

Appunti di Algoritmi

Parte sui Grafi

Definizioni

Grafo connesso: se G e' un grafo non orientato, definiamo G connesso se esiste un cammino da ogni vertice ad ogni altro vertice.

Grafo fortemente connesso: se G e' un grafo orientato, definiamo G fortemente connesso se esiste un cammino da ogni vertice ad ogni altro vertice.

Grafo debolmente connesso: se G e' un grafo orientato, definiamo G debolmente connesso se il grafo ottenuto da G dimenticando la direzione degli archi e' connesso.

Ciclo: un ciclo e' un cammino x_1, \dots, x_n , di lunghezza almeno 1, tale che $x_1 = x_n$.

Grafo aciclico: un grafo senza cicli e' detto aciclico.

Grafo completo: un grafo completo e' un grafo che ha un arco tra ogni coppia di vertici.

Albero libero: un albero libero e' un grafo non orientato, connesso e aciclico.

Albero radicato: un albero radicato e' un grafo non orientato, connesso e aciclico, con un vertice designato ad essere radice.

Foresta: una foresta e' un grafo non orientato, aciclico ma non necessariamente connesso.

BFS (visita in ampiezza)

```
VISITA( $G, s$ )
   $D \leftarrow \text{make\_empty}$ 
   $\text{color}[s] \leftarrow \text{grigio}$ 
   $\text{add}(D, s)$ 
  while  $\text{non\_empty}(D)$  do
     $u \leftarrow \text{first}(D)$ 
    for  $\forall v : v \text{ e' bianco ed } e' \in \text{adj}[u]$  do
       $\text{color}[v] \leftarrow \text{grigio}$ 
       $\pi[v] \leftarrow u$ 
       $\text{add}(D, v)$ 
     $\text{color}[u] \leftarrow \text{nero}$ 
     $\text{remove\_first}(D)$ 
```

DFS (visita in profondita') – versione iterativa

```
VISITA(G, s)
  D ← make_empty
  color[s] ← grigio
  add(D, s)
  while non_empty(D) do
    u ← first(D)
    while ∃v bianco ∈ adj[first[u]] do
      color[v] ← grigio
      π[v] ← u
      add(D, v)
    color[first[u]] ← nero
    remove_first(D)
```

DFS (visita in profondita') – versione ricorsiva

```
VISITA(G, s)
  color[s] ← grigio
  for ∀v : v e' bianco ed e' ∈ adj[s] do
    π[v] ← u
    VISITA(G, v)
  color[s] ← nero
```

DFS (visita in profondita') – ordinamento topologico

```
DFS_TOPOLOGICAL(G, s, Lista)
  color[s] ← grigio
  d[s] ← time ← time + 1
  for ∀v : v e' bianco ed e' ∈ adj[s] do
    π[v] ← s
    DFS_TOPOLOGICAL(G, v, Lista)
  color[s] ← nero
  d[s] ← time ← time + 1
  in testa a Lista inserisci s
```

Proprieta'

una condizione necessaria e sufficiente per l'esistenza di un ordinamento topologico e' l'aciclicita'.

DFS (visita in profondita') – raggiungibilita' di un nodo

```
RAGGIUNGIBILI(G, s)
  for  $\forall u \in V$  do
    R[u]  $\leftarrow$  false
    color[u]  $\leftarrow$  bianco
  RAGGIUNGIBILI-Ric(G, s)

RAGGIUNGIBILI-Ric(G, u)
  color[u]  $\leftarrow$  grigio
  R[u]  $\leftarrow$  true
  for  $\forall v \in \text{adj}[u]$  do
    if color[v] e' bianco then
      RAGGIUNGIBILI-Ric(G, v)
  color[u]  $\leftarrow$  nero
```

Dijkstra (algoritmo Greedy)

```
MST_DIJKSTRA(G, s)
  Q  $\leftarrow$  V
  for  $\forall v \in V$  do
    d[v]  $\leftarrow$   $\infty$ 
  d[s]  $\leftarrow$  0
   $\pi[s] \leftarrow$  nil
  while Q  $\neq$   $\emptyset$  do
    u  $\leftarrow$  nodo con d minimo in Q
    for  $\forall v \in \text{adj}[u]$  do
      if  $v \in Q$  e  $d[u] + W(u, v) < d[v]$  then
        d[v]  $\leftarrow$   $d[u] + W(u, v)$ 
         $\pi[v] \leftarrow$  u
```

