Algorithmic Design

LEZIONI	LUN	MAR	MER	GIO
9 - 11	alg design		stat ML	alg design
11 - 13	stat methods	stat methods	stat ML	stat ML
14 - 16	big data	big data		stat ML
16 - 18	big data	big data		stat methods

Alberto Casagrande
acasagrande@units.it
ricevimento: room 330 h2bis, monday and tuesday, 11-12

book: *Introduction to algorithms*, Cormen

moodle
key: Dijkstra

05/03/18

Algorithms

low level descriptions of well defined computational procedures meant to be automatized.

Computational model = single processor single thread model (RAM)
- infinite memory
- infinite variables
- basic algebraic operations
- basic algebraic relations

We will assume constant execution time for each of the above operations.

pseudo-code rules:
- if then, while, for, repeat statements
- ← denotes assignments
- blocks are identified by indentation
- array indexes begin from 1 (not 0)
- parameters are passed by value

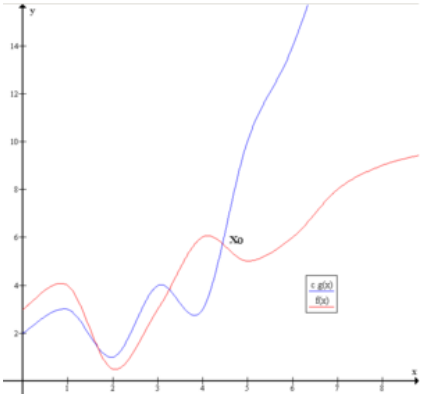
Big-o notation

speedup theorem
given any real $c > 0$ and any Turing machine solving a problem in time f(n), there is another machine that solves the same problem in time at most $cf(n) + n + 2$.

This means that constants do not matter, so we end up considering asinthotic time complexity.

$$\textbf{def. } O(f) = \{g | \exists\, c > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 \ |g(n)| \leq c|f(n)|\}$$

for example, $n + 1 \in O(n) \subset O(n^2) \subset O(n^{n!})$



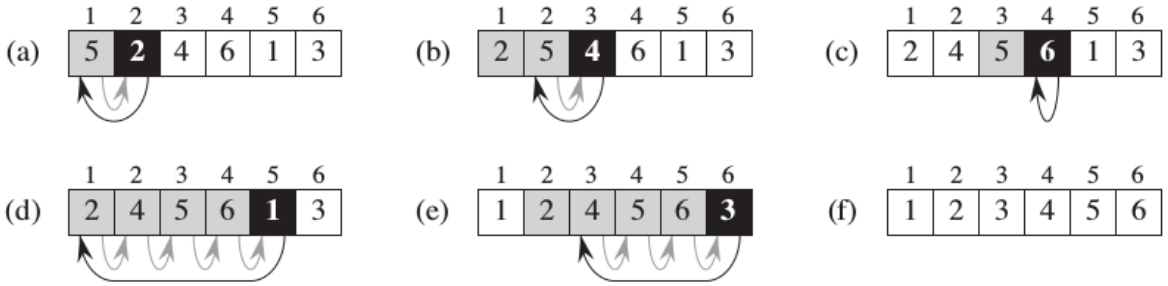
def. $\Omega(f) = \{g | \exists\, c > 0, n_0 \in \mathbb{N} : \forall n \geq n_0 \; |g(n)| \geq c|f(n)|\}$
$\Theta(f) = O(f) \cap \Omega(f)$

Sorting algorithms

p.26 Cormen

input is an array A
output is A such that for each $i \in [1, |A|)$, $A[i] \leq A[i+1]$

Insertion sort



insertion sort

Can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.

```
insertion_sort(A)
  for j ← 2 to |A| do                  # |A|-1 times
      k ← A[j]                         # O(1)
      i ← j − 1                        # O(1)
      while i > 0 and A[i] > k do      # at most j-1
          A[i + 1] ← A[i]              # O(1)
          i ← i − 1                    # O(1)
      A[i + 1] ← k                     # O(1)
```

Let's compute the complexity of this algorithm:

$$T(|A|) = \sum_{j=2}^{|A|} \sum_{i=1}^{j-1} (O(1) + O(1)) = \sum_{j=2}^{|A|} \sum_{i=1}^{j-1} O(1)$$

$$= \sum_{j=2}^{|A|} O(j-2) = \sum_{j=2}^{|A|} O(j) = O(\tfrac{|A|(|A|+1)}{2}) = O(|A|^2)$$

Class	Sorting algorithm
Data structure	Array
Worst-case performance	O(n^2) comparisons, swaps
Best-case performance	O(n) comparisons, O(*1*) swaps
Average performance	O(n^2) comparisons, swaps
Worst-case space complexity	O(n) total, O(*1*) auxiliary

insertion sort summary

08/03/18

Select problem

→ p.220, 302-308 Cormen
→ time_bounds_selection.pdf

Given an array A we want to find the j-th element of the sorted A, without sorting it.

- input: an unsorted array A and an index $j \in [1, |A|]$
- output: a value $v \in A$ such that A contains $j-1$ elements smaller than v

example
input: $A = [5, 2, 4, 6, 1, 3]$ and $j = 5$
output: 5

Quickselect algorithm

Median of the medians

Since the complexity of sorting is $O(nlogn)$, we aim at having a faster algorithm, maybe $O(n)$.
A possible strategy is the following one:

1. split A in $\lceil \frac{n}{5} \rceil$ blocks of dimension 5 in $O(n)$
2. compute the block medians in $O(n)$
3. compute the median of all the medians (pivot) recursively
4. compute $[k_1, k_2]$ such that for all $k \in [k_1, k_2]$ $A[k]$ is equal to the median of the medians
5. if $j \in [k_1, k_2]$ then $A[j]$ is the median of the medians, otherwise:
   a. if $j < k_1$ go back to 4. using $[0, k_1)$
   b. if $j > k_2$ go back to 4. using $(k_2, |A|]$

alternative definition:

4. given the interval $[k_1, k_2]$, compute $k \in [k_1, k_2]$ such that $A[k]$ is equal to the median of the medians
5. if $j = k$ then $A[j]$ is the median of the medians, otherwise:
   a. if $j < k$ go back to 4. using $[0, k-1]$
   b. if $j > k$ go back to 4. using $[k+1, |A|]$

