Algorithms and data structures

Claudine Chaouiya

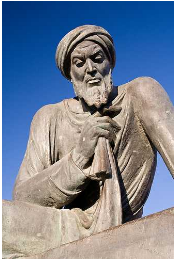
Instituto Gulbenkian de Ciência
PhD Program in Computational Biology
Oeiras, PORTUGAL

chaouiya@igc.gulbenkian.pt

October 2008

Outline

1. Introduction
   - What's an algorithm?
   - Abstract types *vs* data structures
   - Iteration, Induction, Recursion...
   - Problem classification and algorithm quality
   - Few mathematical complements - asymptotic notation
   - Few mathematical complements: Recurrence relations
2. Sequential data structures
3. Binary Trees
4. Graphs



Al-Khwarizmi (780-850), Khiva, Uzbekistan

A set of operating rules whose application allows the resolution of a given problem through a finite number of operations.

Example: Euclid's algorithm (one of the oldest algorithms known, around 300 BC) calculates the greatest common divisor of two non-zero integers

```
FUNCTION GCD(a, b)
 IF b = 0 RETURN a
 ELSE RETURN GCG(b, a mod b)
```

```
FUNCTION GCD(a, b)
  WHILE b ≠ 0
   t := b
   b := a mod b
   a := t
  RETURN a
```

We have

- a problem Find the number of
- an instance is defined by some data 'A' in "GAGATCAGACC"
- resolution produces some results 4

A program is an *implementation* of an algorithm *i.e.* its translation into a language "understandable" by a computer

but... **do not reduce algorithms to computational problem resolution**

an algorithm is independent of the programming language used to implement it

Defining an algorithm:

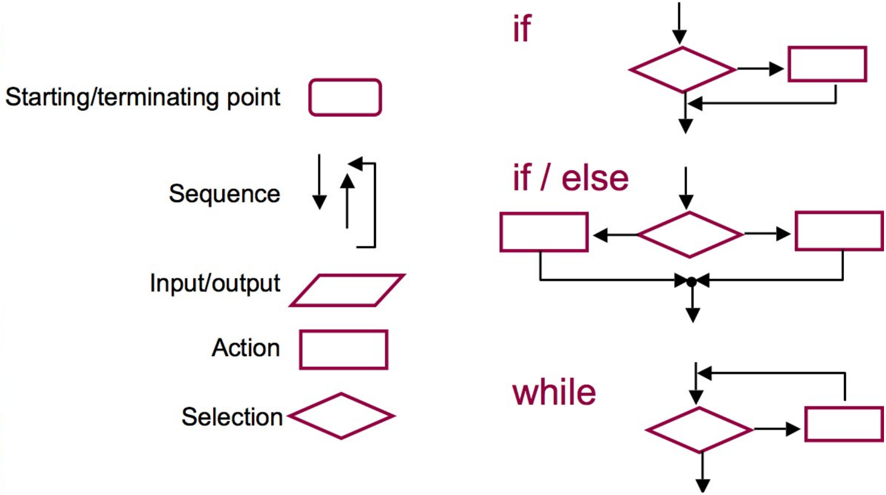
a finite set of operations on a given amount of that must terminate
each operation must be: **defined** (non ambiguous) & **effective** (can be performed by a computer)

Pseudo-code: informal language
Flowchart: graphical presentation

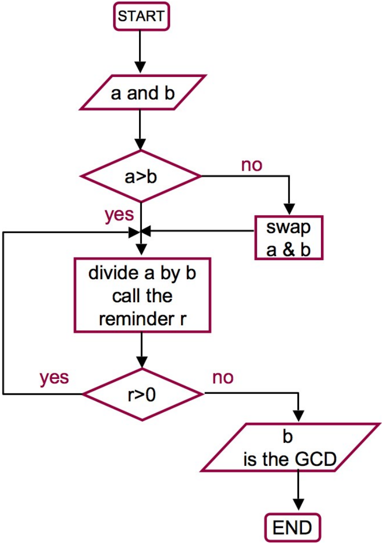
3 Control structures:

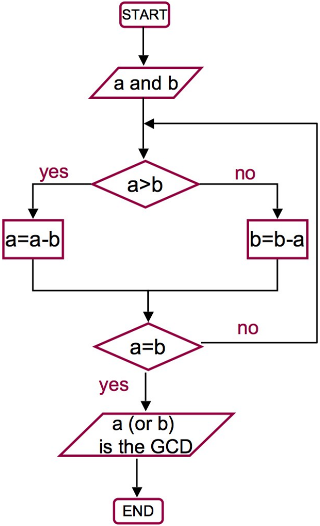
- *sequence*
- *selection (if, if/else, switch)*
- *repetition (while, do / while, for)*

Flowchart


Starting/terminating point
Sequence
Input/output
Action
Selection
if
if / else
while

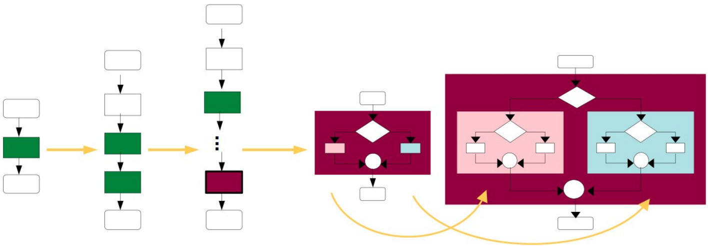
Again... Euclid's algorithm





Good principles (Structured programming)

- define the specification (what the algorithm does, not how it does)
- legibility and comments
- modularity
- avoid branching instruction (**go to**), use
  - loops **while, for**,
  - conditional instruction **if - then - else**
  - procedures or functions
- use recursion (recursivity), which allows short and clear description



Abstract types: abstractions used to formulate problems (lists trees, graphs...)

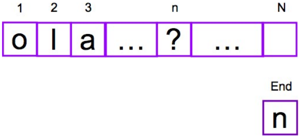
Data structures: concrete representations (implementations) of abstract types.

- we use `real` or `integer` in most (if not all) of programming languages as abstract types (don't care of their implementation).
- the abstract type `list` is the primary type provided by LISP

Name	List
Uses	integer,element
Operations	ith: (List,integer)$\rightarrow$element
	card:(List)$\rightarrow$element
	...

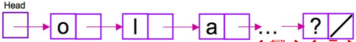
- a contiguous representation

type `LIST` =**array** `[1..N]` **of** `char`



- a linked representation

type `LIST` = $\uparrow$ `cell;`
`cell=`**record** `val:char;`
`link:LIST`
end;



Recursivity:

applying a function as a part of the definition of that same function.

- a *base case(s)*, for which the solution is known $\rightarrow$ termination condition,
- a recursive step.

Factorial

$$\begin{aligned} 0! &= 1 \\ n! &= n(n-1)! \; n > 0 \end{aligned}$$

Fibonacci numbers

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \; n > 1 \end{aligned}$$

Given a problem $\longrightarrow$ decidability (existence of an algorithm)

Termination problem: *Does it exist an algorithm which answers YES or NO to the question: "P terminates on D" for any program P and any entry D*

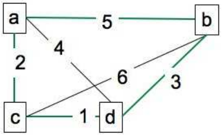
It has been proved that there is no algorithm to solve this problem
$\longrightarrow$ correctness and termination ♠ **Manfred Kerber's course**
$\longrightarrow$ **complexity**

Given a problem $\longrightarrow$ decidability (existence of an algorithm)

Termination problem: *Does it exist an algorithm which answers YES or NO to the question: "P terminates on D" for any program P and any entry D*

It has been proved that there is no algorithm to solve this problem
$\longrightarrow$ correctness and termination ♠ **Manfred Kerber's course**
$\longrightarrow$ **complexity**

Travelling salesman problem (TSP): given a weighted non-oriented complete graph with n nodes, find a minimal hamiltonian cycle.