Prim (algoritmo Greedy)

```
MST_PRIM(G, s)
  Q  $\leftarrow$  V
  for  $\forall v \in V$  do
    d[v]  $\leftarrow$   $\infty$ 
  d[s]  $\leftarrow$  0
   $\pi[s] \leftarrow$  nil
  while Q  $\neq$   $\emptyset$  do
    u  $\leftarrow$  nodo con d minimo in Q
    for  $\forall v \in \text{adj}[u]$  do
      if  $v \in Q$  e  $W(u, v) < d[v]$  then
        d[v]  $\leftarrow$   $W(u, v)$ 
         $\pi[v] \leftarrow$  u
```

L'algoritmo di Prim procede secondo lo schema in cui l'appetibilita' dei nodi cambia (cioe' la distanza dei nodi gia' aggiunti alla soluzione).

Kruskal (algoritmo Greedy)

```
MST_KRUSKAL(G)
  A ← ∅
  ordina gli archi in ordine non decrescente di peso
  for ∀(u, v) nell'ordine do
    if (u, v) e' "sicuro" per A then
      A ← A U (u, v)
```

L'algoritmo di Kruskal procede secondo lo schema in cui l'appetibilita' degli archi non cambia (il peso e' sempre lo stesso).

Kruskal (algoritmo Greedy) – grafo connesso senza cicli

```
MST_KRUSKAL(G)
  A ← ∅
  ordina gli archi in ordine non decrescente di peso
  for ∀(u, v) nell'ordine do
    if (u, v) non crea ciclo in G(A) = (V, A) then
      A ← A U (u, v)
```

Kruskal (algoritmo Greedy) – con insiemi disgiunti (Union/Find)

```
MST_KRUSKAL(G)
  A ← ∅
  for ∀v ∈ V do
    Make_set(v)
  ordina gli archi in ordine non decrescente di peso
  for ∀(u, v) ∈ E nell'ordine do
    if Find(u) ≠ Find(v) then
      A ← A U (u, v)
      Union(u, v)
```

Vicinanza di un nodo

Dato un albero orientato, si definisce "vicinanza di un nodo v" la minima distanza di v da una foglia.

```
CALCOLAVic(G, r)
  for ∀u ∈ V do
    vic[u] ← ∞
  CALCOLAVic-Ric(G, r)
```

```
CALCOLAVic-Ric(G, u)
  if adj[u] = ∅ then
    vic[u] ← 0
  else
    for ∀v ∈ adj[u] do
      CALCOLAVic-Ric(G, v)
      vic[u] ← min(vic[u], vic[v])
  vic[u] ← vic[u] + 1
```

Parte di Algoritmi

Calcolo della complessita'

Significato dei simboli:

- $O(n)$ → limite inferiore della funzione
- $\Omega(n)$ → limite superiore della funzione
- $\Theta(n)$ → andamento esatto della funzione

Un algoritmo di ordinamento si dice stabile se non altera l'ordine relativo di oggetti distinti che siano uguali rispetto alla relazione d'ordine, come ad esempio l'ordine relativo di elementi con chiavi uguali.

Se una sequenza di elementi gia' ordinata secondo un certo criterio viene ordinata secondo un altro criterio per mezzo di un algoritmo stabile, elementi uguali secondo il nuovo criterio risultano ordinati secondo il precedente criterio (es: se nella sequenza da ordinare vi sono due elementi *A* e *B* di uguali chiavi, e *A* si trova prima di *B* in tale sequenza, anche nella sequenza ordinata *A* deve trovarsi prima di *B*).

Un algoritmo di ordinamento su array si dice sul posto (o in loco, o in place) se in ogni istante al piu' un numero costante di elementi viene memorizzato al di fuori dell'array, cioe' se l'algoritmo riordina l'array senza usare un array ausiliario o comunque uno spazio proporzionale al numero di elementi.

