Appunti di Algoritmi e strutture dati

Riccardo Lo Iacono

Dipartimento di Matematica & Informatica
Università degli studi di Palermo
Sicilia
a.a. 2022-2023

Indice.

– 1 – Introduzione.

Dato un problema, è importante chiedersi come trovare una soluzione allo stesso. Inoltre, supposto che esista una soluzione algoritmica A, è opportuno poter confrontarla con una soluzione B.

– 1.1 – Algoritmi e complessità.

La valutazione di algoritmi può essere effettuata secondo diversi criteri. In generale, di interesse sono la velocità di crescita in termini di *spazio* e *tempo*.
In generale, il tempo necessario ad un algoritmo, espresso come funzione della taglia del problema, è detta *complessità di tempo*. Dicasi *complessità asintotica di tempo* il comportamento limite della complessità di tempo, al crescere della taglia[1] del problema.

– 1.2 – Notazioni asintotiche.

Le diverse complessità di un'algoritmo possono essere studiate secondo tre aspetti: *caso ottimo, caso pessimo, caso medio.*

– 1.2.1 – Caso ottimo: notazione Omega.

La notazione Omega Ω definisce un limite inferiore ad un funzione $f(n)$. In generale, data una certa funzione $g(n)$ si definisce $\Omega(g(n))$ come segue.

$$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}, n_0 \in \mathbb{N}, c, n > 0 \mid 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0\}$$

– 1.2.2 – Caso pessimo: notazione O-grande.

La notazione O-grande definisce un limite superiore ad una funzione $f(n)$. In generale, data una certa funzione $g(n)$ si definisce $\mathcal{O}(g(n))$ come segue.

$$\mathcal{O}(g(n)) = \{f(n) : \exists c \in \mathbb{R}, n_0 \in \mathbb{N}, c, n_0 > 0 \mid f(n) \leq c \cdot g(n), \forall n \geq n_0\}$$

– 1.2.3 – Caso medio: notazione Theta.

La notazione Theta Θ definisce dei limiti ad una funzione $f(n)$. In generale, data una certa funzione $g(n)$ si definisce $\Theta(g(n))$ come segue.

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N} \mid c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0\}$$

[1] Indica la misura della quantità di dati in input.

– 2 – Modelli di computazione.

Prima di poter parlare di algoritmi, è necessario stabile il *modello di calcolo* con cui si intende risolvere un problema; si pensi ad un modello di calcolo come un modello che descrive come l'output sia ottenuto in funzione dell'input.

Nella presente sezione si descriveranno due dei modelli più diffusi quali *modello RAM* e *macchina di Turing*.

– 2.1 – Modello RAM.

Il modello RAM[2], modella un computer in cui le istruzioni non modificano se stesse. Procedendo all'analisi strutturale di una RAM questa si compone di due nastri: uno di input, uno di output, ciascuno con la propria testina; un programma e una memoria.

Il nastro di input è rappresentato come una sequenza di celle, ciascuno contente un intero: ogni volta che un'istruzione di lettura è eseguita la testina è spostata di un posto verso destra.

Il nastro di output ha una struttura analoga a quello di input, con la differenza che inizialmente è completamente vouto. Quando un'istruzione di scrittura è eseguita sulla cella puntata dalla testina è impresso un intero, successivamente la testina si sposta di un cella a destra.

La memoria è un insieme di registri $r_0, r_1, \ldots, r_i, \ldots$, ciascuno con la capacità di memorizzare un intero di taglia arbitraria.

Il programma, non risiedente in memoria, per cui si assume non auto-modificante, è una mera sequenza di istruzioni[3], opzionalmente, etichettate.

Nota: Tutte le computazioni avvengo in r_0, l'*accumulatore*.

[2] La sigla RAM sta ad indicare *Random Access Machine*.

[3] La natura delle istruzioni è trascurabile fintanto che queste somiglino a istruzioni macchina reali.

– 2.1.1 – Istruzioni e analisi di un programma RAM.