ABCD → 5+6+1+4=16
ABDC → 5+3+1+2=11
ACBD → 2+6+3+4=15

- Algorithm: enumerate all hamiltonian cycles, choose the best
- $\frac{(n-1)!}{2}$ hamiltonian cycles

$n = 20 \rightarrow 19$ centuries on a computer able to determine 10^6 cycles p/sec

TSP is a well-known representant of a class of problems classified as NP-hard.

- performance analysis, independently of implementation
- comparison of algorithms

Complexity of an algorithm = time and/or memory space necessary for its execution

A reference computer:

- access (and storage) done in a fixed amount of time
- one operation performed at a time

Execution time $\propto \sharp$ *elementary operations*

Examples:

1. search an item in a list $\rightarrow$ number of comparisons
2. sorting a list $\rightarrow$ number of comparisons and of moves
3. matrix product $\rightarrow$ number of product and sum operations

To calculate the complexity (in time), count elementary operations,

- sequence: add
- conditional branching: upper-bound
- loop: $\sum_i .P(i)$, $P(i)$ being the number of operations for the ith execution of the lopp (i control variable of the loop)
- function call: number of operations of the function
- recursive function: solving recurrence relations $T(n) = f(T(k)),\ k < n$.

Example: the factorial function

```
FUNCTION fact(Integer n): Integer n {
   p=1
   FOR i=2 TO n
      p=p*i
   RETURN p
}
```

elementary operation: the product of 2 integers $\sum_{i=1...n} 1 = n$

```
FUNCTION fact(Integer n): Integer n {
   IF (n==0) RETURN 1
   ELSE RETURN n*fact(n-1)
}
```

$T(0) = 0$ and $T(n) = T(n-1) + 1, \; \forall n \geq 1$
easily solved: $T(n) = n$.

Example: sequential search (an integer x in a list L of size n)

```
FUNCTION search(List L, Integer x) : Boolean {
   i=1
   WHILE (i<=n AND (x!=L[i])
         i=i+1
   IF (i>n) RETURN false
   ELSE RETURN true
}
```

Example: sequential search (an integer x in a list L of size n)

```
FUNCTION search(List L, Integer x) : Boolean {
   i=1
   WHILE (i<=n AND (x!=L[i])
           i=i+1
   IF (i>n) RETURN false
   ELSE RETURN true
}
```

elementary operations: comparisons (one by iteration)

if $x \notin L \rightarrow n$, otherwise $\rightarrow$ rank of x in L

Loop invariants: properties true at each iteration

at the 1^{st} iteration $j = 1$

at the k^{th} iteration $j = k$ and $\forall i = 1 \ldots k-1, L[i] \neq x$

End condition(s):

if at the k^{th} iteration $k \leq Card(L)$ and $L[k] = x$

if $k = Card(L) + 1$

Other performance criteria

1. memory size: usual compromise between space and time
2. simplicity: implementation and maintainability
3. adequacy to the data: *e.g.* for a sorting algorithm, is the list almost sorted?

Other performance criteria

1. memory size: usual compromise between space and time
2. simplicity: implementation and maintainability
3. adequacy to the data: *e.g.* for a sorting algorithm, is the list almost sorted?

D_n set of entries of size n, $C_A(d)$ complexity of algorithm A for entry d:

- best case complexity: $Min_A(n) = min\{C_A(d), \quad d \in D_n\}$
- worst case complexity: $Max_A(n) = max\{C_A(d), \quad d \in D_n\}$
- average case complexity: $Aver_A(n) = \sum_{d \in D_n} p(d)\, C_A(d)$, $p(d)$ probability to get entry d. If all entries are equally likely, then

$$Aver_A(n) = \frac{1}{Card(D_n)} \sum_{d \in D_n} C_A(d).$$

$$Min_A(n) \leq Aver_A(n) \leq Max_A(n), \quad \forall n$$

Insertion sort

```
PROCEDURE INSERTION_SORT(List A)
    FOR i = 1 to Card(A)-1
        value = A[i]
        j = i-1
        WHILE j >= 0 AND A[j] > value
             A[j + 1] = A[j]
             j = j-1
        A[j+1] = value
```

Insertion sort

```
PROCEDURE INSERTION_SORT(List A)
    FOR i = 1 to Card(A)-1
        value = A[i]
        j = i-1
        WHILE j >= 0 AND A[j] > value
             A[j + 1] = A[j]
             j = j-1
        A[j+1] = value
```

- The outer loop carried out $n-1$ times.
- The inner loop carried out i times in the worst case; half that often on average. The number of comparisons in the worst case is

$$\sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i \; \frac{n(n-1)}{2}$$

In the average case it is $n(n-1)/4$

Product of square matrices $(n \times n)$: $C = AB$

```
FUNCTION PRODUCT(Matrix A,B): Matrix {
   FOR i=1 TO n
      FOR j=1 TO n
           C[i,j]=0
           FOR k=1 TO n
                C[i,j]=C[i,j]+A[i,k]*B[k,j]
    RETURN C
}
```

Product of square matrices $(n \times n)$: $C = AB$

```
FUNCTION PRODUCT(Matrix A,B): Matrix {
   FOR i=1 TO n
      FOR j=1 TO n
           C[i,j]=0
           FOR k=1 TO n
                C[i,j]=C[i,j]+A[i,k]*B[k,j]
    RETURN C
}
```

Here, elementary operations are the multiplications of integers,

$$Min(n) = Aver(n) = Max(n) = \sum_{1}^{n}\sum_{1}^{n}\sum_{1}^{n} 1 = n^3$$

Sequential search (an integer x in a list L of size n)

```
FUNCTION search(List L, Integer x) : Boolean {
   i=1
   WHILE (i<=n AND (x!=L[i])
           i=i+1
   IF (i>n) RETURN false
   ELSE RETURN true
}
```

Elementary operations are comparisons, $Min(n) = 1$ and $Max(n) = n$.
What about the average case knowing that: $p(x \in L) = q$ and if $x \in L$, $p(L[i] = x) = p(L[j] = x), \forall i, j = 1, \ldots, n$

Sequential search (an integer x in a list L of size n)

```
FUNCTION search(List L, Integer x) : Boolean {
   i=1
   WHILE (i<=n AND (x!=L[i])
          i=i+1
   IF (i>n) RETURN false
   ELSE RETURN true
}
```

Elementary operations are comparisons, $Min(n) = 1$ and $Max(n) = n$.
What about the average case knowing that: $p(x \in L) = q$ and if $x \in L$, $p(L[i] = x) = p(L[j] = x)$, $\forall i, j = 1, \ldots, n$

- $D_{n,i}$ set of entries s.t. $L[i] = x$, $p(D_{n,i}) = \frac{q}{n}$,
- $D_{n,0}$ set of entries s.t. $x \notin L$, $p(D_{n,0}) = 1 - q$,
- $cost(D_{n,i}) = i$, for $i \neq 0$ and $cost(D_{n,0}) = n$,

$$Aver(n) = \sum_{i=0}^{n} p(D_{n,i}) * cost(D_{n,i}) = (1-q)n + \frac{q}{n}\sum_{i=1}^{n} i = (1-q)n + \frac{q(n+1)}{2}$$

Asymptotic notations

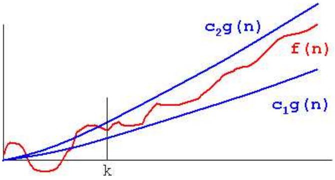
Bounding the asymptotical execution time of an algorithm

A fonction $IN \to IN$: (size of the problem) $\to$ (number of operations)

- Notation Θ (asymptotically tight bound):

$$\Theta(g(n)) = \{f(n) \ : \ \exists c_1, c_2 > 0, \exists k, \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq k\}$$

$f \in \Theta(g(n)$ is written $f(n) = \Theta(g(n))$



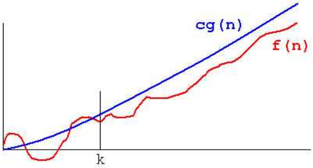
source: http://www.nist.gov/dads/

Examples: $1/2n^2 - 3n = \Theta(n^2)$ but $n^3 \neq \Theta(n^2)$

For all polynomial $P(n) = \sum_{i=0}^{d} a_i n^i$, $a^d > 0$, $P(n) = \Theta(n^d)$.

