INF421, Lecture 1
Computability, Complexity
Arrays and Lists

Leo Liberti

LIX, École Polytechnique, France

Course

- **Objective**: teach notions AND develop intelligence
- **Evaluation**: TP noté en salle info, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 31/8, 7/9, 14/9, 21/9, 28/9, 5/10, 12/10, 19/10, 26/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI:30-34)
- **Books**:
  1. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
  2. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  3. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  4. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Blog**: `inf421.wordpress.com`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

Breaking news

Too many students!

No space in *salles informatiques*

ABSOLUTELY NO CHANGE IS POSSIBLE — DON'T EVEN ASK!!!

Other info

- Lectures are meant to develop your intelligence, **NOT** to prepare you to TDs
- $\Rightarrow$ discover links between lectures and TDs yourselves!
- Learn theory and algorithmics in lectures, Java in TDs
- $\Rightarrow$ not much Java code in lectures
- Slides: published online *after* the lectures

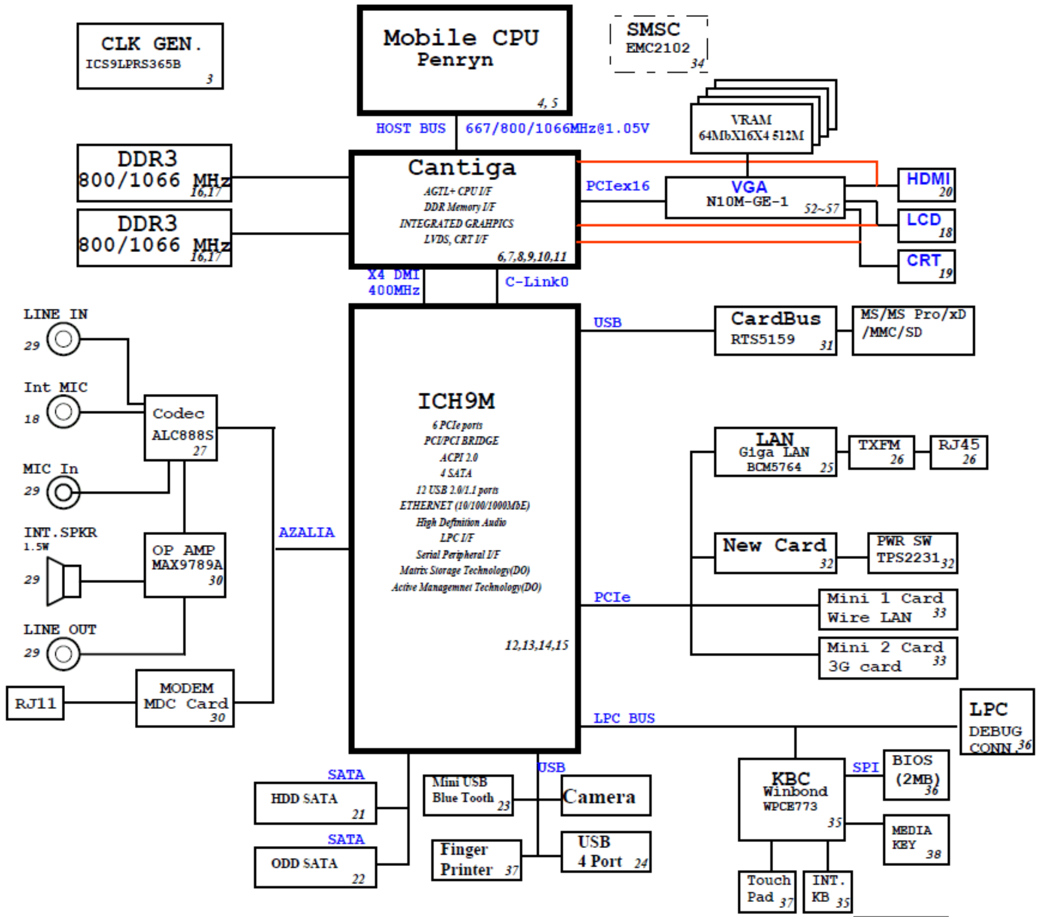
Lecture summary

- Computability
- Complexity
- Arrays
- Lists

Computability (informal)

