Collezione di Algoritmi

Luca Tagliavini

25 febbraio 2022

Algoritmi

Algoritmo 1 (Depth-First Search)**.** *Provede una ricerca in profondità su alberi. Nel codice seguente viene mostrata la versione per alberi binari ma è facile adattarlo a alberi n-ari. Ne esitono tre varianti:* pre,in,post-*ordine.*

```
begin dfs(Tree t, fn visit)
    if t ≠ null then
        /* visit può essere chiamata prima, in mezzo o dopo le
           chiamate ricorsive per ottenere rispettivamente
           ricerche pre,in,post-ordine                        */
        visit(t)
        dfs(t.left, visit)
        dfs(t.right, visit)
    end
end
```

Algoritmo 2 (Breadth-First Search)**.** *La ricerca in ampiezza visita l'albero livello-per-livello. Per ottnere il funzionamento desiderato useremo una* ***Queue****.*

```
begin bfs(Tree t, fn visit)
    Queue queue ← t
    while queue is not empty do
        Tree t ← queue
        visit(t)
        if t.left ≠ null then
            queue ← t.left
        end
        if t.right ≠ null then
            queue ← t.right
        end
    end
end
```

Le operazioni ← *queue* e *queue* ← rappresentano rispettivamente l'estrazione e l'inseriemnto in coda.

Algoritmo 3 (Ricerca BST)**.** *Algoritmo utilizzato per effettuare una ricerca binaria su un Binary Search Tree, che rispetta dunque le seguenti proprieta':*

1. *Ogni nodo ha associato una* ***chiave*** *e un campo* ***data***
2. *Le chiavi a* ***sinistra*** *sono $\leq$ a quella del nodo padre*
3. *Le chiavi a* ***destra*** *sono $\geq$ a quella del nodo padre*

Se ne possono scrivere una versione ricorsiva o iterativa, di seguito riportate.

Versione ricorsiva*:*

```
begin search(BFS t, Key key) → BFS
    if t = null or t.key = key then
        return null
    else if t.key ≥ key then
        return search(t.left, key)
    else
        return search(t.right, key)
    end
end
```

Versione iterativa*:*

```
begin search(BFS t, Key key) → BFS
    while t ≠ null or t.key ≠ key do
        if t.key ≥ key then
            t ← t.right
        else
            t ← t.left
        end
    end
    return t
end
```

Algoritmo 4 (Ricerca Massimo BST)**.** *Ricerca del massimo valore chiave in un albero BST. Si ricordi che per le proprietà degli alberi di ricerca, il valore massimo sarà quello più a destra.*

```
begin max(BFS t) → BFS
    while t ≠ null and t.right ≠ null do
        t ← t.right
    end
    return t
end
```

Algoritmo 5 (Ricerca Minimo BST)**.** *Ricerca del minimo valore chiave in un albero BST. Si ricordi che per le proprietà degli alberi di ricerca, il valore massimo sarà quello più a sinistra.*

```
begin min(BFS t) → BFS
    while t ≠ null and t.left ≠ null do
        t ← t.left
    end
    return t
end
```

Algoritmo 6 (Ricerca Successore BST)**.** *Algoritmo che trova il successore di un dato nodo sorgente* v*, ossia il nodo con il più piccolo valore maggiore di* v*. Si hanno due casi:*

1. **v *ha nodo destro**: identifichiamo* successor(v) *come il* min(v)*.*
2. **v *non ha nodo destro**: identifichiamo* successor(v) *un nodo antenato di* v *tale che* v *sia contenuto nel sottoalbero sinistro di tale antenato.*

```
begin successor(BFS t) → BFS
    if t = null then
        return null
    else if t.right ≠ null then
        return min(t.right)
    else
        BFS parent ← t.parent
        /* ci si ferma quando si trova un antenato che ha il
           nodo d'input (o un sottoalbero contenente esso)
           come nodo di sinistra                              */
        while parent ≠ null and parent.left ≠ t do
            t ← p
            parent ← p.parent
        end
        return parent
    end
end
```