- Notation O (asymptotic upper bound):

$$O(g(n)) = \{f(n) \ : \ \exists c_1, k > 0, \quad 0 \leq f(n) \leq cg(n), \forall n \geq k\}$$



- Notation Ω (asymptotic lower bound):

$$\Omega(g(n)) = \{f(n) \ : \ \exists c_1, k > 0, \quad 0 \leq cg(n) \leq f(n), \forall n \geq k\}$$

Additional remarks

Complexity of some algorithms depends on several parameters:
e.g. on graphs, numbers of nodes and edges

$$f(n,p) = O(g(n,p)) \Leftrightarrow \exists c \in IR^{*+}, \exists (n_0,p_0) \in IN^2 \; s.t.$$

$$\forall n > n_0, \forall p > p_0, \quad f(n,p) \leq g(n,p).$$

- $g = O(g)$ and $g = \Theta(g)$
- $f = O(g),\ g = O(h) \Rightarrow f = O(h)$
- $f = O(g) \Rightarrow \lambda f = O(g),\ (\lambda \in IR^{*+})$
- $f = \Theta(g) \Rightarrow g = \Theta(f)$
- $f = \Theta(g),\ g = \Theta(h) \Rightarrow f = \Theta(h)$
- $f = \Theta(g) \Rightarrow \lambda f = \Theta(g),\ (\lambda \in IR^{*+})$
  - $f_1 = O(g_1),\ f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(max(g_1, g_2))$ (idem for Θ)
  - f_1 and f_2 s.t. $f_1 - f_2 \geq 0$,
    $f_1 = O(g_1),\ f_2 = O(g_2) \Rightarrow f_1 - f_2 = O(g_1)$
    $f_1 = \Theta(g_1),\ f_2 = \Theta(g_2),\ g_2 = O(g_1),\ g_1$ is not $O(g_2) \Rightarrow f_1 - f_2 = \Theta(g_1)$
  - $f_1 = O(g_1),\ f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$ (idem for Θ)

Determine if n is odd or even	$O(1)$	constant
Finding an item in a sorted array using binary search	$O(logn)$	logarithmic
Finding an item in an unsorted list	$O(n)$	linear
Sorting a list with heapsort	$O(nlogn)$	quasilinear
Sorting a list with insertion sort	$O(n^2)$	quadratic
Multiplying two $n \times n$ matrices by a simple algorithm	$O(n^3)$	cubic
Finding the shortest path on a weighted directed graph	$O(n^d), d > 1$	polynomial
Exact solution of the travelling salesman problem (shortest path in a network, visiting each node once)	$O(c^n)$	exponential

n	n^2	n^5	2^n	n!	
1	1	1	2	1	
2	4	32	4	2	
4	16	1024	16	24	
10	100	100000	1024	3628800	
20	400	3200000	1048576	2,4329E+18	
60	3600	777600000	1,15292E+18	8,32099E+81	
1	60	12960000	1,92154E+16	1,38683E+80	seconds
	1	216000	3,20256E+14	2,31139E+78	minutes
		3600	5,3376E+12	3,85231E+76	hours
		150	2,224E+11	1,60513E+75	days
			609315018,1	4,39761E+72	years

Execution time of a recursive algorithm generally defined as a recurrence relation: cost $T(n)$ for an entry of size n is function of $T(p)$, $p < n$.

Example: function `fact`: $T(0) = 0$ and $T(n) = T(n-1) + 1$, $\forall n \geq 1$

A recurrence relation always composed by two equations: 1/ for the base case, and 2/ for the general case

1. Linear recurrence relations of order k:

$$T(n) = f(n, T(n-1), \ldots, T(n-k)) + g(n)$$

   with, $k \geq 1$ a constant (integer), f linear function of $T(i), i = n-k \ldots, n-1$, g a function of n.

2. Partition recurrence relations:

$$T(n) = aT(n/b) + d(n)$$

   with, a, b constants, d a function of n.

Some examples of resolution

Linear recurrence relations, write the relation for $n, n-1, \ldots, 1$, multiply by a convenient factor, sum and simplify:
$T(n) = T(n-1) + 2^n, \quad T(0) = 1$

Some examples of resolution

Linear recurrence relations, write the relation for $n, n-1, \ldots, 1$, multiply by a convenient factor, sum and simplify:
$T(n) = T(n-1) + 2^n, \quad T(0) = 1$

$$\begin{array}{ll} T(n) & = T(n-1) + 2^n \\ T(n-1) & = T(n-2) + 2^{n-1} \\ T(n-2) & = T(n-3) + 2^{n-2} \\ \ldots & \ldots \\ T(1) & = T(0) + 2 \\ T(0) & = 1 \end{array}$$

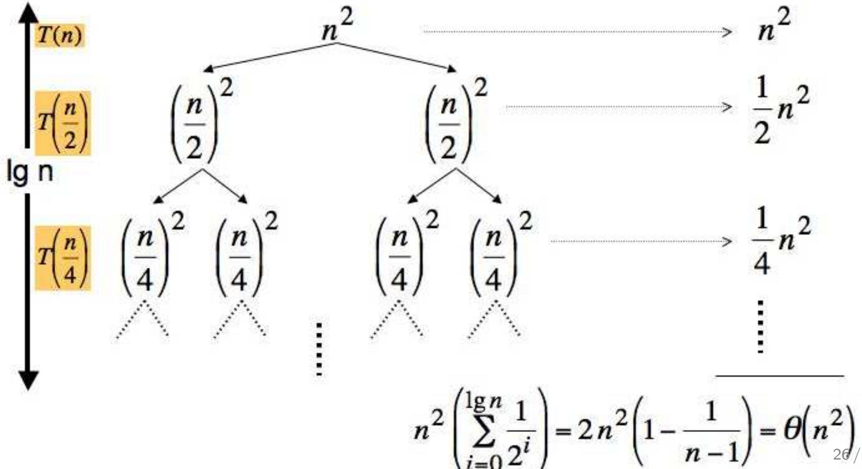
$\Rightarrow T(n) = \sum_{i=0}^{n} 2^i = 2^n - 1$

Some examples of resolution

Partition recurrence relations, using a recursive tree

$$T(n) = 2T(n/2) + n^2, \; T(0) = cste$$

Let assume $n = 2^p$ and $cste = 0$



Outline

1. Introduction
2. Sequential data structures
   - Generalities
   - Search, insertion, deletion
   - Queues and stacks
   - Sorting
3. Binary Trees
4. Graphs

Sequential data structures

Sequential structures or Lists (linked and arrays)

Finite sequence of elements of a given type.