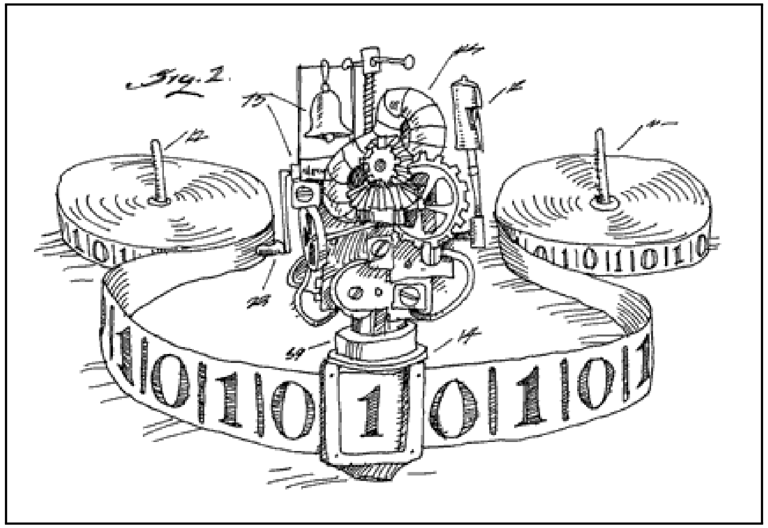
Computer

- Central Processing Unit (CPU)
- Random-Access Memory (RAM)
- Long-term storage:
  - Hard Disks (HD)
  - Compact Discs (CD)
  - Digital Versatile Discs (DVD), ...
- Input/Output (IO):
  - Keyboard
  - Mouse
  - Ports (network, USB, etc.)
  - Screen, ...



Turing Machine (TM)

- A finite alphabet of symbols (e.g. $\{0, 1, \Box\}$)
- An infinitely long tape divided into cells
- A tape "head" that can perform the following actions:
  - read symbols off a cell
  - write symbols on a cell
  - move to the next or previous cell on the tape
  - do nothing



- An infinite amount of time instants

  The head can do one action only at each time instant
- A set of instructions for the head

Simulating in TMs

Would a further action

$\boxed{\text{move to } k\text{-th next cell on tape}}$

make the TM "more powerful"?

- powerful $=$ able to perform more tasks
- simulate the new TM (T') using the old TM (T):

  "move to k-th next cell" $=$ repeat k times "move to next cell"

- $\Rightarrow T$ can do whatever T' can do
- $\Rightarrow$ same power

A task, a TM

- Set of instructions is <u>given</u>
- Determines the task a TM can do

1. read cell content 2. **if** 0, write 1 3. **else if** 1, write 0 4. **else if** $\square$, do nothing 5. **endif** 6. move to next cell 7. **repeat from** (Line **??**)	*Flip binary digits on input data*

- <u>Program makes TM act on input data</u>

Encode the program

- Programs are text
- Text can be encoded as a sequence of numbers
- Any number sequence can be encoded as a sequence of binary numbers
- $\Rightarrow$ A program can be an input to a TM

Universality

- Consider the following TM U:
  - Input:
    - a TM T encoded as a number
    - a valid input ι for T
  - Output: the output $T(\iota)$
  - Program: it must be able to "simulate" any TM

$$\forall T, \iota \quad U(T, \iota) = T(\iota)$$

- U is called a **Universal Turing Machine** (UTM)
- The program of U is known as an **interpreter**

Other UTMs

- Different models of computations
  - λ-calculus
  - RAM machines
  - (some) Diophantine equations
  - (some) cellular automata
- Let M be a model of computation
- M is **Turing-complete** if it can simulate a UTM
- M Turing-complete and can be simulated by a UTM: M is **Turing-equivalent**

Church's thesis

All Turing-complete models of computations are also Turing-equivalent

Can't find anything more powerful than a UTM

I printed "Church's hypothesis" in the polycopié by mistake: it should be "thesis"