Algoritmo 7 (Ricerca Predecessore BST)**.** *Algoritmo per la ricerca del prede-*

cessore in un BST.

```
begin predecessor(BFS t) → BFS
    if t = null then
        return null
    else if t.left ≠ null then
        return max(t.left)
    else
        BFS parent ← t.parent
        /* ci si ferma quando si trova un antenato che ha il
           nodo d'input (o un sottoalbero contenente esso)
           come nodo di destra                                  */
        while parent ≠ null and parent.right ≠ t do
            t ← p
            parent ← p.parent
        end
        return parent
    end
end
```

Algoritmo 8 (Inserimento BST)**.** *Algoritmo per l'inserimento di un valore in un Binary Search Tree.*

```
begin insert(BFS t, BFS item) → BFS
    if t = null then
        return item
    else if t.key > key then
        t.left ← insert(t.left, item)
    else
        t.right ← insert(t.right, item)
    end
    return t
end
```

NOTA: il valore inserito *item* e' si di tipo **BFS**, tuttavia dev'essere un singolo nodo foglia, non puo' contenere altro che una chiave e un dato associato.

Algoritmo 9 (Selection Sort)**.** *Ordina un array scambiando ad ogni passo i l'elemento di minimo valore in $i..n$ con quello in posiione posizione i.*

```
begin selection_sort(T[1..n] v)
    for i ← 1 to n do
        /* come prima cosa trovo il minimo                    */
        m ← i
        for j ← m + 1 to n do
            if v[j] < v[m] then
                m ← j
            end
        end
        /* successivamente, se necessario, scambio i due
           elementi                                           */
        if m ≠ i then
            tmp ← v[m]
            v[m] ← v[i]
            v[i] ← tmp
        end
    end
end
```

Costo complessivo $\Theta(n^2)$.

Algoritmo 10 (Insertion Sort)**.** *Esegue n passi per ordinare l'array, garantendo che al passo $2 \leq k \leq n$ i primi $1..k$ elementi saranno ordinati. Ad ogni k-esimo passo sposta dunque l'elemento in posizione k nel luogo giusto all'interno di $1..k-1$.*

```
begin insertion_sort(T[1..n] v)
    for i ← 2 to n do
        /* troviamo la giusta posizione in cui piazzare v[i]
           tra 1..i − 1                                       */
        j ← 1
        for j to i-1 do
            if v[j] > v[i] then
                break
            end
        end
        /* spostiamo v[j..i − 1] in v[j + 1...i]              */
        for t ← i back to j do
            v[t] ← v[t − 1]
        end
        v[j] ← v[i]
    end
end
```

Costo complessivo $\Theta(n^2)$.

Algoritmo 11 (Bubble Sort)**.** *Esegue n cicli, ad ogni ciclo sposta n volte le coppie di elementi affinche' siano ordinate. In questo modo, dopo il primo ciclo avremo che l'ultimo elemento e' il massimo, dopo il secondo ciclo che il penultimo elemento e' il secondo massimo, e così via.*

```
begin bubble_sort(T[1..n] v)
    for i ← 1 to n do
        changed ← false
        for j ← 1 to n − i do
            /* se v[j] è maggiore del suo successore, scambiali
               */
            if v[j] > v[j + 1] then
                tmp ← v[j + 1]
                v[j + 1] ← v[j]
                v[j] ← tmp
                changed ← true
            end
        end
        if not changed then
            break
        end
    end
end
```

Costo complessivo nel caso pessimo $\Theta(n^2)$*, ma nel caso ottimo, in cui l'array è già ordinato, ha costo* $\Theta(n)$.

Algoritmo 12 (Quick Sort)**.** *Il QuickSort e' un algoritmo di sorting con un input leggermente divers. Prende infatti un array* $v[1..n]$ *e due indici* i *ed* f *tali che* $1 \leq i < f \leq n$*. Essneod il Quicksort un algoritmo* ***divide-et-impera****, si hanno due fasi:*

1. ***divide****: scegliamo un elemento dell'array chiamato* ***pivot****, tra* i *ed* f*, diciamo in posizione* m*, e dividiamo l'array in due sottoarray* $v[i..m-1]$ *e* $v[m+1..f]$ *tali per cui:*

$$\forall j \in \{i, \ldots, m-1\}.\ v[j] \leq v[m]$$
$$\forall k \in \{m+1, \ldots, f\}.\ v[k] > v[m]$$