Operations:

- insertion, deletion
- lookup
- concatenation...

Arrays:

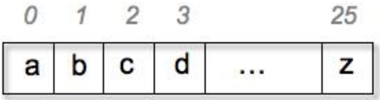
Set of elements accessible by their index.

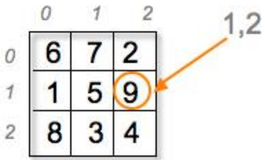
Generally, all elements have the same type (*e.g.* array of integers)

Static arrays (fixed size) *versus* dynamic arrays

Constant access time (contiguous storage, and index access)

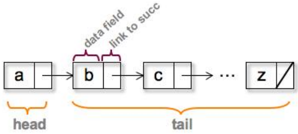
Not adequate for insertion or deletion



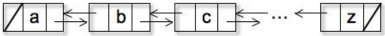


Linked lists

are recursive structures:
A list is either the empty list, or it is a *head* (an element) followed by a *tail* (a list).
Singly-linked list has one link per node that points to the successor in the list, or to a *null* value (or empty list) if it is the last node.



Doubly-linked list has two links per node that point to the predecessor in the list, or to a *null* value it is the first node, and to the successor, or to a *null* value if it is the last node.



Circularly-linked list is a singly or doubly linked list s.t. the first and final nodes are linked together.

Search, insertion, deletion

Search

Searching an element in an array:

```
FUNCTION search(Elt x, Array T): Integer p {
   p=0
   WHILE (p<card(T) AND T[p]<> x)
      p=p+1
   if (p<card(T)) RETURN p
   else RETURN -1
}

FUNCTION search(Elt x, Array T): Integer p {
   p=0
   WHILE (p<card(T) AND T[p]< x)
      p=p+1
   if (p<card(T) AND T[p]=x) RETURN p
   else RETURN -1
}
```

worst case $x \notin T$ ($\Omega(n)$), best case $x = T[0]$ ($O(1)$)

Search

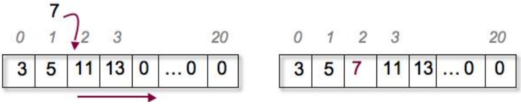
Searching an element in a singly-linked list:

```
FUNCTION search(Elt x, List T): Integer p {
   p=0
   L=T
   WHILE (L.succ<>null AND L.data<>x)
      L=L.succ, p=p+1
   if (L.data=x) RETURN p
   else RETURN -1
}
```

Insertion, deletion

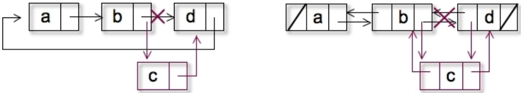
Inserting (*deleting*) an element in an array:

- find the position
- move remaining elts forward (*backwards*)
- insert *delete*)



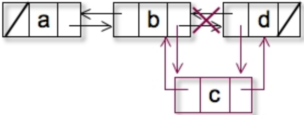
Inserting (*deleting*) an element in a linked list:

- find the position
- insert (*delete*)



Insertion

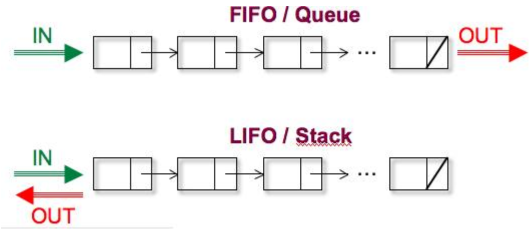
```
FUNCTION insert(Elt x,Integer n, List L){
  p=create(L,x,null,null)
  IF (L=null) L=p
  ELSE if(n=0)
      p.succ=L, L.pred=p, L=p
  ELSE
      q=L, i=0
      WHILE(q.succ<>null AND i<n)
        q=q.succ, i=i+1
      IF (i=n)
        p.pred=q.pred, p.succ=q, q.pred=p
      ELSE
        p.pred=, q.succ=p
```

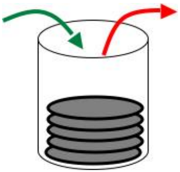


- What happens if $n > card(L)$?
- What happens if $n < 0$?

Queues and stacks

Insertion (and deletion) allways done at the same point:





Sorting

Insertion sort already seen.

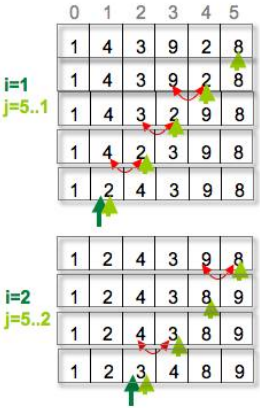
```
PROCEDURE INSERTION_SORT(List A)
    FOR i = 1 to Card(A)-1
        value = A[i]
        j = i-1
        WHILE j >= 0 AND A[j] > value
             A[j + 1] = A[j]
             j = j-1
        A[j+1] = value
```

- Simple to implement
- Efficient if the number of elements is small
- average time is $n^2/4$, linear in the best case

Bubble sort

Probably the most inefficient sorting algorithm in common usage!

```
FUNCTION bubble_sort(List A)
   FOR i=0 TO card(N)-2
      FOR j=N-1 DOWNTO i
       IF A[j-1]>A[j]
           swap(A[j-1], A[j])
```



What are the best and worst cases? Why is it in $\Theta(n^2)$?

How could you improve it? What are then the best and worst cases orders?

Merge sort

A *divide-and-conquer* algorithm. Given a problem P of size n

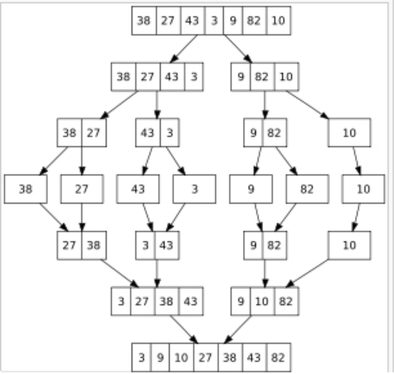
- base case: direct solution for P when n is small enough,
- divide: break down P into two or more sub-problems of size $q < n$,
- conquer: determine the solution of the sub-problems
- combine: the solution of $P(n)$ is a combination of the solutions of the sub-problems.

Let n be the size of the list:

1. if $n = 0$ or 1, the list is sorted;
2. if $n > 1$, divide the list into 2 sublists of about $n/2$;
3. sort the 2 sublists recursively (re-applying merge sort);
4. merge the 2 sublists back into one sorted list.