Un algoritmo di ordinamento si dice sul posto se la sua complessita' spaziale e' al piu' **$O(\log n)$** .

Complessita' degli algoritmi di ordinamento:

	<u>Spazio</u>	<u>Tempo</u>
Selection sort	$O(1)$	$O(n^2)$
Insertion sort	$O(1)$	$O(n^2)$ casi peggiore e medio, $O(n)$ caso migliore
Merge sort	$O(n)$	$O(n \log n)$ casi peggiore e medio
Quicksort	$O(\log n)$	$O(n^2)$ caso peggiore, $O(n \log n)$ casi medio e migliore
Heapsort	$\Theta(1)$	$O(n \log n)$
Integer sort	$O(n+k)$	$\Theta(n+k)$
Counting sort	$O(n+k)$	$\Theta(n+k)$

Classificazione degli algoritmi di ordinamento:

Selection sort	<u>non</u> stabile	<i>sul posto</i>
Insertion sort	stabile	<i>sul posto</i>
Merge sort	stabile	<u>non</u> <i>sul posto</i>
Quicksort	<u>non</u> stabile	<i>sul posto</i>
Heapsort	<u>non</u> stabile	<i>sul posto</i>

Teorema master delle equazioni di ricorrenza

$$T(1) = d$$

$$T(n) = aT(n/b) + cn^s$$

dove:

$d \geq 0$ → il lavoro sul problema di dimensione 1 puo' essere nullo

$a \geq 1$ → e' il numero delle chiamate ricorsive

$b > 1$ → sottoproblemi che devono essere piu' piccoli del problema

$c > 0$ → test per stabilire se si e' nel caso base (e' un tempo aggiuntivo)

$s \geq 0$ → tempo per la divisione e la ricombinazione delle chiamate ricorsive

se:

$$a < b^s \rightarrow T(n) = O(n^s)$$

$$a = b^s \rightarrow T(n) = O(n^s \log n)$$

$$a > b^s \rightarrow T(n) = O(n^{\log_b a})$$

Quicksort (generico)

```
public static void quicksort(int[] a, int inf, int sup)
{
    if (inf < sup)
    {
        int pivot = a[inf];
        int i = inf;
        int j = sup;

        while (i < j)
        {
            if (pivot < a[i+1])
            {
                swap(a, i+1, j);
                j--;
            }
            else
            {
                swap(a, i, i+1);
                i++;
            }
        }

        quicksort(a, inf, i-1);
        quicksort(a, i+1, sup);
    }
}
```

Quicksort (con tripartizione)

```
public static void quicksort(int[] a, int inf, int sup)
{
    if (inf<sup)
    {
        int x = inf;
        int i = inf;
        int j = sup;

        while (i<j)
        {
            if (a[i]<a[i+1])
            {
                swap(a, i+1, j);
                j--;
            }
            else if (a[i]>a[i+1])
            {
                swap(a, x, i+1);
                x++;
                i++;
            }
            else
            {
                i++;
            }
        }

        quicksort(a, inf, x-1);
        quicksort(a, i+1, sup);
    }
}
```

Quicksort (con ricerca del mediano)

```
public static int mediano(int[] a)
{
    int len = a.length;

    if (len>1)
    {
        int inf = 0;
        int sup = len - 1;
        int med = (inf+sup) / 2;

        return qsortMed(a, inf, med, sup);
    }
    else if (len==1)
        return a[0];
    else
        return -1;
}

public static int qsortMed(int[] a, int inf, int med, int sup)
{
    if (inf==med && med==(sup+1))
```

```

        return a[med];
    else
    {
        int pivot = a[inf];
        int i = inf;
        int j = sup;

        while (i<j)
        {
            if (pivot<a[i+1])
            {
                swap(a, i+1, j);
                j--;
            }
            else
            {
                swap(a, i, i+1);
                i++;
            }
        }

        if (i<med)
            return qsortMed(a, i+1, med, sup);
        else
            return qsortMed(a, inf, med, i-1);
    }
}