One iteration on a randomized set of 100 elements from 0 to 99

	12	15	11	2	9	5	0	7	3	21	44	40	1	18	20	32	19	35	37	39
	13	16	14	8	10	26	6	33	4	27	49	46	52	25	51	34	43	56	72	79
Medians	17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	83
	22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
	96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91

median of medians

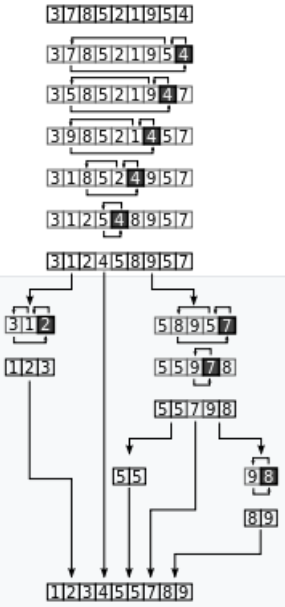
The selection algorithm becomes:

```
select(A, j, begin=1, end=|A|)
  // reorder the array
  // sto supponendo che j sia 140...
  if (end-begin+1 < 140)                          // O(1)
    insertion_sort(A, begin, end)                 // O(|A|)
    return A[j]
  // find the median of medians
  pivot = select_pivot(A,begin,end)               // O(|A|/5) + T(|A|/5)
  // tri_partition partitions A[] in three parts:
  // 1) A[begin,...,k_1-1] contains all elements smaller than the pivot
  // 2) A[k_1,..., k_2] contains all occurrences of the pivot
  // 3) A[k_2+1,...,end] contains all elements greater than the pivot
  // finally it returns k_1 and k_2
  (k_1, k_2) = tri_partition(A,begin,end,pivot)   // O(|A|)
```

```
  // recursively call select in the correct interval
  if j < k1
    return select(A,j,begin,k1 − 1)
  if j > k2
    return select(A,j,k2 + 1, end)
  return A[j]
```

some observations on the algorithm:

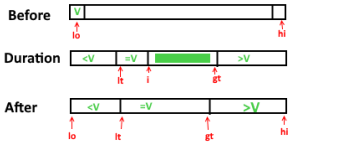
- `select_pivot` finds the pivot, which is the median of the block of medians.
- `tri_partition` takes all the elements in A smaller than the pivot at the begin of A and those greater at end
- Median of medians can also be used as a pivot strategy in quicksort, yielding an optimal algorithm, with worst-case complexity O(*n* log *n*).
- in this case the complexity of insertion sort is O(1) because we used the const value 140.



quicksort

Select pivot algorithm

```
select_pivot(A, begin, end)
  num_of_blocks <- (end-begin+1)/5          # O(1)
  # initialize medians array
  medians <- array[num_of_blocks]           # O(|A|)
  # compute the medians for each block
  for i in [0,num_of_blocks]                # O(|A|/5) = O(|A|)
    # beginning point for each block
    cbegin = begin + 5 * i                  # O(1)
    # reorder the block
    insertion_sort(A, cbegin, cbegin+4)     # O(1)
    # add the i-th median to the array
    medians[i+1) <- A[cbegin+2]             # O(1)
  # find the median of medians
  return select(medians, j, num_of_blocks, (1+num_of_blocks)/
2)) # T(|A|/5)
```

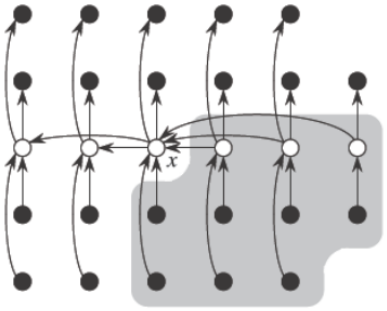


3 - way partitioning overview

the whole complexity of *select_pivot* is
$T(n) = T(\lceil n/5 \rceil) + O(n/5) = T(\lceil n/5 \rceil) + O(n).$

Let x be the pivot (median of medians) and notice that:

- *at least* half of the medians for each block are $\geq x$
- the blocks are $\lceil \frac{n}{5} \rceil$
- for each block there are at least 3 elements $\geq x$
- you have to exclude only two cases: the block containing x and the block having less than 5 elements (if 5 does not exactly divide n)



Considering all of this, the number of elements greater than pivot is at least $3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3}{10}n - 6$. The same inequality holds for the number of elements lower than the pivot.

So the complexity for *select* is

$$T(n) = \begin{cases} O(1) & n < 140 \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n + 6) + O(n) & n \geq 140 \end{cases}$$

> ***Theorem.*** $T(n) \in O(n)$

proof. (by induction) *p.222 Cormen*
We want to prove that there exists a $c > 0$ such that $T(n) \leq cn \ \ \forall n > 0$.

Let's choose c such that $T(n) \leq cn \; \forall n < 140$ and a s.t. $O(n) \leq an \; \forall n > 0$. For each $n \geq 140$ we have
$T(n) \leq c\lceil \frac{n}{5} \rceil + c(\frac{7}{10}n + 6) + a\,n$
$\quad \leq c\,n/5 + c + 7c\,n/10 + 6c + a\,n$
$\quad \leq c\,n + (-c\,n/10 + 7c + an)$
If the second portion is ≤ 0 we get $T(n) \leq c\,n$.
This is true because $n \geq 140 > 70$, so the inequality holds iff $c \geq 10an/(n - 70)$. If we choose $c \geq 20a$, noticing that $n/(n - 70) \geq 2$ we get the inequality. ■

If the complexity of tri_partition is O(n) then the complexity of elect is O(n).

12/03/18

Strassen's algorithm

→ p.735-741 Cormen

Given two matrices A and B, we can divide both in 4 parts, which in total are 8 blocks:

$$A = \begin{pmatrix} A_{11} A_{12} \\ A_{21} A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} B_{12} \\ B_{21} B_{22} \end{pmatrix}$$

the method:
1. Split A and into 8 submatrices;
2. Compute 10 intermediate matrices by summing or subtracting submatrices from step 1. (S1, S2, . . . , S10);
3. Compute 7 matrices: P1, . . . , P7, by multiplying S1, . . . , S10;
4. Compute C11, . . . , C22 by summing/subtracting P1, . . . , P7.

The complexity cost is:
1. $O(1)$ using indexes
2. for each intermediate matrix is: $O(\frac{n^2}{2})$, which is equivalent to $O(n^2)$;
3. 7 times the cost of multiplying each sub-matrix: $7T(\frac{n}{2})$;
4. $O(n^2)$

where

$$T(n) = \begin{cases} O(1) & n > 1 \\ 7T(\frac{n}{2}) + O(n^2) \end{cases}$$

Setting $a = 7, b = 2$ and using master theorem we get $T(n) \in O(n^{log_2 7})$.