Merge sort

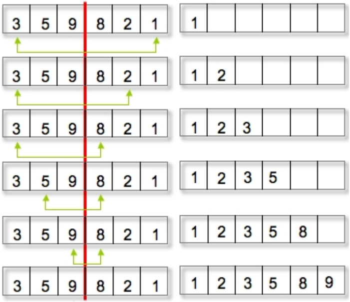
```
FUNCTION merge_sort(List A,Integer left,right){
   IF (left<right)
     middle=(left+right) DIV 2
     merge_sort(A,left,middle)
     merge_sort(A,middle+1,right)
     merge(A,left,right)
}
```



from Wikipedia

Merging pseudo-code

```
FUNCTION merge(Array A; Integer left,mid,right){

   FOR (i=left TO mid)
     aux[i]=A[i]
   FOR (i=right DOWNTO mid+1)
     aux[right+mid+1-i]=A[i]
   i=left, j=right
   FOR (k=left TO right)
     IF (aux[i]<aux[j])
        A[k]=aux[i], i=i+1
     ELSE
        A[k]=aux[j], j=j-1
}
```



What is the cost of merge for an array of n elements? What is the cost of the merge-sort? (you might assume that $n = 2^p$)

Quick sort

Let n be the size of the list, and $left = 0$, $right = n - 1$

1. divide the list from *left* to *right* into 2 sublists s.t. all elements of the first list are smaller all elements of the second, call *mid* the position of the partition;
2. conquer by recursively sorting the two sublists (from *left* to $mid - 1$, from $mid + 1$ to *right*);
3. if $right - left = 0$ do nothing!

```
quick_sort(Array A; Integer L,R){
  IF (L<R)
    M=partition(A,L,R)
    quick_sort(A,L,M-1)
    quick_sort(A,M+1,R)
}
```

```
quick_sort(Array A; Integer L,R){
  IF (L<R)
    M=partition(A,L,R)
    quick_sort(A,L,M-1)
    quick_sort(A,M+1,R)
}

partition(Array A; Integer L,R):Integer M{
  pivot=A[L], i=L+1, j=R
  WHILE (A[i]<=pivot) i=i+1
  WHILE (A[j]>=pivot) j=j-1
  WHILE (i<j)
    swap(A[i],A[j])
    WHILE (A[i]<=pivot) i=i+1
    WHILE (A[j]>=pivot) j=j-1
  swap(A[L],A[j])
  RETURN M=j
}
```

- What happens if `pivot` is the smallest element?
- What would be a good property for the pivot?
- What is the worst case? In this case, what is the order of the quick-sort?

On average, the quick-sort performs in $O(n \lg n)$ (number of comparisons.)

- Prove that the best case is in $\Theta(n lg\ n)$.

ADDENDUM on the resolution of recurrence relations

Theorem: *Let consider* $a \geq 1$ *and* $b > 1$*, 2 constants, and* $f(n)$ *a function, and let* $T(n)$ *defined for positive integers by:*

$$T(n) = aT(n/b) + f(n),$$

where n/b *is either* $\lfloor n/b \rfloor$ *either* $\lceil n/b \rceil$*. Then,* $T(n)$ *can be asymptotically bounded as follows:*

1. *If* $f(n) = O(n^{\log_b a - \epsilon})$ *with* $\epsilon > 0$*, then* $T(n) = \Theta(n^{\log_b a})$*.*
2. *If* $f(n) = \Theta(n^{\log_b a})$*, then* $T(n) = \Theta(n^{\log_b a} \lg n)$*.*
3. *If* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *with* $\epsilon > 0$*, and if* $af(n/b) \leq cf(n)$ *for a constant* $c < 1$ *and* n *large enough, then* $T(n) = \Theta(f(n))$*.*

In all cases, compare $f(n)$ with $n^{log_b a}$. The solution is determined by the maximum of these 2 functions:

- $n^{log_b a}$ is greater, the solution is $T(n) = \Theta(n^{log_b a})$.
- both functions have the same “size”, the solution is multiplied by a logarithmic factor: $T(n) = \Theta(n^{log_b a} \, lg \, n) = \Theta(f(n) \, lg \, n)$.
- $f(n)$ is greater, $T(n) = \Theta(f(n)$ (plus a regularity condition on f)..

These 3 cases do not cover all possibilities: $T(n) = 2T(n/2) + n \, lg \, n$ (f is not *polynomially* greater than $n^{log_b a} = n$ since $(n lg \, n)/n = lg \, n$ is asymptotically less than n^{ϵ}, whatever the positive constante ϵ.

Example:
$T(n) = 9T(n/3) + n$: $f(n) = n$, $n^{log_b a} = n^2$, $f(n) = O(n^{log_3 9 - \epsilon})$, with $\epsilon = 1$, $\Rightarrow$ $T(n) = \Theta(n^2)$.
see Cormen *et al* for details and proof of the theorem

Outline

1 Introduction

2 Sequential data structures

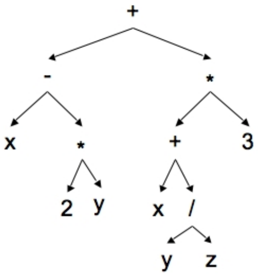
3 Binary Trees
- Representations of binary trees
- Traversing trees
- Binary Search Trees

4 Graphs

Binary trees

A **binary tree** *is empty ($\emptyset$) or on the form $B =< o, B_1, B_2 >$ where B_1 and B_2 are disjoint binary trees and o is a node called* **root**.

Binary tree representing the arithmetic expression
$(x - (2 * y)) + ((x + (y/z)) * 3)$



Note that $< o, < o, \emptyset, \emptyset >, \emptyset >$ and $< o, \emptyset, < o, \emptyset, \emptyset >>$ are different.

Basic operations on binary trees

- test if a tree is empty
- access the root
- access the left child (B_1)
- access the right child (B_2)

Measures on binary trees

- a node has at most 2 children; if its has no child, it is a *leaf*, it is a single node if it has a unique child, an *internal node* otherwise;
- the size of a BT is its number of nodes:

$$size(\emptyset) = 0, \qquad size(< o, B_1, B_2 >) = 1 + size(B_1) + size(B_2)$$

- the depth of a node n in $< o, B_1, B_2 >$ is:

$$depth(n) = 0 \,\text{if}\, n = o$$

$$depth(n) = 1 + depth(p) \,\text{where}\, p \,\text{s.t. n child of}\, p$$

- the depth (or height) of a tree is given as the maximum of its nodes depth.
- the traversing length of a tree B is the sum of its nodes depths:

$$LC(B) = \sum_{n \in B} h(n)$$

The total number of BT of size n is $b_n = \frac{1}{n+1}\binom{2n}{n}$.

Special cases

- *A* **degenerated tree** *is a BT where for each parent node, there is only one associated child node (⇒ in performance measures, the BT behaves like a linked list).*
- *A* **full binary tree** *is a BT in which every node has zero or two children.*
- *A* **complete binary tree** *is a full BT in which all leaves are at the same depth.*
- *A* **perfect binary tree** *is a BT for which all levels are complete, but possibly its last level (in this case, the leaves are grouped at the left).*

- How many degenerated BTs of size 3?
- How many full BTs of sizes 3, 4 and 5?
- Give a complete BT of size 7.
- Give a perfect BT with 5 nodes.
- Give the total number of nodes in a complete BT of depth n.
- Prove that, for a BT of n nodes, its depth h verifies:

$$\lfloor lg\ n \rfloor \leq h \leq n - 1$$

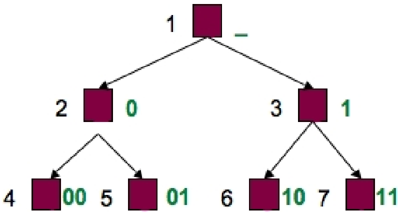
.

Occurrences and hierarchical numbering

Occurrence of a node: a string of 0 and 1, which characterizes the path from the root to that node.

- The occurrence of the root is the empty string.
- If the occurrence of a node is μ, its left child's occurrence is $\mu 0$, its right child's occurrence is $\mu 1$.

In a complete binary tree, the **hierarchical numbering** attributes an increasing natural number (beginning with 1) all nodes from the root, level after level, and from the left to the right on each level.