Le istruzioni fondamentali di un programma RAM sono quelle di *Figura 2.1.* A queste è possibile aggiungere ulteriori istruzioni, sempre con la condizione che questi siano simili ad istruzioni reali. Ciascun'istruzione si compone di due parti un *codice* e un *indirizzo*.

Codice	Indirizzo
LOAD	operando
STORE	operando
ADD	operando
SUB	operando
MULT	operando
DIV	operando
READ	operando
WRITE	operando
JUMP	label
JZERO	label
JGTZ	label
HALT	

Figura 2.1: Tabella delle istruzioni RAM

Gli operandi sono di tre tipi:

1. $= i$: indica l'intero stesso.
2. i: un intero non negativo che indica il contenuto dell'i-esimo registro.
3. $*i$: indica un indirizzamento indiretto. Cioè i è il contenuto del registro j. Se $j < 0$ la macchina si arresta.

Considerando un programma P, il suo significato è definibile tramite due quantità: una funzione $c : \mathbb{Z} \to \mathbb{N}$ e un *contatore di posizione* nel programma.
Inizialmente $c(i) = 0, \forall i \geq 0$ e il contatore di posizione è posto alla prima istruzione del programma. Ai fini di comprendere il significato di un'istruzione, si definisca $v(a)$, definita come segue, il valore dell'operando a.

$$\begin{aligned} v(= i) &= i \\ v(i) &= c(i) \\ v(*i) &= c(c(i)) \end{aligned}$$

In generale, si può pensare ad un programma RAM come una "mappatura" dei dati di in input in quelli di output. Sebbene esistano varie interpretazioni di tale mappatura, due delle più importanti sono le interpretazioni di analizzatore di funzioni e di linguaggi.

Procedendo all'analizzare le due rappresentazioni prima citate, si ha quanto segue.

1. Sia P un programma che legge sempre n interi dal nastro di input, e che al più scriva un carattere su quello di output. Se $x_1, x_2, \ldots, x_n$ sono gli interi nelle prime n celle del nastro di input, P scrive y nella prima cella del nastro di output, e fatto ciò si arresta; si dirà che P computa una funzione $f(x_1, x_2, \ldots, x_n) = y$.

2. Sia Σ un certo alfabeto; rappresentando con $1, \ldots, k$ i simboli dell'alfabeto, una RAM accertatrice di linguaggi si comporta come segue. Se $s = a_1 a_2 \ldots a_n$ è una stringa, si pone s sul nastro di input e nella (n + 1)-esima cella si pone un carattere terminale. Si dirà che un programma P accetta s se, questi legge l'intera stringa e il carattere terminale, scrive sul nastro di output 1 e si arresta. Con tale definizione, un linguaggio accettato è l'insieme delle stringhe accettate.

Esempio: Sia $f(n)$ la funzione di seguito definita, scrivere un programma RAM che la calcoli.

$$f(n) = \begin{cases} n^n, & \text{se } n \geq 1 \\ 0, & \text{altrimenti} \end{cases}$$

In *Figura 2.2* rispettivamente l'implementazione algoritmica e quella in RAM.

```
begin
    read r1
    if r1 ≤ 0 then write 0;
    else
        begin
            r2 ← r1;
            r3 ← r1 − 1;
            while r3 > 0 do
                begin
                    r2 ← r2 * r1;
                    r3 ← r3 − 1;
                end
            write r2;
        end
end
```

2.2.a: Algoritmo per n^n

```
        READ 1
        LOAD 1
        JGTZ pos
        WRITE =0
        JUMP enfif
pos:    LOAD 1
        STORE 2
        LOAD 1
        SUB =1
        STORE 3
while:  LOAD 3
        SUB =1
        JGTZ cont
        JUMP endw
cont:   LOAD 2
        MULT 1
        STORE 2
        LOAD 3
        SUB =1
        STORE 3
        JUMP while
endw:   WRITE 2
endif:  HALT
```

2.2.b: Programma ram per n^n

Figura 2.2: Implementazioni di $f(n)$

– 2.2 – Complessità di un programma RAM.