The matrices are:
$S_1 = B_{12} - B_{22}$
$S_2 = A_{11} + A_{12}$
$S_3 = A_{21} + A_{22}$
$S_4 = B_{21} - B_{11}$
$S_5 = A_{11} + A_{22}$
$S_6 = B_{11} + B_{22}$
$S_7 = A_{12} - A_{22}$
$S_8 = B_{21} + B_{22}$
$S_9 = A_{11} - A_{21}$
$S_{10} = B_{11} + B_{12}$

$P_1 = A_{11} S_1$
$P_2 = S_2 B_{22}$
$P_3 = S_3 B_{11}$
$P_4 = A_{22} S_4$
$P_5 = S_5 S_6$
$P_6 = S_7 S_8$
$P_7 = S_9 S_{10}$

$C_{11} = P_5 + P_4 - P_2 + P_6$
$C_{12} = P_1 + P_2$
$C_{21} = P_3 + P_4$
$C_{22} = P_5 + P_1 - P_3 - P_7$

- [ ] prove the equations and implement the code with matrices of size $\neq 2^n$

15/03/18

Graph algorithms

Strongly connected components

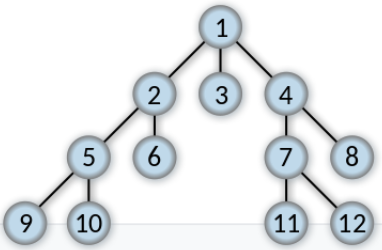
We have two main representations for graphs (V, E):

- **Adjacency list**: map from nodes to the list of their ~~successors~~ adjacent nodes
  $Adj[v] \longrightarrow$ list of nodes $w \in V$ such that $(v, w) \in E$
- **Adjacency matrix**: 0/1 matrix whose rows/columns correspond to the nodes of the graph. In position (i, j) we have 1 iff $(i, j) \in E$.

Breadth first search

→ p.531-539 Cormen

- Used for distance
- it uses a queue (FIFO)



BFS

```
BFS(G,v)                          # O(|V|+|E|)
  for w in V do  G = (V,E)         # O(|V|)
    color[w] <- white
    d[w] <- ∞
    π[w] <- NIL
  d[v] <- 0                       # O(1)
  color[v] <- grey                # O(1)
  Enqueue(Q,v)                    # O(1)
  while |Q| ≠ 0 do                # O(|E|)
     for w in Adj[head(Q)]
       if color(w) == white then
         d[w] <- d[head(Q)]+1
         π[w] <- head(Q)
         color[w] <- grey
         Enqueue(Q,w)
    color[head(Q)] <- black
    Dequeue(Q)
```

So total time complexity is $O(|V| + |E|)$, since every vertex and every edge will be explored in the worst case.
Notice that $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$ depending on how sparse the input graph is.

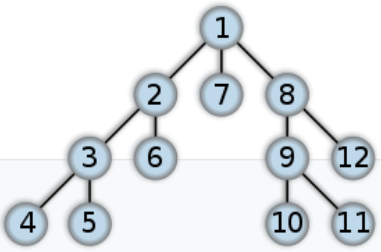
Depth first search

→p.540-549 Cormen

Used for connectivity

We will have:

- $d : V \to \mathbb{N}$ discovery tree (preordering)
- $f : V \to \mathbb{N}$ final processing tree (postordering)
- $\pi : V \to \mathbb{N}$



DFS

```
DFS(G)                              # O(|V|+|E|)
  for w in V do                     # O(|V|)
    color[w] <- white
    π[w] <- nil
    d[w] <- ∞
    f[w] <- ∞
  time <- 0
  for v in V do                      # O(|V|)
    if color[v] == white
      time <- DFS_real(G,v,time)
```

where

```
DFS_real(G,v,time)
  color[v] <- grey
  d[v] <- time
  time = time + 1
  for w in Adj[v] do                # O(|Adj[v]|)
    if color[w] == white
      π[w] <- v
      time <- DFS_real(G,w,time)
  f[v] <- time
  time <- time + 1
  return time
```

and the total complexity is s $O(|V| + |E|)$.

Theorem. Let us consider the two intervals $I_v = [d[v], f[v]]$ and $I_w = [d[w], f[w]]$. Either:

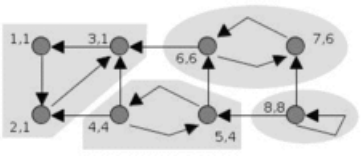
- $I_v \cap I_w = \emptyset$
- $I_v \subseteq I_w$ and w is an ancestor of v in the DFS tree
- $I_w \subseteq I_v$ and v is an ancestor of w in the DFS tree

Tarjan's algorithm for SCC

→ p.615 Cormen
→ tarjan.pdf

It is a sort of DSF + a map lowlink that remembers the smallest $d[v]$ among all the w reachable from v.

d = discovery time
f = final processing time



Tarjan's Algorithm Animation.gif

The algorithm takes a directed graph as input, and produces a partition of the graph's vertices into the graph's strongly connected components. Each vertex of the graph appears in exactly one of the strongly connected components. Any vertex that is not on a directed cycle forms a strongly connected component all by itself.

Each node v is assigned a unique integer $d[v]$, which numbers the nodes consecutively in the order in which they are discovered. It also maintains a value $lowlink[v]$ that represents the smallest index of any node known to be reachable from v through v's DFS subtree, including itself. Therefore if $lowlink[v] = d[v]$ at the end of the recursive call over v, all the nodes reachable from v (e.g. those in the same SSC) have already been completed.

```
Tarjan_SCC(G)
  # initialization
  for v in V    # G(V,E)
    d[v] <- ∞  # f[v] <- ∞
    lowlink[v] <- ∞
    color[v] <- white # v has not been visited
  Q <- [] # successors queue
  SCC <- []
  time <- 0
  for v in V
    if color[v] == white
      time <- Tarjan_SCC_real(G,v,time,Q,SCCs)
  return SCCs
```

Next time we'll define `Tarjan_SCC_real`.