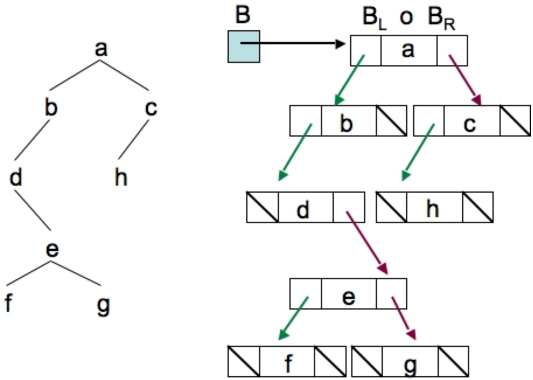


Let consider a node with number i, its left child has number $2i$ and its right child $2i + 1$.

Prove that if a node in a complete tree has occurrence μ and for hierarchical numbering i, then $i = 2^{\lfloor lg\, i \rfloor} + m$, where m is the integer which binary representation is μ.

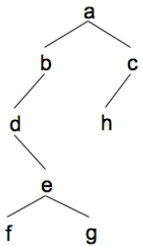
Representations of binary trees

Reproducing the recursive definition of BT:



Representations of binary trees

Or using an array:

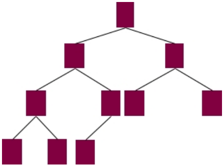


B

$\boxed{2}$

	data	B_L	B_R
0			
1	d	0	9
2	a	4	5
3	g	0	0
4	b	1	0
5	c	11	0
6			
7	f	0	0
8			
9	e	7	10
10	g	0	0
11	h	0	0
12			
13			
14			

Storing perfect binary trees



At most one internal node with a unique left sub-tree and this node is on the last level but one.

Compact sequential representation based on the hierarchical numbering: If a node is numbered i, its left child is numbered $2i$, its right child $2i + 1$.

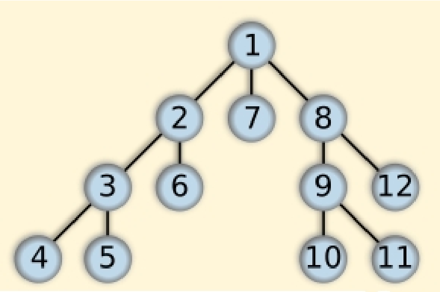
Proof by induction.

Using the hierarchical numbering:

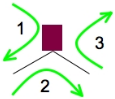
- $2 \leq i \leq n \Rightarrow$ the father of node i is $i \operatorname{div} 2$
- $1 \leq i \leq n \operatorname{div} 2 \Rightarrow$ the left child of node i is $2i$, its right child is $2i + 1$

Note: this representation can be used also for general BT. What happens *e.g.* for a degenerated tree?

Traversing trees
Depth-first traversal



Depth-first traversal




```
FUNCTION traverse(BT A){
  TREATMENT1
  IF (A.left<>null) traverse(A.left)
  TREATMENT2
  IF (A.right<>null) traverse(A.right)
  TREATMENT3
}
```

- pre-order (prefix): only `TREATMENT1`
- in-order (infix): only `TREATMENT2`
- post-order (suffix): only `TREATMENT3`

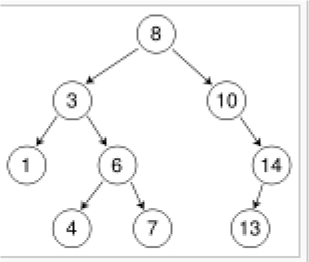
Note that one cannot recover the hierarchical numbering with this traversal.

Binary Search Trees (BST)

Binary tree data structure such that a total order is defined on the values attached to the nodes and:

- left subtree of a node contains only values less than the node's value;
- right subtree of a node contains only values greater than or equal to the node's value.

An example (*from Wikipedia*)



$\Longrightarrow$ related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Pseudo-code for the search in a BST

```
FUNCTION search(BT A,Integer val):Boolean{
  IF (A=null) RETURN false
  ELSE IF (A.value<val)
     RETURN search(A.right,val)
  ELSE IF (A.value>val)
     RETURN search(A.left,val)
  ELSE IF (A.value=val) RETURN true
  ELSE RETURN false
}
```

What is the worst case for this search procedure?
Write the pseudo-code for the insertion of a new value in a BST.

Outline

1 Introduction

2 Sequential data structures

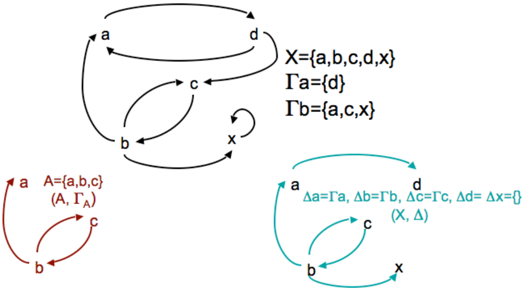
3 Binary Trees

4 Graphs
- Basic definitions
- Abtract data type
- Data structures
- Exploring graphs
- Topological sorting
- (Strongly) connected components

A huge number of real life problems expressed in terms of **relational structures**.

A **graph** $G = (X, \Gamma)$ *is defined by a set* X *(of vertices) and a function* $\Gamma :\to X$ *(the arcs).*

Alternatively a graph is denoted $G = (X, E)$, where E is the set of arcs.
A subgraph of $G = (X, \Gamma)$ is a graph (A, Γ_A) where $A \subset X$ and Γ_A defined by $\forall x \in A, \Gamma_A x = \Gamma x \cap A$
A partial graph of $G = (X, \Gamma)$ is a graph (X, Δ) where $\forall x, \Delta x \subset \Gamma x$.



Given (X, E),

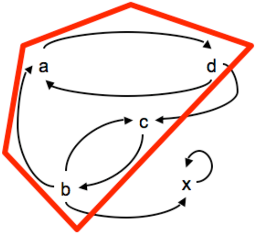
- for an arc $u = (x, y) \in E$, x is the **initial vertex (source)**, y the **terminal vertex (target)**, (y is said to be a **successor** of x),
- two arcs are **adjacent** if they are different and share a common vertex,
- two vertices $x, y \in X$ are **adjacent** if $x \neq y$ and $(x, y) \in E$ or $(y, x) \in E$,
- the **indegree** $deg_i(x)$ of $x \in X$ is the cardinal of $\{(y, x) \in E\}$,
- the **outdegree** $deg_o(x)$ of $x \in X$ is the cardinal of $\{(x, y) \in E\}$,
- the **degree** of x is $deg(x) = deg_i(x) + deg_o(x)$.

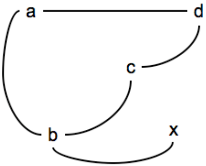
Given (X, E),

- a **path** is a sequence $(u_1, \dots u_n)$ of arcs in E, s.t. the target of u_i is the source of u_{i+1} $(i = 1 \dots n-1)$,
- the length of a path is the number of its arcs,
- a path $(u_1, \dots u_n)$ is **simple** if $u_i \neq u_j$, $\forall i, j = 1, \dots n, i \neq j$, otherwise, it is **composite**,
- alternatively a path $(u_1, \dots u_n)$ which meets the vertices $x_1, \dots x_{n+1}$ is denoted $[x_1, \dots x_{n+1}]$,
- a path is **elementary** if is does not meet the same vertex twice,
- a **circuit** is a finite path $[x_1, \dots x_k]$ in which $x_1 = x_k$,
- a **loop** is a circuit of length 1 (a single arc (x, x)),
- if Γ is reflexive (*i.e.* $(x, y) \Rightarrow (y, x) \in E$, the graph is said **non-oriented** or **symmetric**,
- an **edge**, is a set of two vertices $\{x, y\}$ s.t. $(x, y) \in E$ or $(y, x) \in E$) $\longrightarrow$ chains and cycles.