Programming languages

- All programs are expressed in a language
- Consider simple language ℓ:
  - alphabet $\{0, (,)\}$
  - if s is a valid sentence, (0) is valid
  - 0 is a valid sentence
  - $\Rightarrow \quad \ell = \{0, (0), ((0)), \ldots\}$
- Question the *expressive power* of a programming language
- If language L can express an interpreter for a UTM, then L is **universal**
- If L can express **concatenation**, **tests** and **loops** then it is universal [Böhm and Jacopini, 1966]

Imperative vs. declarative

- Consider input and output for a TM T
- $\mathcal{I}(T) =$ set of all valid inputs for T
- $\mathcal{O}(T) =$ set of all valid outputs for T
- TM can be seen as a function $T : \mathcal{I} \to \mathcal{O}$
- Two possible descriptions of the function $x!$

Imperative	**Declarative**

input integer $x \geq 1$
let $y = 1$
for $z \in \{1, \ldots, x\}$ **do**
  let $y \leftarrow yz$
end for
	$y = \prod_{z=1}^{x} z$

Computable numbers

- $\mathbb{T}$ = TMs with empty input and output in $\mathbb{R}$
- The set

$$\mathscr{C} = \bigcup_{T \in \mathbb{T}} \mathcal{O}(T)$$

  is the set of **computable numbers** [Turing, 1936]
- Not all reals are computable
- (Proof by cardinality: there are at most countably many TMs, so countably many computable numbers, but uncountably many reals — so most reals are uncomputable)

Decision problems

- **Problem**: a question, parametrized over symbols taking infinitely many values, with possible answers YES or NO
- Every set of parameter values is an **instance**
- "Is the length of the program of TM T greater than k?"
  - parameters: T and k
  - there are infinitely many TMs T and integers k
  - only possible answers: YES or NO
- Given a problem P, is there a TM that solves it?
- **Solve** $=$ TM terminates with correct answer in finite time
- If $\exists$ TM solving P, P is **decidable**, otherwise **undecidable**

Halting problem

- Consider the **halting problem**:
  Given a TM T, will it terminate?
- Suppose $\exists$ TM H solving the halting problem
- So $H(T) =$ YES if T terminates, and NO otherwise
- Define TM K such that:
  - if H outputs NO then K halts
  - if H outputs YES then K loops forever
- Consider $H(K)$:
  - if $H(K) =$ YES then K does not halt
  - if $H(K) =$ NO then K halts
- $\Rightarrow H$ does not solve the halting problem

The halting problem is undecidable

From TM to computer

Code and data segments

- Computer is an approximate UTM
- Must be able to store TM programs
- Memory (RAM) holds both data <u>and</u> program code
- Certain memory addresses point to *instructions*
- Other addresses point to *variable values*

Imperative languages

- Variable symbols: $x_1, x_2, x_3, \ldots$
- Semantics:
  - $x_i \to$ address of memory storing value of x_i
  - type of data stored in x_i (boolean, integer, float, class,...)
- Logical/arithmetic operators and functions
- Flow control: assignments, **if**, **for**, **while**, ...

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

Programs

- By [Böhm and Jacopini, 1966], need loops, tests and concatenation to have a universal language
- Programs are concatenations of basic operations
- **Algorithm**: program written in “pseudocode”
- Can’t be executed, but easier to understand

Complexity

Complexity

- Consider a decidable problem P and two different algorithms to solve it: which is best?
- Time/space complexity:
  - **time complexity**: time taken to terminate
  - **space complexity**: necessary memory
- *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)