Il modello RAM sinora descritto è definito come *modello RAM con costo uniforme*: in questo caso ogni istruzione richiede un'unità di tempo e ogni registro un'unità di spazio. Un'ulteriore modello è il *modello RAM con costo logaritmico*, analizzato nel paragrafo seguente.

– 2.2.1 – RAM con costo logaritmico.

Nel modello con costo logaritmico si tiene conto della dimensione finita della memoria, nella fattispecie della dimensione limitata di una WORD. In tal senso, sia $l(i)$ la seguente funzione che stabilisce il numero di bit per rappresentare l'operando

$$l(i) = \begin{cases} \lfloor \log |i| \rfloor + 1, & \text{se } i \neq 0 \\ 1, & \text{se } i = 0 \end{cases}$$

Si ha quindi che il costo per accedere agli operandi è quanto segue

$$\begin{aligned} =i: &\quad l(i) \\ i: &\quad l(i) + l(c(i)) \\ *i: &\quad l(i) + l(c(i)) + l(c(c(i))) \end{aligned}$$

Analizzando, a titolo di esempio, l'istruzione **ADD** *i, da quanto sopra si necessita $l(i) + l(c(i)) + l(c(c(i)))$ per accedere ad *i, a cui aggiungere il costo di accesso all'accumulatore pari a $l(c(0))$.

– 2.3 – La macchina di Turing.

Prima di discutere la macchina di Turing, è necessario parlare di *relazione polinomiale*

Definizione: Due funzioni $f_1(n)$ e $f_2(n)$ sono in relazione polinomiale se esistono $p_1(x)$ e $p_2(x)$ tali da soddisfare la seguente relazione.

$$f_1(n) \leq p_1(f_2(n)) \wedge f_2 \leq p_2(f_1(n)) \qquad ,\forall n$$

Esempio: Siano $f_1(n) = 2n^2$ e $f_2(n) = n^5$: si osserva che queste sono in relazione polinomiale. Infatti se $p_1(x) = 2x$ e $p_2(x) = x^3$ segue: $2n^2 \leq 2n^5$ e $n^5 \leq (2n^2)^3$

Si considera ora un nuovo modello di calcolo, la *macchina di Turing*.

Definizione: Una macchina di Turing (MT) multi-nastro si compone di k nastri, ciascuno dei quali è diviso in celle, ciascuna delle quali contiene un simbolo di nastro. Le operazioni sono dettate da un programma primitivo:il *controllo finito*.

In una computazione, in accordo col controllo finito e il carattere puntato dalla testina di ciascun nastro, una MT può effettuare almeno un delle seguenti operazioni.

1. Cambiare lo stato del controllo finito.
2. Sovrascrivere uno o più caratteri nelle celle indicate dalle testine.
3. Per ciascun nastro, in maniera indipendente, spostare la testina verso destra, sinistra o lasciarla li.

Formalmente una generica MT viene identificata tramite la settupla[4]

$$(Q, T, I, \delta, b, q_0, q_f)$$

L'attività di una MT può formalmente essere descritta tramite istantanee (ID). Quest'ultime sono k-ple $\alpha_1, \ldots, \alpha_k$, con α_i una stringa del tipo xqy tale che, xy sia la stringa dell'i-esimo nastro esclusi i black iniziali e finali, con q lo stato della MT.

[4] Per il significato di ciascun componente si veda "Appunti di Informatica teorica".

– 2.4 – Relazione tra MT e RAM.

Principale applicazione delle MT è quello di determinare un *lower bound* al tempo e allo spazio necessari alla risoluzione di un problema.