2. ***impera****: ordina i due sottoarray chiamando ricorsivamente l'algoritmo QuickSort.*

```
begin partition(T[1..n] v, int i, int f)
    /* scelta deterministica del pivot x                    */
    x ← v[i]
    inf ← i + 1
    sup ← f
    while true do
        while inf ≤ f and v[inf] ≤ x do
            inf ← inf + 1
        end
        while v[sup] > x do
            sup ← sup − 1
        end
        /* a questo punto abbiamo trovato due elementi da
           scambiare                                        */
        if inf < sup then
            tmp ← v[sup]
            v[sup] ← v[inf]
            v[inf] ← tmp
        else
            break
        end
    end
    /* spostiamo l'array in posizione sup e restituiamo la
       nuova posizione del pivot                            */
    tmp ← v[sup]
    v[sup] ← v[i]
    v[i] ← tmp
    return sup
end
```

```
begin quick_sort(T[1..n] v)
    quick_sort_rec(v, 1, n)
end

begin quick_sort_rec(T[1..n] v, int i, int f)
    if i ≥ f then
        return
    end
    m ← partition(v, i, f)
    quick_sort_rec(v, i, m-1)
    quick_sort_rec(v, m+1, f)
end
```

L'algoritmo ha costo $\Theta(n)$ nel caso pessimo, che si verifica con la scelta di un ***pivot tale che sia massimo*** *del vettore. Tuttavia nel*

caso ottimo si ha costo $\Theta(n \log_2 n)$ *dal master theorem, così come si può povare che nel* ***caso medio*** *si ha costo* $O(n \log_2 n)$.

Si noti che con questo specifico algoritmo di partizione è failce trovare istanze in input che portano al caso pesismo (i.e. massimo come primo valore), per cui possiamo dunque rendere la scelta del pivot pseudo-casuale onde evitare tale problema.

Algoritmo 13 (Merge Sort)**.** *Il MergeSort e' un algoritmo divide et impera che richiede tempo pseudo-lineare per compiere il suo scopo. Come tipico è diviso in due parti:*

1. ***divide****: si divide a metà l'array in input, senza modificarlo in alcun modo*
2. ***impera****: si chiama ricorsivamente MergeSort sui sottoarray per ordinarli, e poi vengono riuniti nell'array finale nell'ordine corretto (si guardano la testa e la coda).*

```
begin merge_sort(T[1..n] v)
 | merge_sort(v, 1, n)
end

begin merge_sort_rec(T[1..n] v, int i, int f)
 | if i ≥ f then
 |  | return
 | end
 | m ← integer of (f−i)/2
 | merge_sort_rec(v, i, m)
 | merge_sort_rec(v, m+1, f)
 | merge(v, i, m, f)
end
```

```
begin merge(T[1..n] v, int i, int m, int f)
 | /* copiamo i due sottoarray                                  */
 | l ← v[1..m − i]
 | r ← v[1..f − m + 1]
 | /* facciamo l'unione di essi                                 */
 | a, b ← 1
 | k ← i
 | while a < m − i ∧ b < f − m + 1 do
 |  | if l[a] ≤ r[b] then
 |  |  | v[k] ← l[a]
 |  |  | a ← a + 1
 |  | else
 |  |  | v[k] ← r[b]
 |  |  | b ← b + 1
 |  | end
 |  | k ← k + 1
 | end
 | /* copiamo qualunque elemento sia rimasto in l o r           */
 | while a < m − 1 do
 |  | v[k] ← l[a]
 |  | a ← a + 1
 |  | k ← k + 1
 | end
 | while b < m − 1 do
 |  | v[k] ← l[b]
 |  | b ← b + 1
 |  | k ← k + 1
 | end
end
```