The complete loop

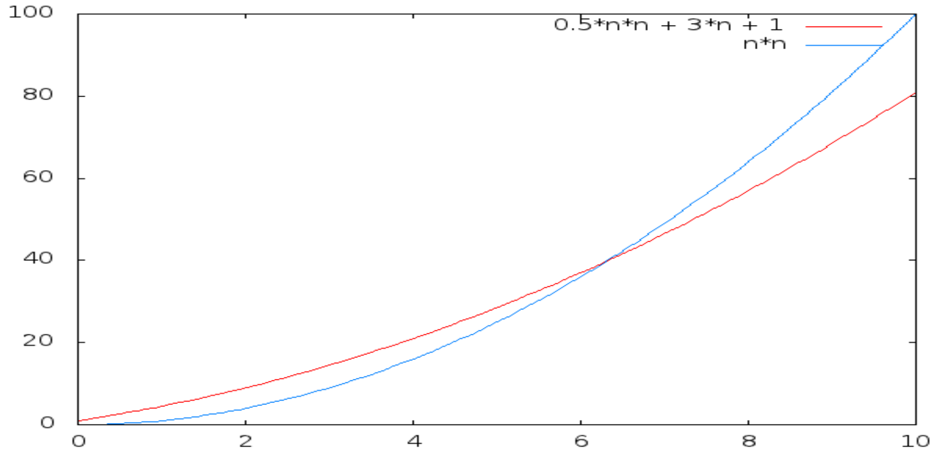
Let P be the following program:

```
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
- $t_P(n) = 1 + n(t_A + 3)$
- $t_{(i<n)} = 1, t_{(i+1)} = 1, t_{(i=\cdot)} = 1 \Rightarrow (\ldots + 3)$

Orders of complexity

- In the above program, suppose $t_A = \frac{1}{2}n$
- Then $t_P = \frac{1}{2}n^2 + 3n + 1$
- When n is large, t_P "behaves like" n^2



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

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

$$\exists c > 0 \ \exists n_0 \in \mathbb{N} \ \forall n > n_0 \ (f(n) \le cg(n))$$

- For $\frac{1}{2}n^2 + 3$, $c = 1$ and $n_0 = 2$

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	$O(1)$

- Find the best (most slowly increasing) function $g(n)$ when saying "$f(n)$ is $O(g(n))$"

$2n+1$ is $O(n^4)$, but it's best to say $O(n)$

Constant complexity

- 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;
```

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

- **Array**: represents a vector $x = (x_0, \ldots, x_{n-1})$

$$x : \begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & x_4 \\ \hline \end{array}$$

- Array **allocation**: reserving the necessary memory
- Size n decided at allocation time
- Usually array size does not change

  changes are expensive

- Array **deallocation** when no longer useful

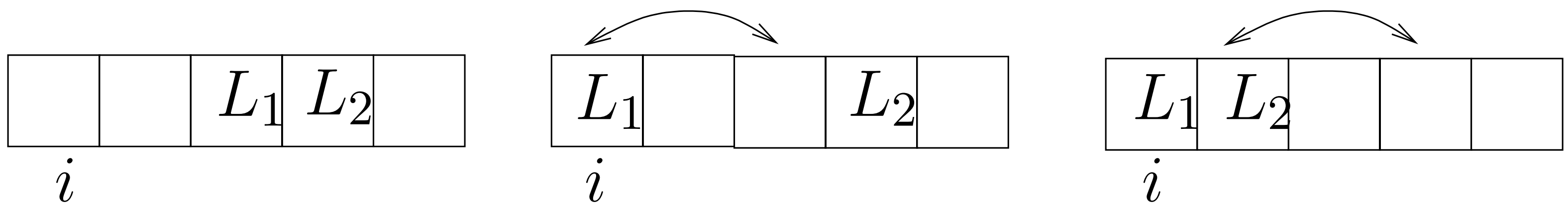
  can be automatic, e.g. in Java

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 (contiguous) subsequence L to position i:
start moving from L_1 if $i < L_1$, and from L_m if $i > L_1$



$*$: can simulate these operations using pointers, or dealloc/realloc

Incomplete loop

Loop over $x \in \{0,1\}^n$ while $x_i = 1$, setting $x_i \leftarrow 0$, stop when $x_i = 0$

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)

Worst-case complexity with input $x = (1, \ldots, 1)$

Average case complexity 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 $\frac{1}{2}$

Average case complexity 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 $\left(\frac{1}{2}\right)^{k+1}$

Average case complexity 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 $0 \le k < n$)

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 2/2

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

Average case complexity 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 case complexity 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:

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

Average case complexity 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.

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

Proof

Geometric series $\sum_{k\ge0} q^k = \frac{1}{1-q}$ for $q \in [0,1)$. Differentiate w.r.t. q, get $\sum_{k\ge0} kq^{k-1} = \frac{1}{(1-q)^2}$; multiply by q, get $\sum_{k\ge0} kq^k = \frac{q}{(1-q)^2}$. For $q = \frac{1}{2}$, get $\sum_{k\ge0} k2^{-k} = (1/2)/(1/4) = 2$.