Considerando la relazione tra MT e RAM, risulta ovvio che una RAM possa simulare una MT a k nastri, semplicemente mantenendo un cella del nastro della MT in un registro. Si supponga una MT con complessità di tempo $T(n) \geq n$; ne segue che una RAM legga gli input in $\mathcal{O}(T(n))$ se a costo uniforme, in $\mathcal{O}(T(n)\ln T(n))$ se con costo logaritmico. In ambo i casi, il tempo di una RAM è limitato superiormente dal tempo della MT. Un risultato inverso, cioè la possibilità che una MT simuli un RAM, vale solo se la RAM è con costo logaritmico. In tal caso vale il teorema a seguire.

Teorema 2.1.
Sia L un linguaggio accettato da una RAM con costo logaritmico, in tempo $T(n)$. Se la RAM non fa uso di istruzioni MULT *e/o* DIV*, allora la complessità di tempo è al più $\mathcal{O}\left(T^2(n)\right)$.*

Dimostrazione: Sia rappresentato ciascun registro della RAM, non contenente zeri, come in *Figura 2.3*.

#	#	i_1	#	c_1	#	#	$\cdots$	#	#	i_k	#	c_k	#	#	$\cdots$

Figura 2.3: MT simulante RAM con costo logaritmico.

Il nastro è dunque una sequenza di coppie (i_j, c_j), scritte in binario, separate da un delimitatore. Il contenuto dell'accumulatore è memorizzato su di un secondo nastro, e un terzo nastro è usato come supporto. In aggiunta a questi vi sono due nastri aggiuntivi: uno per gli input e uno per gli output della RAM. Segue che un passo della RAM è rappresentato da un insieme finito di passi della MT.

Si procede ora col dimostrare che una RAM con costo computazionale k richiede al più $\mathcal{O}\left(k^2\right)$ passi di una MT. Il costo per memorizzare c_j in i_j è $l(c_j) + l(i_j)$, da cui si conclude che la lunghezza di caratteri non blank è $\mathcal{O}(k)$.
La simulazioni di istruzioni diverse da STORE sono $\mathcal{O}(k)$, poiché il costo principale è dato dalla ricerca nel nastro. Analogamente il costo di uno STORE è dato dal tempo di ricerca nel nastro, in aggiunta al tempo necessario per copiarlo entrambe $\mathcal{O}(k)$. Da cio si deduce che, a meno di MULT e/o DIV, un'istruzione RAM può essere simulata in $\mathcal{O}(k)$ passi della MT. Infine poiché sotto il criterio logaritmico ciascuna istruzione RAM costa almeno un'unita di tempo, il costo totale speso dalla MT è $\mathcal{O}\left(k^2\right)$ □

– 3 – Design di algoritmi efficienti.

La costuzione di algoritmi efficiente è generalmente effettuata sfruttando apposite tecniche di programmazione, e di diverse strutture dati.

– 3.1 – Strutture dati.

In questa sezione saranno trattate brevemente strutture quali *grafi* e *alberi*. L'analisi di strutture quali *insiemi*, *dizionari* e varianti degli alberi saranno analizzate nelle successive sezioni.

– 3.1.1 – Grafi.

Definizione: Un grafo $G = (V, E)$ si compone di un insieme finito e non vuoto di vertici V, e uno di archi E. Si dirà che G è *orientato* (o diretto) se

$$\forall (v, w) \in E, v \to w^{5} \tag{1}$$

Si dirà che G non è orientato se non vale *Eq.* (1).

Dato G un grafo orientato, si dirà che due vertici v e w sono *adiacenti* se $(v, w) \in E$. Se G è invece un grafo non orientato, i dirà che due vertici v e w sono *adiacenti* se $(v, w) \in E \land (w, v) \in E$: si assume infatti che $(v, w) = (w, v)$.

Definizione: Dato G un grafo, orientato o meno, dicasi *cammino* una sequenza di archi del tipo $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$.

Definizione: Un cammino semplice[6] di lunghezza almeno 1, che inizia e finisce in uno stesso vertice, è detto *ciclo*.

Osservazione: In un grafo non diretto, un ciclo ha lunghezza almeno 3.