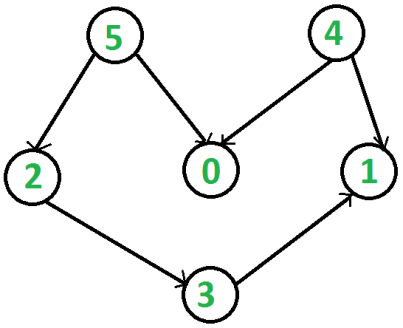
19/03/18

```
Tarjan_SCC_real(G,v,time,Q,SCCs)
  # update v infos
  d[v] <- time
  lowlink[v] <- time
  time <- time + 1
  color[v] <- grey
  Enqueue(Q,v)
  for w in Adj[v] do
    if color[w] == white  # w has not been visited
       time <- Tarjan_SCC_real(G,w,time,Q,SCCs)
       lowlink[v] <- min(lowlink[v], lowlink[w])
    else
       if color[w] == grey  # w has already been visited
         lowlink[v] <- min(lowlink[v], lowlink[w])
  if lowlink[v] == d[v] then
    # v is a representative for a new SCC
    SCC <- []
    while Head(Q) != v do
      SCC.append(Head(Q))
      Dequeue(Q)
    SCC.append(SCC)
  return time
```

The whole complexity of `tarjan_SCC_real` is $O(|V|)$, so the complexity of `tarjan_SCC` is $O(|V| + |E|)$.
We cannot perform better than this algorithm for SCCs, because it has the same complexity of reading the whole graph.

Topological Sort

→ p. 612 Cormen
Definition of Strongly Connected Components. Tarjan's algorithm to compute the strongly connected components of a graph. Intuition, code, and complexity analysis. Definition of topological sort and how to compute it.



DAG

Given a graph $G = (V, E)$ we want to find a sorting for V (i.e. a map $S : V \to \mathbb{N}$) such that if $v \rightsquigarrow_G w$ (w is reachable from v) then $S(v) \subseteq S(w)$.

For example, topological sorts of this graph is
(5 4 2 3 1 0) or (4 5 2 3 1 0).

Notice that:

- Topological Sorting for a graph is not possible if the graph is not a **Directed Acyclic Graph**
- There can be more than one topological sorting for a graph

Def. The **collapsed graph** of a graph $G = (V, E)$ is the graph $G_{SCC} = (SCCs, \{E(SCC_1, SCC_2) : \exists v \in SCC_1, \exists w \in SCC_2\, (v, w) \in E\})$.

DFS has an important feature which is the final processing time f, and we can use it because the largest f corresponds to the lowest processing index in the topological sort. So we can use DFS and inherit the sorting induced by f. This is true only for acyclic graphs and takes $O(|V| + |E|)$.

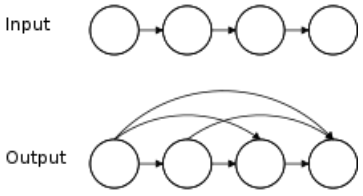
Prop. $v \rightsquigarrow_G w \Leftrightarrow f[v] \geq f[w]$, where f is computed by DFS.

In the case of a cyclic directed graph you can compute Tarjan with more than one node.

Transitive closure problem

→ *fischer_meyer_trans_closure.pdf*
→ *farzan_trans_closure.pdf*

Definition of transitive closure of graph. Reduction of the transitive closure problem to Boolean matrix multiplications. Fischer's and Meyer's algorithm: intuition, code, and complexity.



Transitive closure

Given $G = (V, E)$ we want to compute $G^* = (V, E^*)$, where $E^* = \{(v, w) | v \rightsquigarrow_G w\}$.

We want to model it almost as a matrix multiplication problem using the adjacency matrix.

We will see how the problem of finding the transitive closure of a graph is equivalent to the problem of boolean multiplication.

22/03/18

A naive algorithm is to compute $v \rightsquigarrow_G w$ for each $v, w \in V$, which has a very bad cost of $O(|V|^2) \cdot O(|V| + |E|) = O(|V|^2 \cdot |E|)$.

Fischer and Meyer algorithm

(for transitive closure problem)

1. compute the SCC collapsed graph, and go from a general graph to an acyclic graph
2. sort the nodes of the new graph in a way in which $AdjM(G)$ is upper triangular, and this can be done by using topological sort. We know that if $S(v) \leq S(w)$then w is not reachable from v, thus $AdjM(G)$ is upper triangular.
3. Now we can split it in 4 blocks $M = AdjM(G) = \begin{pmatrix} A & C \\ 0 & B \end{pmatrix}$ so we get $M^* = AdjM(G^*) = M \cdot \ldots \cdot M = \begin{pmatrix} A^* & A^*CB^* \\ 0 & B^* \end{pmatrix}$ by using the transitive closure, not matrix multiplication.
4. Rebuild the transitive closure of G. This is easy because all the elements in a SCC have the same transitive reachability set.

Let's find the complexity of this algorithm:

1. is tarjan complexity $\Theta(|V| + |E|)$
2. is topological sort $O(|V|^2)$
3.
4. $O(|V|^2)$

$T(n)$ is the complexity of computing M^* where n is the size of the square matrix M, so

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + O(n^{log_2 7}) & n > 1 \end{cases}$$

by using master theorem we get $T(n) \in O(n^{log_2 7})$.

The overall complexity of Fischer-Meyer is $O(|V|^2) + O(n^{log_2 7}) = O(|V|^{log_2 7})$.
This is not the best algorithm possible to perform transitive closure, but probably is this the best for very large and dense graphs. In the case of small graphs it is better to avoid the calculation of $AdjM$.

```
collapse(G, MN)
# MN is an array of arrays, each of them represents a new node
  V2M <- zero(|V|)
  i <- 0
  # looping on all the nodes included into one single array
  while i < |MN| do
    for v in MN[i] do
      V < MN[v] <- i
    AdjR[i] <- []
    i <- i+1
  for v in range(1,|AdjG|) do
    for w in Adj(v) do
      AdjR[V2M[v]].append(V2M[])
  return AdjR
```

Adj list to adj matrix

```
Adj2AdjM(Adj)
# get the corresponding adj matrix
  M <- zero(|Adj|, |Adj|)
  for v in range (1, |Adj|) do
    for w in Adj[v] do
      M[v,w] <- 1
  return M
```

Upper triangular matrix transitivity closure:

```
UTMatrixTC(M)
# M is a square matrix
  n <- n_of_rows(M)
  if (log_2 n ∉ ℕ) then
    M <- fix_it(M)
  return UTMatrixTC_real(M)
```

```
UTMatrixTC_real(M)
  n <- n_of_rows(M)
  if (n==1) then
    return M
  (A, B, C) <- split_three_blocks(M,n)
  A* <- UTMatrixTC_real(A)
  B* <- UTMatrixTC_real(B)
  D <- MatrixMultiplication(A*,C)
```

```
  D <- MatrixMultiplication(D,B*)
  # replace elements ≠ 0 by 1
  D <- zero_one(D)
  return split_three_blocks(A*, B*, D)
```

```
Fischer_Meyer(G)
  SCCs <- TarjanSCCs(G)
  reverse(SCCs)
  AdjSCC <- collapse(G, SCCs)
  M <- Adj2AdjM(AdjSCC)
  M*<- UTMatrixTC(M)
  return decollapse_AdjM(M*, SCCs)
```

Notice that we have not implemented `decollapse_AdjM` yet.