Per il master theorem e' possibile verificare che il MergeSort ha costo

$O(n \log_2 n)$ *in ogni casistica.*

Algoritmo 14 (Heap Sort)**.** *L'HeapSort sfrutta la struttura Heap (costruita direttamente sull'array di partenza) per ordinare gli elementi. Vediamo l'algoritmo*

e le sue funzioni ausiliarie:

```
begin heap_sort(T[1..n] v)
    heapify(v, n, 1)
    for i ← n back to 1 do
        max ← find_max(v)
        delete_max(v, i)
        v[i] ← max
    end
end
```

```
begin heapify(T[1..n] v, int len, int i)
    if i > len then
        return
    end
    heapify(v, len, i · 2)
    heapify(v, len, i · 2 + 1)
    fix_heap(v, len, i)
end

begin fix_heap(T[1..n] v, int len, int i)
    /* finiamo se l'heap radicato in i non può avere figli */
    if i · 2 > len then
        return
    end
    /* lo scopo di questo algoritmo e' di spostare v[i] che è
       la radice di un sottoheap nella sua foglia destra o
       sinistra in caso esse esistano e non rispettino le
       proprieta' dei max-heap                                */
    max ← i ∗ 2
    if i · 2 ≤ len ∧ v[i · 2 + 1] < v[i · 2 + 1] then
        max ← i · 2 + 1
    end
    /* dopo aver trovato il massimo tra i figli di destra e
       sinistra, sostituiamo se necessario e controlliamo che
       l'albero in cui abbiamo sostituito sia corretto
       (chiamata ricorsiva)                                   */
    if v[i] < s[max] then
        tmp ← v[i]
        v[i] ← v[max]
        v[max] ← tmp
        fix_heap(v, len, max)
    end
end

begin find_max(T[1..n] v) → T
    return v[1]
end

begin delete_max(T[1..n] v, int len)
    /* rimpiazzo il valore massimo (la radice dell'heap) con
       l'ultimo valore e poi chiamo fix_heap per sistemare la
       struttura dati                                         */
    v[1] ← v[len]
    fix_heap(v, len, 1)
end
```

La funzione ***heap_sort*** *chiama* ***heapify*** *che ha costo* $O(n)$ *e poi ha un loop con* n *iterazioni e chiamate a* ***find_max*** *e* ***delete_max*** *dal costo* $O(\log_2 n)$*, dando un costo complessivo:* $O(n + O(n \cdot \log_2 n) = O(n \log_2 n)$.

Algoritmo 15 (Counting Sort)**.** *Il CountingSort è il primo degli algoritmi di sorting* ***non*** *basato su confronti, poichè opera su insiemi ristretti: deve infatti ricevere in input un array di valori* $v[1..n]$ *tali che i valori* $\forall j \in \{1, \ldots, n\}$. $v[j] \in \{0, \ldots, k\}$.

```
begin counting_sort(T[1..n] v, int k)
    c ← 0...0 k times
    for i ← 1 to n do
        c[i] ← c[i] + 1
    end
    pos ← 1
    for i ← 1 to k do
        while c[i] > 0 do
            v[pos] ← i
            c[i] ← c[i] − 1
            pos ← pos + 1
        end
    end
end
```

Ha costo $O(n + k)$ *ma nel caso in cui* $k = \Theta(n)$ *si ha costo lineare* $O(n)$.

Algoritmo 16 (Buket Sort)**.** *Il BucketSort ha un'idea simile al CountingSort tuttavia ci consente di ordinare anche array di strutture complesse, a patto che esse contengano una chiave con cui indicizzarle e tale chiave sia compresa in*

un intervallo $[1..k]$.

```
begin bucket_sort(T[1..n] v, int k)
    c ← empty_list...empty_list k times
    for i ← 1 to n do
        /* appendi v[i] alla lista in c[v[i].key]                    */
        /* con c ← ... indichiamo l'operazione appendi               */
        c[v[i].key] ← v[i]
    end
    /* reinseriamo i valori delle liste in c in v                    */
    pos ← 1
    for i ← 1 to k do
        while c[i] is not empty do
            /* si sottoinende che l'operazione di estrazione
               dalla lista ← c rimuova anche l'elemento dalla
               lista                                                 */
            v[pos] ← c[i]
            pos ← pos + 1
        end
    end
end
```

Ha costo $O(n+k)$ *ma si nota che nel caso in cui* $n \log_2 n < n + k$ *l'algoritmo non risulta conveniente rispetto ai sorting pseudo-lineari.*