Given $G = (X, E)$,

- G is **complete** if $(x, y) \notin E \Rightarrow (y, x) \in E$,
- G is **strongly connected** if $\forall x, y \in X$ there is a **path** joining x and y,
- G is **connected** if $\forall x, y \in X$ there is a **chain** joining x and y,





- a **tree** is a connected non-oriented graph without cycle,
- a **root** in an oriented graph is a vertex r s.t. every vertex can be reached from r,
- an **arborescence** is an oriented graph which has a root and s.t. the corresponding non-oriented graph is a tree.

Given a graph $G = (X, E)$, non-oriented with $|X| = n$, the following properties are equivalent:

1. *G is connected without cycle (a tree),*
2. *G is connected and if an edge is deleted it is no more connected,*
3. *G is connected and has $n-1$ edges,*
4. *G has no cycle, and the addition of one edge creates a cycle,*
5. *G has no cycle and has $n-1$ edges,*
6. *all pair of vertices is connected by a unique chain.*

To specify a graph, give: the set of vertices and the set of arcs (pairs of vertices).

vertices are arbitrary numbered

Basic operations over vertices:

node : integer $\longrightarrow$ vertex
arc : vertex,vertex $\longrightarrow$ Boolean
num : vertex $\longrightarrow$ integer
deg_o : vertex $\longrightarrow$ integer
ith_succ : vertex, integer $\longrightarrow$ vertex

by convention, successors of a vertex are numbered in an increasing order: $i < j \Rightarrow num(ith\_succ(x, i)) < num(ith\_succ(x, j))$.

Scheme often encountered to process all successors of a vertex x:

```
FOR i=1 TO deg_o(x)
    process(ith_succ(x,i))
```

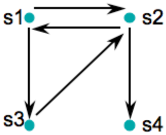
When the graph can evolve, one has to consider

Basic operations over the graph:

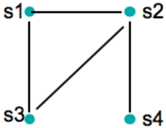
card : graph $\longrightarrow$ integer
empty_graph : $\longrightarrow$ graph
add_node : graph $\longrightarrow$ graph
add_arc : vertex,vertex,graph $\longrightarrow$ graph
arc : vertex,vertex,graph $\longrightarrow$ Boolean
deg_i : vertex, graph $\longrightarrow$ integer
ith_succ : vertex,integer,graph $\longrightarrow$ vertex

Using contiguous representations (arrays)

called **adjacency matrix**

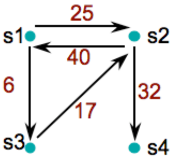


F	T	T	F
T	F	F	T
F	T	F	F
F	F	F	F



F	T	T	F
T	F	T	T
T	T	F	F
F	T	F	F

The case of weighted graphs

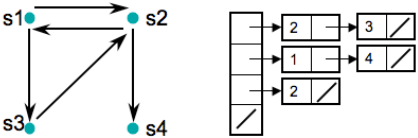


-1	25	6	-1
40	-1	-1	32
-1	17	-1	-1
-1	-1	-1	-1

space in $\theta(n^2)$ with $n = card(G)$

Using linked structures (lists)

Using the lists of successors for each vertex (called adjacency lists)

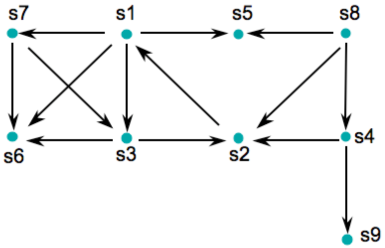


space in $\Theta(n+p)$ with $n = card(G)$, $p = \sum_{i=1\ldots n} deg\_o(node(i))$

Depth First Search, recursive version

```
FUNCTION dfs(Graph G){
  FOR i=1 to card(G)
     mark[i]=false
  FOR i=1 to card(G)
     IF NOT(mark[i])
          dfs_visit(node(i))
}
```

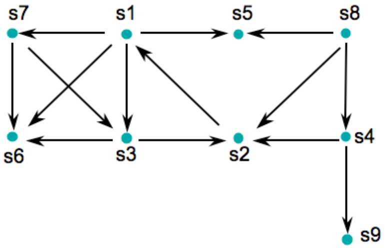
```
FUNCTION dfs_visit(Vertex v){
  mark[num(v)]=true
  FOR j=1 to deg(v)
     s=ith_succ(v,j)
     k=num(s)
     IF NOT(mark[k])
          dfs_visit(s)
}
```



Depth First Search, recursive version

```
FUNCTION dfs(Graph G){
  FOR i=1 to card(G)
     mark[i]=false
  FOR i=1 to card(G)
     IF NOT(mark[i])
          dfs_visit(node(i))
}
```

```
FUNCTION dfs_visit(Vertex v){
  mark[num(v)]=true
  FOR j=1 to deg(v)
     s=ith_succ(v,j)
     k=num(s)
     IF NOT(mark[k])
          dfs_visit(s)
}
```



s1, s3, s2, s6, s5, s7, s4, s9, s8

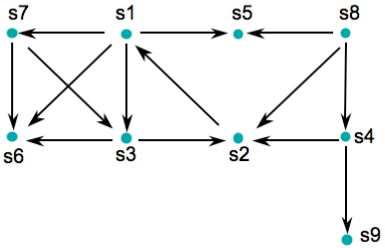
Complexity analysis:

- adjacency matrix: $\Theta(n^2)$
- successors lists: $\Theta(max(n, p))$

```
FUNCTION dfs_visit(Vertex v){
  mark[num(v)]=true
  *** PROCESS1(v) ***
  FOR j=1 to deg(v)
     s=ith_succ(v,j)
     k=num(s)
     IF NOT(mark[k])
        dfs_visit(node[k])
  *** PROCESS2(v) ***
}
```

Two classical orders for graph exploration (as for trees):

- prefix order (PROCESS1): **s1, s3, s2, s6, s5, s7, s4, s9, s8**
- suffix order (PROCESS2): **s2, s6, s3, s5, s7, s1, s9, s4, s8**



Arc classification

A **spanning tree** of a connected graph G is:

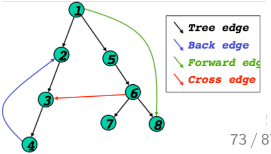
- (informally) a selection of edges that form a tree spanning every vertex,
- a maximal set of edges of G that contains no cycle,
- a minimal set of edges that connect all vertices,

DFS produces a spanning tree (or a forest if G is not connected) and allows a classification of the arcs:

- forward edges from a node to one successor,
- backward edges from a node to one predecessor,
- cross edges none of the previous ones,
- tree edges belong to the spanning tree itself, classified separately from forward edges.

If the graph is non-oriented, all of its edges are tree or backward edges.

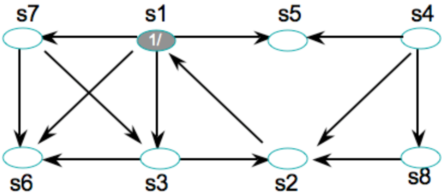
A graph G is acyclic iff dfs does not generate any backward edge.



Adapted algorithm (from Cormen et al.)