Average case complexity 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:

$$\sum_{k=0}^{n-1} k 2^{-(k+1)} + n 2^{-n} \leq \sum_{k=0}^{n-1} k 2^{-k} + n 2^{-n} = \sum_{k=0}^{n} k 2^{-k}$$

Thm.

$\lim\limits_{n\to\infty} \sum_{k=0}^{n} k 2^{-k} = 2$

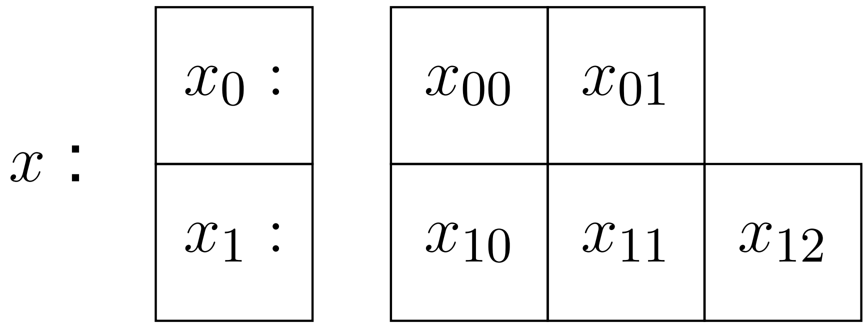
Proof

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

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

Jagged arrays

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



- **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 represent a relation
- Let $V = \{v_1 \ldots, v_n\}$ and E a relation on V

  E is a set of ordered pairs (u, v)
- **Representation**:
  - jagged array with n components
  - i-th array contains all v_j's 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

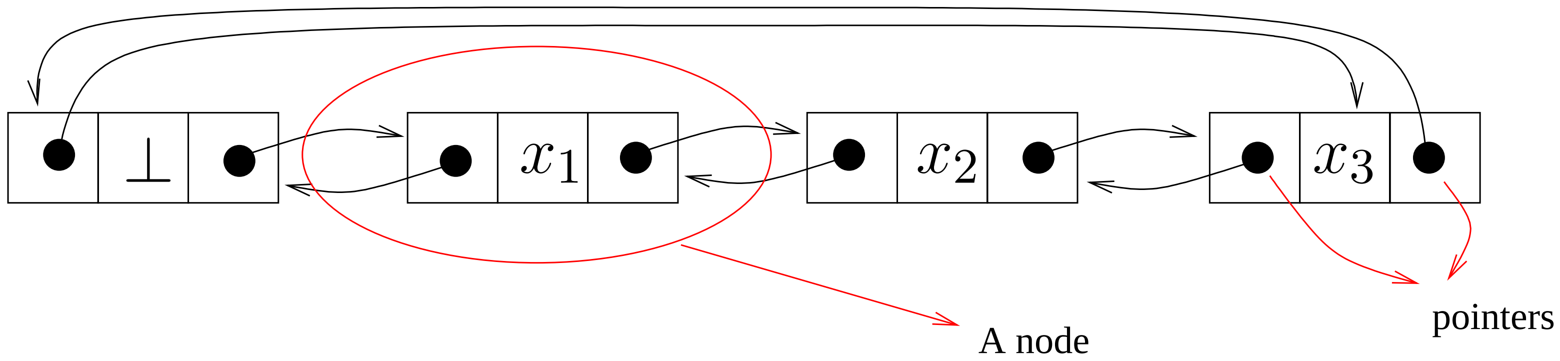
Application: Networks

Array shortcomings

- Fixed size known in advance
- Inserting/removing is inefficient
- Changing relative positions of elements is inefficient