Algoritmo 17 (Selezione dek k-esimo)**.** *Supponiamo di voler selezionare il k-esimo minimo in un array, senza dunque la necessità di ordinarlo interamente. Possiamo usare una sorta di SelectionSort accorciato, dove svolgiamo solo i primi k passi in modo da avere i primi k elemento ordinati e restituiamo $v[k]$.*

```
begin kselect(T[1..n] v, int k)
    for i ← 1 to k do
        min ← i
        for j ← i + 1 to n do
            if v[j] < v[min] then
                min ← j
            end
        end
        if min ≠ i then
            swap(v, min, i)
        end
    end
    return v[k]
end
```

L'algoritmo ha chiaramente costo $O(kn)$.

Algoritmo 18 (Selezione dek k-esimo (heapsort))**.** *Posso costruire una versione alternativa per valori di k bassi utilizzando un HeapSort accorciato.*

```
begin heap_select(T[1..n] v, int k)
    heapify(v, k, i)
    for i ← 1 to k − 1 do
        delete_min(v, k-i)
    end
    return find_min(v)
end
```

*Otteniamo dunque un costo complessivo di $O(n+k\log_2 n)$, dove $O(n)$ è dato da **heapify** e $O(k\log_2 n)$ è dato dal ciclo e **delete_min**.*

Algoritmo 19 (Selezione dek k-esimo (QuickSort))**.** *Possiamo ancora adattare l'approccio divide-et-impera del QuickSort per il problema della selezione del k-esimo. Sfrutteremo la funzione **parition** usata nel problema della bandiera nazionale, in modo da avere tre sottoinsiemi (valori minori del pivot, valori uguali al pivot, valori maggiori del pivot). Poi essendo una selezione analizzeremo solo*

uno dei tre sottoinsiemi con la chiamata ricorsiva.

```
begin quick_select(T[1..n] v, int k)
    /* si sceglie il pivot in modo arbitrario                    */
    pivot ← v[1]
    v_1 ← {x ∈ v | x < pivot}
    v_2 ← {x ∈ v | x = pivot}
    v_3 ← {x ∈ v | x > pivot}
    if k ≤ |v_1| then
        return quick_select(v_1, k)
    else if k ≤ |v_1| + |v_2| then
        return pivot
    else
        return quick_select(v_3, k − |v_1| − |v_2|)
    end
end
```

Nel caso peggiore l'algoritmo mantiene costo $O(n^2)$ come QuickSort, tuttavia nel caso migliore e medio si ha costo $O(n)$ (si può provare che $T_{mid}(n) \leq 4n$)

Algoritmo 20 (Sottovettore massimo)**.** *Rivediamo in programmazione dinamica il problema del sottovettore massimo, che essendo un problema di ottimizzazione dove bisogna controllare più volte gli stessi sottovettori si presta all'applicazione della tecnica di programmazione dinamica.*
L'idea è quella di risolvere i sottoproblemi $S[i]$ che ci danno il valore della migliore somma di un sottovettore di $v[1..i]$ che contenga $v[i]$ come ultimo elemento. Possiamo poi combinare le soluzioni ottenute dai sottoproblemi precedenti per trovare la soluzione finale.

```
begin max_subset(int[1..n] v) → int
    best ← 1
    results[1..n]
    results[1] ← v[1]
    for i ← 2 to n do
        if results[i − 1] + v[i] < v[i] then
            results[i] ← v[i]
        else
            results[i] ← results[i − 1] + v[i]
        end

        if results[best] < results[i] then
            best ← i
        end
    end
    return results[best]
end

begin max_subset_start(int[1..n] v, int[1..n] results, int best) →
 int
    i ← best
    while results[i] ≠ v[i] do
        i ← i − 1
    end
    return i
end
```

L'algoritmo ha costo $O(n)$, migliore rispetto alla versione divide-et-impera. Oltretutto abbiamo un algoritmo per trovare gli indici di inizio e fine dell'sottoarray scelto.