```

Segmento di somma massima

```

public void segmento(int[] a)
{
    int sum = a[0];
    int haltSum, tmp, left, right;

    for (int k=0; k<a.length; k++)
    {
        haltSum = Math.max(haltSum+a[k], 0);

        if (haltSum==0)
            tmp = k + 1;

        sum = Math.max(sum, haltSum);

        if (sum==haltSum)
        {
            left = tmp;
            right = k;
        }
    }

    System.out.println("Somma: "+sum+" (indici: "+left+" "+right+").");
}

```

Integer sort

a e' un array di n interi compresi nel range **0...k-1**, con possibili ripetizioni.

```
public static void intSort(int k, int[] a)
{
    int[] counters = new int[k];
    int n = a.length;

    for (int i=0; i<k; i++)
        counters[i] = 0;

    for (int j=0; j<n; j++)
        counters[a[j]]++;

    int x = 0;

    for (int v=0; v<k; v++)
    {
        while (counters[v]>0)
        {
            a[x] = v;
            x++;
            counters[v]--;
        }
    }
}
```

Counting sort

Versione normale e stabile:

```
public static void csort(int k, ItemWithIntKey[] a)
{
    int n = a.length;
    ItemWithIntKey[] b = new ItemWithIntKey[n];
    int[] c = new int[k];

    for (int i=0; i<n; i++)
        c[a[i].key()]++;

    int totale = 0;

    for (int j=0; j<k; j++)
    {
        int temp = c[j];
        c[j] = totale;
        totale += temp;
    }

    for (int i=0; i<n; i++)
    {
        b[c[a[i].key()]] = a[i];
        c[a[i].key()]++;
    }
}
```

Versione **scrausa, al contrario e "antistabile"**:

```
public static void csort(int k, ItemWithIntKey[] a)
{
    int n = a.length;
    ItemWithIntKey[] b = new ItemWithIntKey[n];
    int[] c = new int[k];

    for (int i=0; i<n; i++)
        c[a[i].key()]++;

    int totale = 0;

    for (int j=0; j<k; j++)
    {
        int temp = c[j];
        c[j] = totale;
        totale += temp;
    }

    for (int i=0; i<n; i++)
    {
        b[c[a[i].key()]] = a[i];
        c[a[i].key()]++;
    }
}
```

Coda con priorità

Una coda con priorità non obbedisce ad una disciplina FIFO, in quanto da essa si estrae ogni volta l'elemento con priorità più alta. È consuetudine che la priorità più alta possibile sia indicata dal numero 1, e le priorità inferiori da numeri interi più grandi (l'estrazione sarà quindi sempre sul minimo).

Possibili realizzazioni:

Liste ordinate (in ordine crescente): l'elemento con priorità più alta, cioè il minimo, è il primo, ma l'inserimento è un inserimento al posto giusto in lista ordinata.

Quindi:

- **Estrazione del minimo:** complessità $\Theta(1)$
- **Inserimento (casi medio e peggiore):** complessità $\Theta(n)$

Liste non ordinate: l'inserimento può avvenire sempre in testa, ma l'estrazione è una ricerca del minimo in sequenza non ordinata.

Quindi:

- **Estrazione del minimo:** complessità $\Theta(n)$
- **Inserimento:** complessità $\Theta(1)$

Struttura-dati Heap

Una struttura-dati Heap è un **albero d-ario** con le seguenti proprietà:

- la chiave di ciascun nodo è minore o uguale alla chiave di ogni suo figlio non nullo
- l'albero è quasi-completo da sinistra (cioè binario quasi-completo)

L'albero (Heap) viene realizzato per mezzo di un array in cui i nodi sono memorizzati consecutivamente per livelli. Se i nodi vengono memorizzati nell'array a partire dall'indice 1 (lasciando inutilizzato l'elemento di indice 0), i figli del nodo memorizzato nell'elemento di indice i risultano memorizzati rispettivamente negli elementi di indice $2i$ e $2i+1$:

- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i+1$
- $\text{parent}(i) = \lfloor i/2 \rfloor$