→ leggere qui per capire transitive clousure

26/03/18

Weighted graphs

Definition of weighted graph and examples. Definition of the single source shorted path problem. Definition of (min/max)heap. Definition of binary heap; complexity of building a binary heap, inserting a new key, extracting the minimum, and decreasing a value. Fibonacci heap: complexity of building a Fibonacci heap, inserting a new key, extracting the minimum, and decreasing a value. Dijkstra's algorithm: intuition, code, limitations, and complexity.

→ *p.595-601 Cormen*
graphs + a weighted function $W : E \to \mathbb{R}$ labeling the edges

Single source shortest path problem

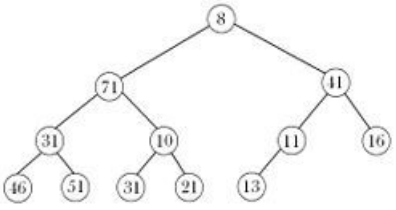
Given a weighted graph (G, W) and $s, d \in G$ we want to find a path $e_0, ...e_n$ from s to d such that:

1. $e_0 = (s, v)$ for some $v \in G$ (s is the source)
2. $e_i = (v_i, v_{i+1})\, e_{i+1} = (v_{i+1}, v_{i+2})$ for any $i \in [0, n-1]$
3. $e_n = (w, d)$ for some $w \in G$
4. $\sum_{i=0}^{n} w(e_i) = min \sum_{i=0}^{m} w(\bar{e_i})$ among all paths $\bar{e_0}, ..., \bar{e_m}$ satisfying 1., 2. and 3. (the path minimizes the sum of the distances from the initial node)

Minheap is an abstract data structure that provides fast:

1. insertion
2. min estraction
3. key decreasing
4. deletion

Minheaps are useful because they support *logarithmic* insertion and deletion and can be built in *linear* time. The i-th location in the array will correspond to a node located on level $\lfloor \log i \rfloor$ in the heap.
The most famous implementation is a **binary heap**, which is an almost complete tree.



Definition:

- A min-max heap is a *complete binary tree* containing alternating *min* (or *even*) and *max* (or *odd*) *levels*. Even levels are for example 0, 2, 4, etc, and odd levels are respectively 1, 3, 5, etc. We assume in the next points that the root element is at the first level, i.e., 0.
- Each node in a min-max heap has a data member (usually called *key*) whose value is used to determine the order of the node in the min-max heap.
- The *root* element is the *smallest* element in the min-max heap.
- One of the two elements in the second level, which is a max (or odd) level, is the greatest element in the min-max heap
- Let x be any node in a min-max heap.
  - If x is on a min (or even) level, then $x.key$ is the minimum key among all keys in the subtree with root x
  - If x is on a max (or odd) level, then $x.key$ is the maximum key among all keys in the subtree with root x.
- A node on a min (max) level is called a min (max) node.

Dijkstra's algorithm

(for sssp)

```
initialize_single_source(G, W, s)
  for v in G  // for every node in G
    d[v] <- ∞ // the distance of v from s is infinity
    π[v] <- null // the predecesor of v in "minimal path" from s is null
  d[s] <- 0
  π[s] <- s
```

```
Dijkstra(G,W,s)
  ~~(d,π) <-~~initialize_single_source(G,W,s)
  // build a minheap containing all the nodes of G
  H <- makeMinHeap(V, key = d)                    // O(|V|)
  while |H|>0 do                                  // O(|V|)
    // u is a node s.t. d[u]= min{d[v]:v in V}
    u <- extractMin(H)                            // O(log2(|V|))
    for w in Adj[u] do                            // Θ(|E|)
      if d[w] > d[u]+W[(u,w)]                     // Θ(1)
        decreaseKey(H,w,d[v]+W[(u,w)])            // O(log2(|V|))
        d[w] <- d[u]+W[(u,w)]                     // Θ(1)
        π[w] <- u                                 // Θ(1)
  return (d,π)
```

notice that:

- if we use a negative weight it converges but the result is wrong
- if s is disconnected from the graph it returns infinite distance

The cost of Dijkstra is:

- $O((|V|+|E|)log_2|V|)$ using binary heap
- $O(|V|log|V|+|E|)\Theta(1)$ using Fibonacci heap

05/04/18

Binary heap in c++

→ *binary_heap.cpp*

nel branch *binary_heap*

We proposed a complete C++ implementation for binary heaps. Both correcteness and complexity of the implemented methods were investigated. With the goal of implementing Dijkstra's algorithm, we also detailed associative binary heaps that allow to map the values insertion order into corresponding heap nodes.
The heapSort algorithm was implemented too.

complexity? depends. The worst case is i=0: visit a branch of the complete tree in O(log_2n)

09/04/18

continuazione del binary heap...
- [ ] scaricare il file da moodle

---

12 aprile

A^* algorithm (for sssp)

→ p.100-107 Cormen
→ minimum_cost_paths.pdf
A algorithm: main idea, pseudo-code, and complexity. Differences between Dijkstra's and A* algorithms: case studies.*

A^* = Dijkstra + heuristic distance
It consists in extracting from minHeap those nodes that minimize $d + h$.

- h is the heuristic distance, which estimates the distance between two nodes in the graph
- t is the destination
- d[v] is the current best distance between s and v
- $\bar{h}$[v] is the guest best distance between t and v
- π[v] is the predecessor of v in the best path from s

```
A*(G, W, h, s, t)
  for v in V
    d[v] <- ∞
    h̄[v] <- ∞
    π[v] <- None
    d[s] <- 0
    h̄[s] <- h(s,w)
    π[v] <- s
    H <- BuildHeap(V, key = h̄)
    while (|H|>0)
      z <- DeleteMin(H)
      if z == w
        return RebuildPath(π, w)
      for u in Adj[z]
        if d[u] > d[z] + w[z][v]
          π[u] <- z
          d[u] = d[z] + w[u][z]
          H.DecreaseKey(u, d[u]+h(u,w))
```

alternatively using the `closed` set

```
    while (|H|>0)
      z <- DeleteMin(H)
      closed <- closed U {z}
      if z == w
        return RebuildPath(π, w)
```

```
      for u in Adj[z]
        if d[u] > d[z] + w[z][v]
          if u ∈closed
            throw "your h sucks"
          d[u] = d[z] + w[u][z]
          π[u] <- z
          H.DecreaseKey(u, d[u]+h(u,w))
```

- [ ] controllare se ho scambiato u e v

The complexity is the same of Dijkstra.