Algoritmo 21 (DFS su grafi)**.** *La DFS sui grafi è molto simile ad una DFS su alberi, tuttavia con i grafi bisogna tenere traccia di quali nodi sono stati* ***non visitati****(bianco),* ***aperti****(grigio) o* ***visitati****(nero). Terremo oltretutto traccia dell'ordine in cui i nodi vengono visitati in modo da poter costuire una* ***foresta DF*** *che indica i nodi visitati nel rispettivo ordine.*

```
global time ← 0
begin DFS(Graph (V, E), fn visit)
    for v in V do
        v.mark ← white
        v.parent ← null
    end
    for v in V do
        if v.marked = white then
            DFS_visit(v)
        end
    end
end

begin DFS_visit(Vertex v, fn visit)
    v.mark ← gray
    time ← time + 1
    v.dt ← time
    for u adjacent to v do
        if u.marked = white then
            u.parent ← v
            DFS_visit(u, visit);
        end
    end
    /* Questa chiamata può essere posta prima del loop per
       ottenere una visita DFS in pre-ordine                */
    visit(v)
    time ← time + 1
    v.ft ← time
    v.mrak ← black
end
```

Algoritmo 22 (BFS su grafi)**.** *La BFS sui grafi è molto simile alla sua contropare sugli alberi, ma anche in questo caso bisogna tenere traccia dei vertigi già visitati. Oltretutto utilizzeremo una struttura arobrescente T per costruire*

*un **Albero di visita** che traccia archi e vertici percorsi durante la visita.*

```
begin BFS(Graph (V, E), Vertex s, fn visit) → Tree
    for v in V do
        v.mark ← false
    end
    Tree tree ← s
    Queue queue ← s
    s.dist ← 0
    s.mark ← true
    while queue is not empty do
        v ← queue
        visit(v)
        for u adjacent to v do
            if u.marked = false then
                u.dist ← v.dist + 1
                u.mark ← true
                u.parent ← v
                queue ← u
            end
        end
    end
    return tree
end
```

Algoritmo 23 (MST: Kruscal)**.** *Descriviamo l'algoritmo di Kruscal per produrre un Minimum Spanning Tree di un dato grafo. L'idea è semplice: Inseriamo ogni vertice in un albero, poi ordiniamo in modo non decrescente di peso gli archi, analizzandone uno a uno e comportandoci nel seguente modo:*

1. *se l'arco forma un ciclo (possiamo vederlo confrontando l'identificatore dei due vertici nella UnionFind) allora non lo inseriamo nella soluzione.*

2. *altrimenti tale arco viene inserito nella soluzione.*

```
begin kruskal(Graph (V, E, w)) → Tree
    UnionFind uf
    Tree tree
    for v in V do
        uf.make_set(v)
    end
    sort_crescente(E, w)
    for (u, v) in E do
        T_u ← uf.find(u)
        T_v ← uf.find(v)
        if T_u ≠ T_v then
            tree ← tree ∪ (u, v)
            uf.union(T_u, T_v)
        end
    end
    return tree
end
```

*L'algoritmo ha costo $O(m \log_2 n)$, dato dal costo dell'ordinamento che è pari a $O(m \log_2 m) = O(m \log_2 n)$ e dal costo delle chiamate a union find. Poichè prevalgono le chiamate a **union** useremo la struttura QuickFind con euristica sul rango, ottenendo un costo $O(m + 2m \log_2 n + n) = O(m \log_2 n)$. Le precedenti equivalenze valgono poichè in un grafo connesso si ha sempre $m \geq n - 1$.*

Algoritmo 24 (MST: Prim)**.** *Descriviamo l'algoritmo di Prim per produrre un Minimum Spanning Tree di un dato grafo. L'idea è la seguente: partire da un vertice sorgente qualsiasi, tenere tracia dei nodi raggiungibili e con quali costi, aggiornando i costi mentre troviamo nodi più vantaggiosi e mantenendo*

la frontiera in una coda con priorità.

```
begin prim(Graph (V, E, w), Vertex s) → Tree
    d[1..n] ← {+∞, ..., +∞}'
    b[1..n] ← {false, ..., false}'
    p[1..n] /* array di padri di ogni nodo nell'MST finale   */
    PriorityQueue q
    d[s] ← 0
    q.insert(s, d[s])
    while q is not empty do
        v ← q /* find e delete_min                            */
        b[v] ← true
        for u adjacent to v not b[u] do
            if d[u] = +∞ then
                q.insert(u, w(v, u))
                d[u] ← w(v, u)
                p[u] ← v
            else if d[u] > w(v, u) then
                q.decrease_key(u, d[u] − w(v, u))
                d[u] ← w(v, u)
                p[u] ← v
            end
        end
    end
    return p
end
```