Lists

Doubly linked list



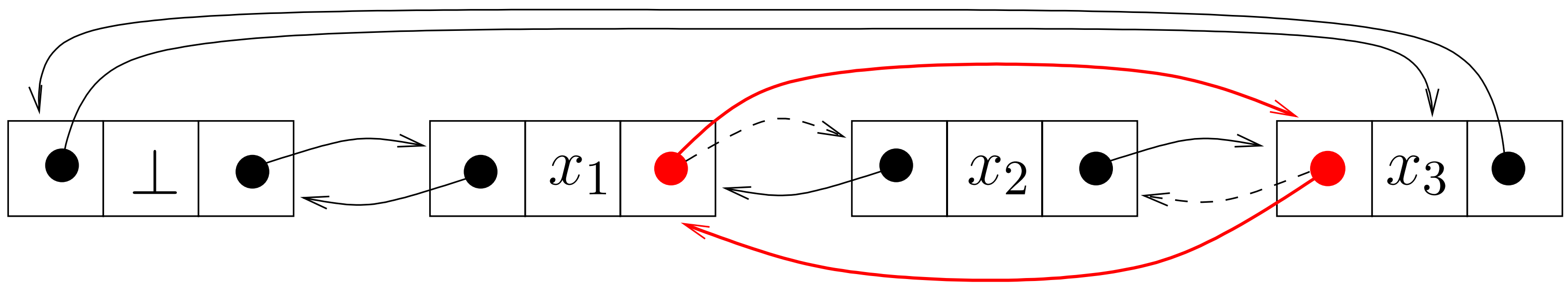
- **Node** N: a list element

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`)



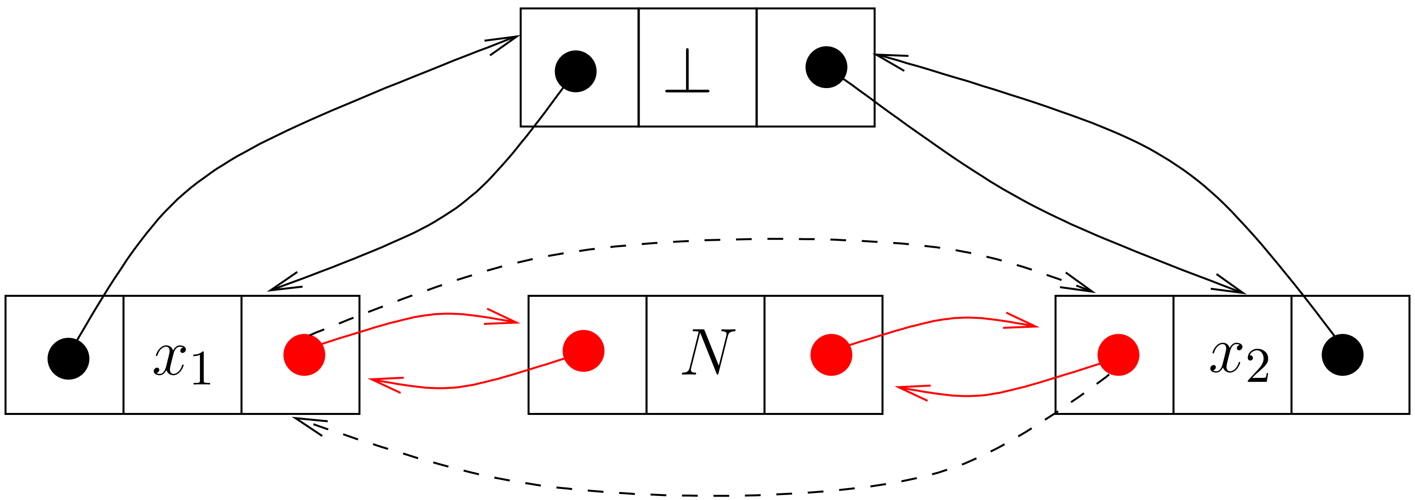
In the example, `this`$= x_2$

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`$= N$

1: `this.prev` $= x_1$;
2: `this.next` $= x_1$.`next` ;
3: x_1.`next` $=$ `this` ;
4: `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 $\perp$

1: **while** $(x.\mathtt{datum} \neq b \wedge x \neq \perp)$ **do**
2: $\quad x = x.\mathtt{next}$
3: **end while**
4: **return** x

Warning: *every test costs 2 basic operations*

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

Warning: *every test costs 2 basic operations*

$\perp.\mathtt{datum} = b$
while $(x.\mathtt{datum} \neq b)$ **do**
  $x = x.\mathtt{next}$
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