Floyd-Warshall algorithm

(for APSP)
→ *p.570-576 Cormen*
→ *boolean_matrices.pdf*

Definition and naive solution based on Dijkstra's algorithm. Floyd-Warshall's algorithm: main idea, limitations, pseudo-code, correctness, and considerations about memory usage. Paths reconstruction.

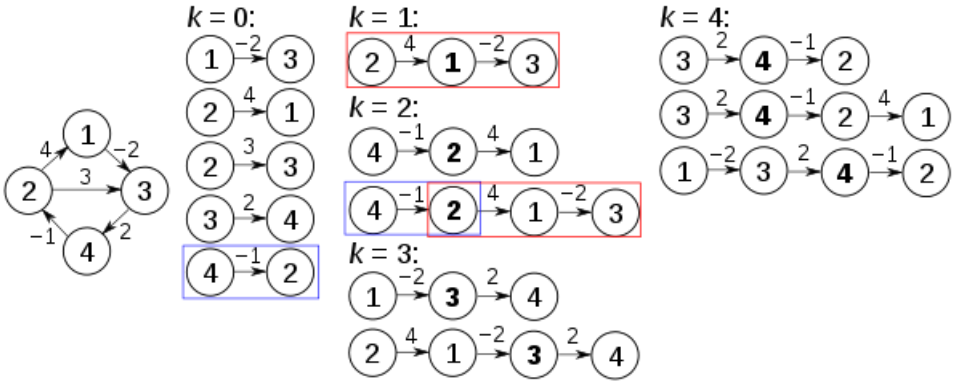
It solves **All pairs shortest path problem**: it finds the shortest paths in a weighted graph with positive or negative edge weights. A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices.

$D^{(0)}, ..., D^{|V|}$ are path distances

$$d_{ij}^{k} = \begin{cases} w_{ij} & if\ k = 0 \\ min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \end{cases}$$

```
Floyd_Warshall(W)
  D^(0) <- W
  for k in 1,..,|V|
  D^(k) <- zeros(|V|,|V|)
    for i in |V|
      for j in |V|
        D^(k)[i][j] <- min(D^(k-1)[i][j]+D^(k-1)[i][k]+D^(k-1)[k][j])
  return D^(k)
```

the complexity is $\Theta(|V|^3)$



Floyd-Warshall example.svg

---

16 aprile

continuazione Floyd-Warshall

Routing problem and Contraction Hierarchies

→ *contraction_hierarchy.pdf*
Definition of the routing problem. Issues and opportunities on large graphs. The contraction hierarchies: motivations, basic idea, contraction algorithm, ordering heuristic, queries. Examples.

The method of **contraction hierarchies** is a technique to speed up shortest-path routing by first creating precomputed "contracted" versions of the connection graph.

Contraction hierarchy non va implementato!

---

23 aprile

continuazione contraction hierarchy

Pattern matching

→ *p.923-931 , 985 Cormen*
The pattern matching problem: definition. Naive algorithm: idea, pseudo-code, complexity, and examples.
The Knuth-Morris-Pratt's algorithm: the prefix(-suffix) function and its usage, pseudo-code, complexity, and working example.

Naive solution
Σ alphabet
Σ^* set of finite sequences of elements in Σ
$x \in \Sigma^*$ is a string
$|x|$ is the length of $x \in \Sigma^*$
if $x, y \in \Sigma^*$, $xy \in \Sigma^*$ is the concatenation of x and y

```
PM_Naive(T, P)
  for s in [1, |T|-|P|] do
    i <- 0
    while i < |P| and P[i] == T[s+1] do
      i <- i+1
    if i == |P| then
      yield s
```

$\Theta(|T| \cdot |P|)$

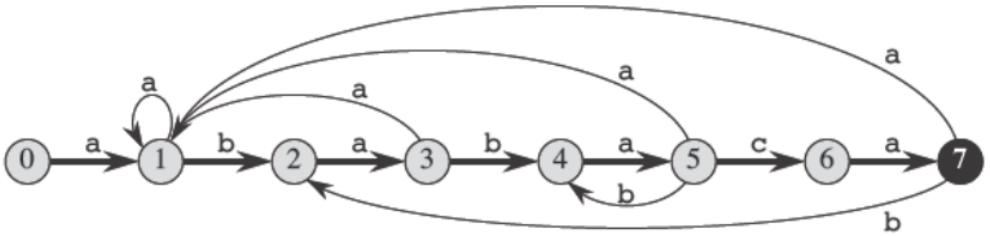
---

24 aprile

Knuth-Morris-Pratt

→ *pag.1002 Cormen*
→ *knut_morris_pratt.pdf*

string matching automata
$\delta : Q \times \Sigma \longrightarrow Q$
$\delta(q, a) = \sigma(P_q a)$



(a)

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i		1	2	3	4	5	6	7	8	9	10	11
$T[i]$		a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

$\delta(i, T[i])$:
$\delta(0, a) = \sigma(P_0 a) = \sigma(a) = 1$
$\delta(1, b) = \sigma(P_1 b) = \sigma(ab) = 2$
$\delta(2, a) = \sigma(P_2 a) = \sigma(aba) = 3$
$\delta(3, a) = \sigma(P_3 a) = \sigma(abaa) = 1$
...

```
FiniteAutomationMatcher(T,δ,m)
  n = length(T)
  q = 0
  for i = 1 to n
    q = δ(q, T[i])
    if q == m
      pattern P occurs with shift i-m
```

its matching time on a text string of length n is $\Theta(n)$. This matching time, however, does not include the preprocessing time required to compute the transition function δ.