L'algoritmo fa n chiamate di ***delete_min*** *che hanno costo* $O(n \log_2 n)$*, n chiamate di* ***insert*** *con costo* $O(n \log_2 n)$ *e m chiamate di* ***decrease_key*** *che hanno costo* $O(m \log_2 n)$*. Il totale ammonta a* $O(n \log_2 n + n \log_2 n + m \log_2 n)$ *e poichè* $m \geq n - 1$ *si ha come costo complessivo:* $O(m \log_2 n)$*.*

Algoritmo 25 (Cammini minimi: Bellman-Ford)**.** *Un cammino è un percorso $\pi = (v_0, v_1, \ldots, v_k)$ che va dalla sorgente v_0 al nodo destinazione v_k. Un cammino si dice minimo se il valore della sommatoria*

$$\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

è il minimo tra tutti i cammini esistenti nel grafo. L'algoritmo di Bellman-Ford risolve il problea della ricerca di un cammino minimo da un nodo s a tutti gli altri nodi del grafo.

```
begin bellman_ford(Graph (V, E, w), Vertex s) → Tree
    n ← number of vertices in V
    int pred[1..n]
    double D[1..n]
    for v in V do
        D[v] ← ∞
        pred[v] ← −1
    end
    D[s] ← 0
    repeat
        for (u, v) in E do
            if D[v] > D[u] + w(u, v) then
                D[v] ← D[u] + w(u, v)
                pred[v] ← u
            end
        end
    until n−
    /* controllo per cicli negativi                    */
end
```

> *Il costo dell'algoritmo è determinato dal secondo ciclo, che ha chiaramente costo $O(nm)$.*

L'idea del controllo per i cili negativi è semplice. Se esistono cicli negativi allora possiamo ancora trovare un cammino conveniente anche dopo aver completato Bellman-Ford, che tuttavia avrebbe già dovuto trovare tutti i cammini migliori. Possiamo sfruttare questo semplice pseudocodice:

```
for (u, v) in E do
    if D[v] > D[u] + w(u, v) then
        throw error cicli negativi
    end
end
```

Algoritmo 26 (cammini minimi: Dijkstra (uguale a Prim))**.** *L'algoritmo di Dijkstra è estremamente simile a Prim (attenzione al calcolo della distanza dei*

nodi dalla sorgente s), tuttavia ci consente di trovare in modo ottimale cammini minimi in grafi ***senza archi di peso negativo****.*

```
begin dijkstra(Graph (V, E, w), Vertex s) → Tree
    double d[1..n] ← {+∞, ..., +∞}'
    int p[1..n] /* array del percorso del cammino
        v → u ⇒ p[u] = v                                   */
    PriorityQueue q
    d[s] ← 0
    q.insert(s, d[s])
    while q is not empty do
        v ← q /* find e delete_min                         */
        for u adjacent to v do
            if d[u] = +∞ then
                q.insert(u, d[u] + w(v, u))
                d[u] ← d[u] + w(v, u)
                p[u] ← v
            else if d[u] > w(v, u) then
                q.decrease_key(u, d[u] − d[v] − w(v, u))
                d[u] ← d[v] + w(v, u)
                p[u] ← v
            end
        end
    end
    return p
end
```

Proprio come Prim (cambia solo il calcolo della distanza tra i nodi) questo algoritmo ha costo $O(m \log_2 n)$*.*