Heapsort

```
public static void heapsort(int[] a)
{
    heapify(a, 0);

    int j = a.length - 1;

    while (j>0)
        a[j] = extractMax(a, j--);
}

public static void heapify(int[] a, int i)
{
    int lastIndex = a.length - 1;
    int j = 2*i + 1;

    if (j<=lastIndex)
    {
        heapify(a, j);
        heapify(a, j+1);

        fixHeap(a, i, lastIndex);
    }
}

public static void fixHeap(int[] heap, int i, int lastIndex)
{
    int node = heap[i];
    int j;

    while ((j=2*i+1)<=lastIndex)
    {
        if ((j+1)<=lastIndex && heap[j+1]>heap[j])
            j++;

        if (node<heap[j])
        {
            heap[i] = heap[j];
            i = j;
        }
        else
            break;
    }

    heap[i] = node;
}

public static int extractMax(int[] heap, int lastIndex)
{

```

```

int max = heap[0];
heap[0] = heap[lastIndex];

lastIndex--;

fixHeap(heap, 0, lastIndex);

return max;
}

```

Tavole ad accesso diretto (hash tables)

Fattore di carico:

e' il grado di riempimento di una tavola, e si indica con $\alpha=(n/m)$, dove n e' il numero di elementi da indicizzare e m e' il numero di celle disponibili a contenere il codice di hash.

Tutte le operazioni richiedono tempo **$O(1)$** .

Per ridurre le probabilita' di collisioni, una buona funzione hash dovrebbe essere in grado di distribuire in modo uniforme le chiavi nello spazio. Questo si puo' ottenere con la proprieta' dell'uniformita' semplice: **$Q(i)=(1/m)$**

Risoluzione delle collisioni:

- **Liste di collisione:** gli elementi sono contenuti in liste esterne alla tabella: $v[i]$ punta alla lista degli elementi tali che $h(k)=i$.
- **Indirizzamento aperto:** tutti gli elementi sono contenuti nella tabella: se una cella e' occupata, se ne cerca un'altra libera. Supponiamo, ad esempio, di voler inserire un elemento con chiave k e la sua posizione "naturale" $h(k)$ sia gia' occupata. L'indirizzamento aperto consiste nell'occupare un'altra cella, anche se potrebbe spettare di diritto a un'altra chiave. Cerchiamo la cella vuota (se c'e') scendendo le celle secondo una sequenza di indici: **$c(k,0), c(k,1), c(k,2), \dots, c(k,m-1)$**

Metodi di scansione (relativi all'indirizzamento aperto):

- **Scansione lineare:** $c(k,i) = (h(k)+i) \bmod m$ per $0 \leq i < m$
NB: la scansione lineare provoca effetti di agglomerazione, cioe' lunghi gruppi di celle consecutive occupate che rallentano la scansione.
- **Hashing doppio:** $c(k,i) = (h_1(k)+i \cdot h_2(k)) \bmod m$ per $0 \leq i < m$, con h_1 e h_2 funzioni hash

Alberi di ricerca binari (BST)

Costruttore per sottoalbero vuoto rappresentato da **null**:

```

public class Node
{
    E element;
    Node left, right;

    Node(E elem)
    {

```

```

        element = elem;
        left, right = null;
    }
}

```

Costruttore per sottoalbero vuoto rappresentato da **oggetti vuoti**:

```

public class Node
{
    E element;
    Node left, right;

    Node()
    {
        element, left, right = null;
    }

    void setTo(Node nd)
    {
        element = nd.element;
        left = nd.left;
        right = nd.right;
    }
}

```

Alberi AVL

Fattore di bilanciamento di un nodo **v**:

altezza del sottoalbero sinistro di **v** - altezza del sottoalbero destro di **v**

Un albero si dice bilanciato in altezza se ogni nodo **v** ha fattore di bilanciamento ≤ 1 .

Un albero AVL con **n** nodi ha altezza **O(logn)**.

Il ribilanciamento puo' avvenire tramite 4 rotazioni:

Sinistra-Sinistra (SS)	<i>T e' il sottoalbero sinistro del figlio sinistro di v</i>
Destra-Destra (DD)	<i>T e' il sottoalbero destro del figlio destro di v</i>
Sinistra-Destra (SD)	<i>T e' il sottoalbero destro del figlio sinistro di v</i>
Destra-Sinistra (DS)	<i>T e' il sottoalbero sinistro del figlio destro di v</i>