Transition function
$\sigma : \Sigma^* \to Q$
$\sigma(x) = max\{k : P_k \sqsupset x\}$

```
ComputeTransitionFunction(P,Σ)
  m = length(P)
  for q = 0 to m
    for a ∈ Σ
    k = min(m+1, q+2)
    repeat
      k = k-1
    until P_k ⊐ P_q a
```

```
    δ(q, a) = k
  return δ
```

its running time is $O(m^3|\Sigma|)$

Knuth Morris Pratt

This algorithm avoids computing the transition function and its matching time is $\Theta(n)$, using just an auxiliary function π which we precompute from the pattern in time $\Theta(n)$ and store in an array $\pi[1, ..., m]$

$O(|T| + |P|)$

---

26 aprile + 3 maggio

Boyer Moore Galil

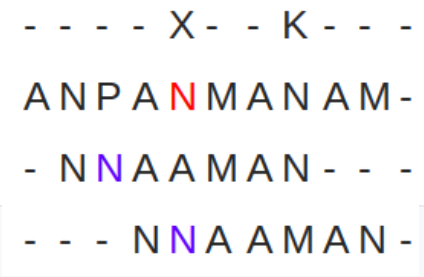
The Boyer-Moore algorithm searches for occurrences of pattern *P* in text string *T* by performing explicit character comparisons at different alignments. Instead of a brute-force search of all alignments (of which there are $m - n + 1$), Boyer-Moore uses information gained by preprocessing *P* to skip as many alignments as possible.

Right to left matching

Characters in *P* and *T* are then compared starting at index *n* in *P* and *k* in *T*, moving backward. The strings are matched from the end of *P* to the start of *P*. The comparisons continue until either the beginning of *P* is reached (which means there is a match) or a mismatch occurs and one of the shift rules is applied.

Bad character rule

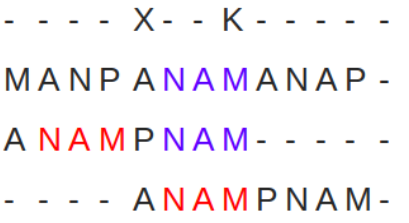
The bad-character rule considers the character in *T* at which the comparison process failed (assuming such a failure occurred). The next occurrence of that character to the left in *P* is found, and a shift which brings that occurrence in line with the mismatched occurrence in *T* is proposed. If the mismatched character does not occur to the left in *P*, a shift is proposed that moves the entirety of *P* past the point of mismatch.




```
def compute BCH(P)
  for a in Σ do
    BCH(a) <- 0
  for i in [1,...,|P|] do
    BCH(P[i]) <- i
  return BCH
```

Good suffix rule

Suppose for a given alignment of ***P*** and ***T***, a substring ***t*** of ***T*** matches a suffix of ***P***, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy ***t'***of ***t*** in ***P*** such that ***t'*** is not a suffix of ***P*** and the character to the left of ***t'*** in ***P*** differs from the character to the left of ***t*** in ***P***. Shift ***P*** to the right so that substring ***t'*** in ***P*** aligns with substring ***t*** in ***T***.
If ***t'*** does not exist, then shift the left end of ***P*** past the left end of ***t*** in ***T*** by the least amount so that a prefix of the shifted pattern matches a suffix of ***t*** in ***T***. If no such shift is possible, then shift ***P*** by ***n*** places to the right. If an occurrence of ***P*** is found, then shift ***P*** by the least amount so that a *proper* prefix of the shifted ***P*** matches a suffix of the occurrence of ***P*** in ***T***. If no such shift is possible, then shift ***P*** by ***n*** places, that is, shift ***P*** past ***t***.



Galil rule

The Galil rule is about exploiting periodicity in the pattern to reduce comparisons.
→ https://stackoverflow.com/questions/38206841/boyer-moore-galil-rule

```
def compute H(P,N)
  k <- 0
  for j in [1,...,|P|-1]
```

```
    if N(j) == j then
      k <- j
    H(|P|-j+1) <- k
  return H
```

N(i) is the longest suffix of P[1,...,i] that is a suffix for P i.e. P[i-j+1...i]=P[|P|-j+1...|P|]
Z(i) s the length of the longest prefix of S[i...|S|] that is also a prefix for S

```
def compute Z(P)
  DZ[1],Z[2] compute explicitely
  j<-2 , i<-3
  while i <= |S| do
    if i >= j+Z(j) then
      evaluate Z(i) explicitely
    else
      i' <- i-j+1
      Z(i) <- Z(i')
      if Z(i) >= Z(j) - (i-j) then
        extend the tail of Z(i)
    if j + Z(j) < i +Z(i) then
      j <- i
    i <- i+1
  return Z
```

The complexity is sublinear in |T|: $O(|T| + \sum |P_i|)$

---

7 maggio

Longest common subsequence problem

→ p.390 Cormen

Notice that substrings do not have to be contiguous.

Naive solution
The complexity is $O(2^{|X|}|Y|)$ where $2^{|X|}$ is the number of substrings and $|Y|$ for the linear scanning of Y.

Recursive solution → p.393

---

8 maggio

DNA
String matching with finite automata

Suffix trie

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/tries_and_suffix_tries.pdf
https://www.hackerearth.com/practice/notes/trie-suffix-tree-suffix-array/
https://www.geeksforgeeks.org/pattern-searching-using-trie-suffixes/

```
TOP <- t_1...t_{i-1}
R̄ ← TOP
while R̄ ≠⊥ do
  if g(R̄, t_i) is not defined
    create Rt_i
```

```
    g(R̄, t_i) ← Rt_i
    if OLDR is defined
      f(OLDR) <- Rt_i
    OLDR <- Rt_i
R̄ ← f(R̄)
TOP <- g(TOP,t_i)
```

```
def BuildTrie(T)
  create ⊥
  for t ∈ Σ
    g(⊥, t) ← ε̄
  f(ε̄) ←⊥
  TOP ← ε̄
  for i ∈ [1, ..., |T|] do
    OLDR <- None
    R̄ ← TOP
    while g(R̄, t_i) is None do
      create R'
      g(R̄, t_i) ← R'
      if OLDR is not None
        f(OLDR) ← R'
      OLDR ← R'
      R̄ ← f(R̄)
    if OLDR' is not None
      f(OLDR') <- g(R̄, t_i)
    TOP ← g(TOP, t_i)
```

- The `BuildTrie` procedure is optimal, i.e. it is $\Theta(|Q|)$ where Q is the set of nodes in $STrie(T)$
- $|Q| \in O(|T|^2)$

10-14 maggio (perse)

suffix trie pseudocodice

- [ ] chiedere gli appunti
- [ ] suffix tree https://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/