Algoritmo 27 (cammini minimi (all-pair): Floyd-Warshall all-pair)**.** *Algoritmo utilizzato per trovare i cammini minimi da una qualunque sorgente ad una qualunque destinazione. Useremo la tecnica della programmazione dinamica e risolveremo i sottoproblemi del tipo: Trovare la distanza minima da x a y passando solo per i primi* $\{1..k\}$ *vertici. Le soluzioni* $S^k_{x,y}$ *avranno la forma del tipo:*

$$S[x,y,0] = \begin{cases} 0 & se\ x = y \\ w(x,y) & se\ (x,y) \in E \\ \infty & se\ (x,y) \notin E \end{cases}$$

$$S[x,y,k>0] = \texttt{min}\{S^{k-1}_{x,y}, S^{k-1}_{x,k} + S^{k-1}_{k,y}\}$$

Si noti che la soluzione generale prende il minimo tra le due opzioni che ci si presentano in ogni caso non banale, ossia:

- ***non*** *prendo il nodo k: allora la soluzione sarà uguale a quella precedente, ovvero* $S[x,y,k-1]$*.*

- *prendo il nodo k: allora la soluzione sarà data dalla soluzione fino a k con $k-1$ nodi, e dalla soluzione da k a y con $k-1$ nodi, ovvero $S[x,k,k-1]+S[k,y,k-1]$.*

---

begin `floyd_warshall(`**Graph** *(V, E,* `w`*))` $\rightarrow$ **Tree**
  $n \leftarrow$ *amount of vertices in V*
  double $S[1..n, 1..n, 0..n]$ `/*` $S[x,y,k]$ `*/`
  `/* riempiamo le soluzioni del tipo` $S[x,y,0]$ `*/`
  for $x \leftarrow 1$ **to** n **do**
    for $y \leftarrow 1$ **to** n **do**
      if $x = y$ **then**
        $S[x,y,0] \leftarrow 0$
      else if $(x,y) \in E$ **then**
        $S[x,y,0] \leftarrow$ `w` *(x,y)*
      else
        $S[x,y,0] \leftarrow \infty$
      end
    end
  end
  `/* calcolo le soluzioni non banali */`
  for $k \leftarrow 1$ **to** n **do**
    for $x \leftarrow 1$ **to** n **do**
      for $y \leftarrow 1$ **to** n **do**
        $S[x,y,k] \leftarrow$ `min`$\{S[x,y,k-1], S[x,k,k-1]+S[k,y,k-1]\}$
      end
    end
  end
  return $S[1..n, 1..n, k]$
end

> *Vista la necessità di popolare una matrice tridimensionale di dimensione n (cardinalità dei vertici), l'algoritmo avrà sicuramente dimensione $O(n^3)$ sia per quanto riguarda il tempo che lo spazio.*

Si può ottimizzare sotto il punto di vista dello spazio l'algoritmo notando che l'indicizzazione di k è supeflua in quanto ci si rifà sempre al valore precedente $k-1$ che viene mantenuto invariato o aggiornato, ma non si ispeziona mai più indietro di $k-1$ dunque basterebbe una matrice $n \times n$ e applicare i cambiamenti in loco anzi che tenendone traccia al variare di k.
La soluzione seguente contiene anche una matrice di appoggio per poter rico-

striure i cammini da un nodo all'altro.

```
begin floyd_warshall_optimized(Graph (V, E, w)) → Tree
    int n ← amount of vertices in V
    double S[1..n, 1..n]
    Vertex next[1..n, 1..n]
    /* riempiamo le soluzioni del tipo S[x, y, 0]                    */
    for x ← 1 to n do
        for y ← 1 to n do
            if x = y then
                S[x, y] ← 0
                next[x, y] ← −1
            else if (x, y) ∈ E then
                S[x, y] ← w (x,y)
                next[x, y] ← y
            else
                S[x, y] ← ∞
                next[x, y] ← −1
            end
        end
    end
    /* calcolo le soluzioni non banali                               */
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                /* si noti che il primo valore del minimo S^{k−1}_{x,y} è
                   già in S[x, y]                                    */
                if S[x, k] + S[k, y] < S[x, y] then
                    S[x, y] ← S[x, k] + S[k, y]
                    next[x, y] ← next[x, k]
                end
            end
        end
    end
    return S[1..n, 1..n, k]
end

begin floyd_warshall_path(Vertex[1..n] next, Vertex x, Vertex
 y)
    if x ≠ y ∧ next[x, y] < 0 then
        errore: non esiste un cammino da x a y
    else
        print(x)
        while x ≠ y do
            x ← next[x, y]
            print(x)
        end
    end
end
```