Dato un grafo G questi può essere rappresentato in vari modi, tra queste vi è la rappresentazione come *matrice di adiacenza*. Questa è una matrice $\|V\| \times \|V\|$ di 0 e 1 ove $a_{ij} == 1 \iff (v_i, v_j) \in E$. Considerandone i vantaggi e gli svantaggi, tale rappresentazione è ottimale poiché permette di verificare la presenza di un arco in tempo costante, d'altro canto pero lo spazio richiesto è $\mathcal{O}\left(\|V\|^2\right)$.
Ulteriore rappresentazione è effettuata per *liste di adiacenza*: per ciascun vertice di V è memorizzata una lista di vertici ad esso adiacente.

[5] Con $v \to w$ si intende che (v, w) è una coppia orientata.
[6] Un cammino si dice semplice se ogni arco viene percorso una sola volta

– 3.1.2 – Alberi.

Definizione: Dato G un grafo orientato, aciclico, questi è detto albero se soddisfa le seguenti proprietà.

1. Esiste un vertice v, la *radice*, tale che in esso non entrino archi.
2. Ogni vertice, meno la radice, ha un unico arco d'entrata.
3. Esiste un cammino, che dimostrasi unico, dalla radice verso ogni vertice.

Nota: Se G è un grafo diretto, composto da alberi questi questo è detto *foresta*.

Definizione: Sia $F = (V, E)$ una foresta. Se $(v, w) \in E$ dicasi v *antenato* e w *discendente*. Un vertice senza discendenti è detto *foglia*. Dicasi inoltre l'insieme di un vertice e dei suoi discendenti *sotto-albero* di F.

La *profondità di un vertice* in un albero, è la lunghezza del cammino che vi è dalla radice al vertice. Dicasi *altezza di un vertice* la lunghezza del cammino più lungo dal vertice ad una sua foglia; se il vertice è la radice si parlerà di *altezza dell'albero*. In fine il *livello di un vertice* è l'altezza dell'albero a cui si sottrae la profondità del vertice.

Definizione: Un albero binario è detto *completo* se, per un qualche k, tutti i vertici con profondità minore di k hanno sia figlio destro che sinistro, e ogni vertice ddìi profondità k è una foglia.

Osservazione: Un albero binario di altezza k ha esattamente $2^{k+1} - 1$ foglie.

Molti algoritmi fanno uso degli alberi, sfruttando in genere la visita dello stesso. Tali visite sono riportate di seguito.

- **pre-ordine:** si procede col visitare la radice successivamente, procedendo ricorsivamente, il sotto-albero sinistro e destro.
- **post-ordine:** si procede a visitare il sotto-albero sinistro, il desto e in ultimo la radice.
- **in-ordine:** si procede con il visitare il sotto-albero sinistro, la radice, e in fine il sotto-albero destro.

– 3.2 – Tecniche di programmazione.

In questa sezione si discuteranno alcune tecniche di programmazione, quali *divide-and-conquer* e *programmazione dinamica*.

– 3.2.1 – Divide-and-conquer.

La tecnica divide-and-conquer si fonda sull'idea di suddividere un problema in problemi di taglia inferiore e, risolto ciascun sotto-problema, unire le sotto-soluzioni per risolvere in problema di partenza.

Si supponga di dover ricercare il massimo ed il minimo in un array di elementi S. Un idea semplice è quella di effettuare una procedura del tipo a seguire.

```
begin
    MAX ← x ∈ S
    for y ∈ S, y ≠ x do
        if y > MAX then MAX ← y
end
```

Con un algoritmo tipo quello sopra, si impiegano n - 1 confronti per ricercare il massimo, ed n - 2 per la ricerca del minimo.

