In the example, `this`$= x_2$

1: `this.prev.next` $=$ `this.next` ;
2: `this.next.prev` $=$ `this.prev` ;

Worst case complexity: $O(1)$

Insert a node

Insert current node (`this`) after node x_1



In the example, `this`$= N$

```
this.prev = x1 ;
this.next = x1.next ;
x1.next = this ;
this.next.prev = this ;
```

Worst case complexity: $O(1)$

Find next

- Given a list L and a node x, find next occurrence of element b
- If $b \in L$ return node where b is stored, else return $\perp$

while $(x$.`datum` $\neq b \wedge x \neq \perp)$ **do**
  $x = x$.`next`
end while
return x

Warning: *every test costs 2 basic operations, inefficient*

$\perp$.`datum` $= b$
while $(x$.`datum` $\neq b)$ **do**
  $x = x$.`next`
end while
return x

Now $t_{\text{test}} = 1$

List operations

For a doubly-linked list of size n:

Operations	*Complexity*
Read/write value of i-th node	$O(n)$
Find next	$O(n)$
Size[a]	$O(n)$
Is it empty?	$O(1)$
Read/write value of first/last node	$O(1)$
Remove element	$O(1)$
Insert element	$O(1)$
Move subsequence to position i	$O(1)$
Pop from front/back	$O(1)$
Push to front/back	$O(1)$
Concatenate	$O(1)$

[a] Some implementations are $O(1)$ by storing and updating size

End of Lecture 1