---

15 maggio

Suffix array

pattern matching on a suffix array takes $O(|P|log|T|)$, where P is the pattern and T is the string

```
def ≤l(T_1, T_2, l)
  S_1 ← |T_1|
  S_2 ← |T_2|
  k <- 1
  while k≤ S_1 and k ≤ S_2 and k ≤ l do
    if T_1[k] > T_2[k] then
      return false
    else if T_1[k] < T_2[k] then
```

```
      return true
    k <- k+1
  return k = S_1+1 or k = l+1
```

```
def suffixArrayMatching(T, SA, P)
  ϕ ← |P|
  if ≤ l(P, T_SA[1], ϕ) then
    L_P ← 0
  else if ≤ l(T_SA[|T|], P, ϕ) then
    L_P ← |T| + 1
  else
    (L, R) ← (1, |T|)
    while R-L > 1 do
      M <- (R+L)/2
      if ≤ l(P, T_SA[M], ϕ) then
        R <- M
      else
        L <- M
  L_P ← R
  compute R_P
  return (L_P, R_P)
```

In order to improve complexity we may try to store the longest common prefixes among (P,L) and (P,R)

- [ ] studiare questo codice https://www.geeksforgeeks.org/suffix-array-set-1-introduction/

`txt+i` è semplicemente un puntatore all'i-esimo elemento dell'array txt, e mi permette di stampare il testo solo da un certo punto in poi

LCP array

the algorithm returns the LCP between T_1, T_2 and starts the comparison from char s

```
def LCP (T_1, i_1, T_2, i_2, s = 1)
  while T_1[i_1 + s] = T_2[i_2 + s] and |T_1| ≥ i_1 + s and |T_2| ≥ i_2 + s
    s <- s+1
  return s-1
```

pattern matching becomes:

```
def suffixArrayMatching(T, SA, P)
  ϕ ← |P|
  if ≤ l(P, T_SA[1], ϕ) then
    L_P ← 0
  else if ≤ l(T_SA[|T|], P, ϕ) then
    L_P ← |T| + 1
  else
    (L, R) ← (1, |T|)
    l <- LCP(T, SA[L], P, 0)
    r <- LCP(T, SA[R], P, 0)
    while R-L > 1 do
      M <- (R+L)/2
```

```
      h <- min(r,l)
      if ≤ l(P, T_SA[M], start = h, upto = φ) then
        R <- M
        r <- LCP(T, SA[M], P, 0, h)
      else
        L <- M
        l <- LCP(T, SA[M], P, 0, h)
  L_P <- L
  compute R_P
  return (L_P, R_P)
```

This does not change the complexity of the algorithm

---

17 maggio

```
def LCP(T, i, T̄, t)
  k <- 0
  while i + k ≤ |T| && ī + k ≤ |T̄| && T[k + i] = T̄[k + t] do
    k <- k+1
  return k-1
```

- [ ] studiare questo codice (copiato in locale): https://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/
- [ ] scrivere una seconda versione adattando lo pseudocodice del prof e senza std library

`?` in c++ è il **conditional operator**: https://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/

suffix array

... (forse ho perso l'inizio)

```
(L,R) ← (1, |T|)
while R > L+1 do
  M <- (R+L)/2
  if l>=r then
    if L_LCP[M] ≥ l then
      m ← l + LCP(T, SA[M] + l, P, l)
    else
      m ← L_LCP[M]
  else
    if R_LCP[M] ≥ r then
      m <- r + LCP(T, SA[M]+r, P, r)
    else
      m ← R_LCP[M]
  if m == P || P[m] <= T[SA[M]+m] then
    (R,m) <- (M,m)
  else
    (L, l) <- (M, m)
  return L
```

---

21 maggio

```
def UpdatePhase(T, SA, Prm, Bh, H)
  // initialization
  for each bucket (l,r) do
    count[l] <- 0
    for c ∈ [l,r] do
      Prm[SA[c]] <- l
  for each bucket (l,r) do
    for d ∈ [l,r] do
      count[Prm[d]] <- count[Prm[d]]
      Prm[d] <- Prm[d] + count[Prm[d]]
      Bh2[Prm[d]]<- true
    for d ∈ [l,r] do
      if Bh2[Prm[d]] then
        l <- min(J:J>Prm[d],Bh[J] or not Bh2[J])
        for f ∈ [Prm[d]+1,l-1]
          Bh2[f] <- false
    for i in [1,|T|] do
      SA[Prm[i]] <- i
      if Bh2[i] then
        Bh[i] <- true
  return (SA, Prm, Bh)
```

This takes time $\Theta(|T|)$ and after $log_2|T|$ applications of the procedure we get the correct suffix array, so building the suffix array takes time $\Theta(|T|log|T|)$.

---

24 maggio
persa

---

28 maggio

Red black tree
→ *p.308 Cormen*

To delete a node with value v:
1. search for v in the tree (let M be the corresponding node);
2. Search for the immediate successor of v: i.e., search for the leftmost node (C) of the right child of M;
3. M.value ← C.value;
4. replace C with its right son, N;
5. if C and N, then color N black;
6. if N is not the root, then N has a parent, and there are some cases where a recoloring is needed (search for figures on textbooks);

```
def RB-delete(T,v)
  z <- RB-search(T,v)
  if z == none then
    return
  if z.right == leaf then
    y <- z
  else
    y <- searchSuccessor(T,z)
```

```
  y.rigth.parent <- y parent
  if y.parent == none then     # y was root
    T.root == y.right
  else
    if is_left_son(y) then
      y.parent.left <- y.right
    else
      y.parent.right <- y.right
    z_value <- y_value
  if y.right.color == red or y.color == red then
    y.right.color <- black
    delete y
    return
  RBT_delete_fixup(T,y.right)
  delete y
```

```
def RBT_delete_fixup(T,N)
  while(N ≠ T.root and N.color == black) do
    if is_left_son(N) then
      S <- N.parent.right
      if S.color == red then
        S.color == black
        N.parent.color <- red
        left_rotation(T, N.parent)
        S <- N.parent.right
      else # sibling is black
        if S.right.color == black and S.left.color == black then
          S.color <- red
          N <- N.parent
        else
          if S.left.color == red then
            S.left.color <- black
            S.color <- red
            right_rotation(T,S)
            S <- N.parent.right
          S.color <- N.parent.color
          N.parent.color <- black
          left_rotation(T, N.parent)
          N <- S
    else   # N is the right son
    # exactly as before but sides are switched
  N.color <- black
```

- [ ] studiare questo codice e implementarlo: c red black tree insertion

---

Esame

su appuntamento, preferibilmente in gruppo
two parts:

- homework, implement algorithms in c++

    - per l'esame bisogna implementare tutti gli algoritmi fino a quelli sui grafi, e poi solo un altro a scelta di quelli dopo i grafi (tutti i codici dell'argomento)
- oral presentation about a research paper
    - proporlo prima al prof per email (circa 10 lucidi, 15/20 minuti)