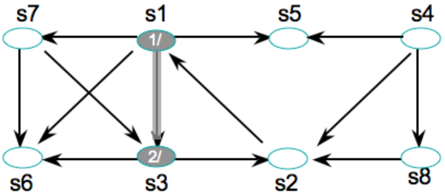
```
DFS(G)
for each vertex u ∈ V[G]
     do color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
     do if color[u] = WHITE
          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY             ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]         ▷ Explore edge (u, v).
     do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK            ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



Adapted algorithm (from Cormen et al.)

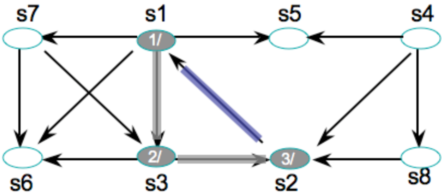
```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
         then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY            ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]        ▷ Explore edge (u, v).
    do if color[v] = WHITE
         then π[v] ← u
              DFS-VISIT(v)
color[u] ← BLACK           ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



Adapted algorithm (from Cormen et al.)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

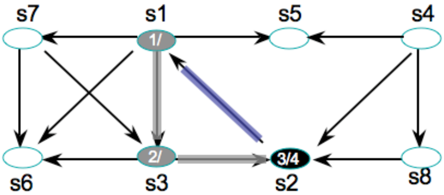
```
DFS-VISIT(u)
color[u] ← GRAY              ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]          ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK             ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



Adapted algorithm (from Cormen et al.)

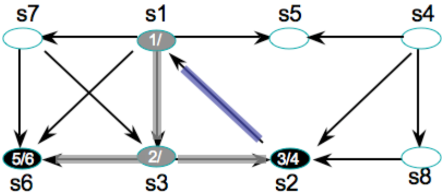
```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY            ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]        ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK           ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



Adapted algorithm (from Cormen et al.)

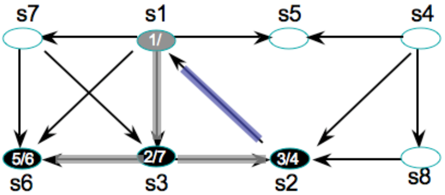
```
DFS(G)
for each vertex u ∈ V[G]
      do color[u] ← WHITE
         π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY              ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]          ▷ Explore edge (u, v).
      do if color[v] = WHITE
            then π[v] ← u
                 DFS-VISIT(v)
color[u] ← BLACK             ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



Adapted algorithm (from Cormen et al.)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

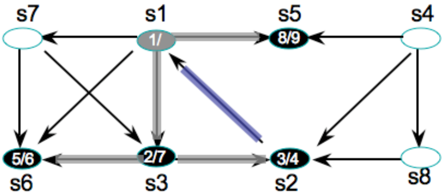
```
DFS-VISIT(u)
color[u] ← GRAY            ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]        ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK           ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



Adapted algorithm (from Cormen et al.)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

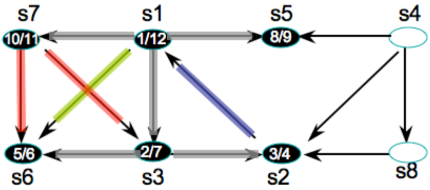
```
DFS-VISIT(u)
color[u] ← GRAY              ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]          ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK             ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```



Adapted algorithm (from Cormen et al.)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY              ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]          ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK             ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

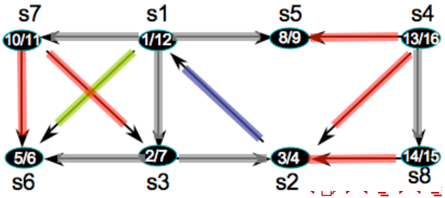


Adapted algorithm (from Cormen et al.)

```
DFS(G)
for each vertex u ∈ V[G]
     do color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
     do if color[u] = WHITE
           then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY              ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]          ▷ Explore edge (u, v).
     do if color[v] = WHITE
           then π[v] ← u
                DFS-VISIT(v)
color[u] ← BLACK             ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

(u, v) is a forward edge (grey or green) if $d[u] < d[v]$

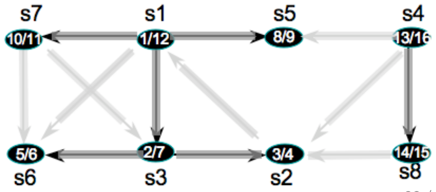
(u, v) is a cross edge (red) if $d[u] > d[v]$



Adapted algorithm (from Cormen et al.)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY               ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]           ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK              ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

$G_\pi = (S, A_\pi)$, with $A_\pi = \{(\pi[v], v),\ v \in S$ and $\pi[v] \neq NIL\}$ is the spanning forest generated by dfs



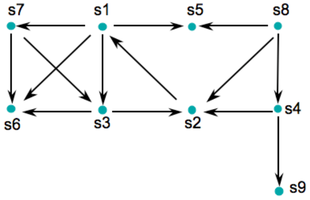
Breadth First Search

Unlike dfs, bfs is not naturally recursive.

Uses a `Queue` (a list with a FIFO policy) with the basic operations:

- `empty(Q)` is true if Q is empty, false otherwise
- `first(Q)` returns the first element of the queue (here a vertex)
- `dequeue(Q)` removes the first element of the queue
- `enqueue(Q,s)` adds the vertex s at the end of the queue

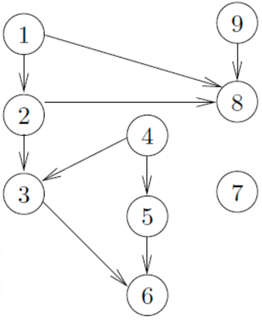
```
FUNCTION bfs(Vertex v){
   Q an empty queue of vertices
   mark[v]=true
   enqueue(Q,v)
   WHILE NOT(empty(Q))
     x=first(Q)
     dequeue(Q)
     FOR i=1 TO deg_o(x)
        y=ith_succ(x,i)
        j=num(y)
        IF NOT(mark[j])
           mark[j]=true
           enqueue(Q,y)
}
```

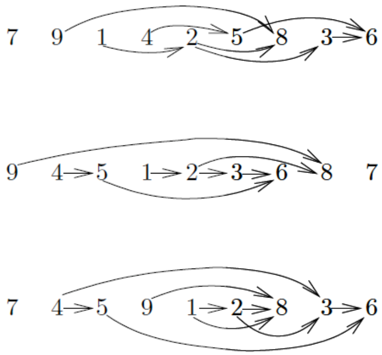


Topological sorting An oriented acyclic graph (or DAG) is a convenient way to represent precedence constraints.

An oriented graph G is acyclic iff dfs on G generates no backward arc.

what is a feasible ordering?



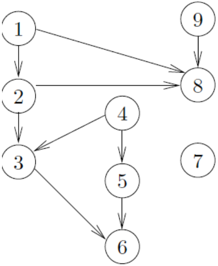


Topological sorting

Modify dfs by adding nodes at the top of a stack when their processing is finished.

→ **decreasing order of dates** `f[v]`

```
FUNCTION dfs_visit(Vertex v){
    mark[num(v)]=true
    FOR j=1 to deg(v)
      s=ith_succ(v,j)
      k=num(s)
      IF NOT(mark[k]) dfs_visit(s)
    push(Q,v)
}
```



DFS can be easily used to determine the connected components of a non-oriented graph (see exercice)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK
f[u] ← time ← time + 1
```

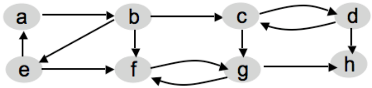
```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times f[u] for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
          in order of decreasing f[u] (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
          separate strongly connected component
```

DFS can be easily used to determine the connected components of a non-oriented graph (see exercice)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK
f[u] ← time ← time + 1
```

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times f[u] for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
         in order of decreasing f[u] (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
         separate strongly connected component
```



DFS can be easily used to determine the connected components of a non-oriented graph (see exercice)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK
f[u] ← time ← time + 1
```

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times f[u] for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
        in order of decreasing f[u] (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
        separate strongly connected component
```