Applicando ora una procedura divide-and-conquer, si dimostra che è possibile ricercare sia il massimo che il minimo in meno di $\mathcal{O}(n^2)$ confronti. Sia considerato l'algoritmo a seguito riportato, e sia $T(n)$ il numero di confronti effettuati.

```
procedure MAXMIX(set S):
    if ||S|| == 2 then
        begin
            let S = {a, b};
            return (MAX(a, b), MIN(a, b));
        end
    else
        begin
            divide S in S1 and S2, with both half of the elements of S;
            (max1, min1) ← MANMIX(S1);
            (max2, min2) ← MANMIX(S2);
            return (MAX((max1, max2)), MIN(min1, min2));
        end
```

Analizzando il numero di confronti si osserva

$$T(n) = \begin{cases} 1, & n = 2 \\ 2T(n/2) + 2, & n > 2 \end{cases}$$

Si dimostra che $T(n) = \frac{3}{2}n - 2$ soddisfa la relazione ricorsiva.

Si consideri il prodotto di due interi a n bit. Con il metodo classico, si impiegherebbero $\mathcal{O}\left(n^2\right)$ operazioni; se si applica invece il metodo a seguire, si limita tale numero a $\mathcal{O}\left(n^{\log_2 3}\right)$. Siano x e y i due interi a n bit. Si proceda con il dividere i due, in altrettante parti: considerando ciascuna parte come numero a n/2 bit, il prodotto può essere espresso come

$$\begin{aligned} xy &= (2^{n/2}a + b)(2^{n/2}c + d) \\ &= 2^n ac + (ad + bc)2^{n/2} + bd \end{aligned}$$

Un'implementazione algoritmica risulta essere la seguente

```
begin
    u ← (a + b)(c + d);
    v ← a * c;
    w ← b * d;
    z ← v * 2^n + (u − v − w)2^{n/2} + w
end
```

il cui costo computazionale risulta essere

$$T(n) = \begin{cases} c, & n = 1 \\ 3T(n/2) + cn, & n > 1 \end{cases}$$

– 3.2.2 – Master Theorem.

Come visto, la tecnica di divide-and-conquer genera delle complessità di tempo espresse da equazioni ricorsive, le quali non risultano semplici da analizzare. Per tale ragione si necessità di uno strumento teorico che ne facilita il calcolo, tale strumento è il *Master Theorem*.

Teorema Master.
Data una funzione ricorsiva del tipo

$$T(n) = aT(\frac{n}{b}) + f(n)$$

con $a \geq 1, b > 1$ *ed* f *asintoticamente positiva, allora valgono i seguenti casi:*

- **Caso 1.** *Se* $f(n) = \mathcal{O}\left(n^{\log_b a - \varepsilon}\right), \varepsilon > 0$, *allora* $T(n) = \Theta(n^{\log_b a})$.
- **Caso 2.** *Se* $f(n) = \Theta(n^{\log_b a})$, *allora* $T(n) = \Theta(n^{\log_b a} \log n)$.
- **Caso 3.** *Se* $f(n) = \Omega(n^{\log_b a + \varepsilon}), \varepsilon > 0$, *allora* $T(n) = \Theta(f(n))$.

– 3.2.3 – Programmazione dinamica.

La programmazione dinamica prevede la risoluzione di problemi, tramite la suddivisione e soluzione ricorsiva di sotto-problemi dello stesso.

Affinché un problema sia risolvibile con la programmazione dinamica è necessario che questo soddisfi certe condizioni:

1. deve presentare una sotto-struttura ottimale, con la quale si intende la possibilità di trovare una soluzione ottimale come combinazione delle soluzioni ai suoi sotto-problemi;

2. i sotto-problemi devono essere sovrapponibili.

In generale la computazione procede da sotto-problemi di taglia inferiore a sotto-problemi di taglia maggiore. Vantaggio di ciò è il fatto che ogni qual volta in problema di taglia minore è risolto, questi è memorizzato cosi da non dover essere eventualmente ricalcolarlo qualora servisse.

Esempio: Si consideri dunque il seguente prodotto, poiché l'ordine di moltiplicazione influenza il numero di operazioni totali, ci si chiede quale ordine convenga scegliere.

$$M = M_1 \times M_2 \times M_3 \times M_4$$

Rispettivamente di dimensioni $[10 \times 20], [20 \times 50], [50 \times 1], [1 \times 100]$.
É ovvio che valutare tutti i possibili prodotti sia sconveniente, specie al crescere di n. Fortunatamente la programmazione dinamica fornisce un'algoritmo con complessità $\mathcal{O}\left(n^3\right)$, la cui implementazione algoritmica è riportata al termine della pagina.
Sia m_{ij} il costo minimo per calcolare $M_i \times \ldots \times M_j, 1 \leq i \leq j \leq n$. Segue

$$m_{ij} = \begin{cases} 0, & \text{se } i = j \\ \min\limits_{i \leq k \leq j} (m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j), & \text{se } j > i \end{cases}$$

ove m_{ik} è il costo minimo per calcolare $M' = M_i \times \ldots \times M_k$, $m_{k+1,j}$ è il costo minimo per valutare $M'' = M_{k+1} \times \ldots \times M_j$, infine $r_{i-1} r_k r_j$ è il costo per valutare $M' \times M''$.

```
begin
    for i ← 1 to n do m_ii ← i
    for l ← 1 to n − 1 do
        for i ← 1 to n − 1 do
            begin
                j ← i + 1
                m_ij ← min i ≤ k ≤ j(m_ik + m_{k+1,j} + r_{i−1} r_k r_j)
            end
    write m_im
end
```

– 3.2.4 – Tecniche Greedy.

L'utilizzo di tecniche Greedy è opportuno per quei problemi d'ottimo, in cui scegliere ad ogni passo l'opzione migliore, comporta una soluzione ottimale nel complesso.

Problema legato alle tecniche Greedy; non è tanto la ricerca dell'algoritmo, quanto più dimostrare che l'algoritmo scelto produca effettivamente la soluzione ottima. In genere, per provare che l'algoritmo scelto sia quello ottimo, si procede con un ragionamento per assurdo, sulla base di quanto segue.

Siano S_g e S_O, rispettivamente, la successione di scelte Greedy e quella ottima. Sia per assurdo $S_g \neq S_O$. Da tale assunzione deve esistere almeno una scelta per cui le due soluzioni differiscano. Si considera la prima di tali scelte e si procede con definire una nuova successione di scelte S'_O, ottenuta da S_O sostituendo la scelta non Greedy con quella Greedy. Si verificano i seguenti casi:

1. risulta che S'_O sia meno ottimizzata rispetto S_O. Da ciò segue, poiché le scelte Greedy per definizione sono da effettuare passo dopo passo, segue che effettivamente $S_O \neq S_g$.
2. risulta che S'_O sia ugualmente o meglio ottimizzata di S_O. Da cui applicare la scelta Greedy è conveniente.

Se iterando per ciascuna delle scelte che differiscono tra S_g e S_O, non si rientra nel primo caso, S_g è ottima.

– 4 – Problema e algoritmi di ordinamento.

Il problema dell'ordinamento (o sorting), può essere formulato come segue: dato un insieme di elementi $a_1, a_2, \ldots, a_n$, sui quali è posta una relazione d'ordine totale, si è interessati ad una permutazione π, tali che $a_{\pi(i)} \leq a_{\pi(i+1)}, \forall i < n$.

– 4.1 – Proprietà degli algoritmi di ordinamento.

Gli algoritmi possono essere suddivisi in quattro categorie.

1. Incrementali: ogni passo incrementa di uno la sotto-sequenza ordinata.
2. In loco: la sequenza è ordinata sull'array di input, con l'aiuto di un array di supporto.
3. Adattivi: l'efficienza varia col variare dell'input.
4. Stabili: gli elementi duplicati sono posti in fondo alla sequenza ordinata, nell'ordine di disposizione originale.

– 4.2 – Algoritmi quadratici.

In questa sezione si analizzeranno alcuni algoritmi di ordinamento la cui complessità è $\mathcal{O}(n^2)$.

– 4.2.1 – Selection sort.

Ricevuto l'input, l'algoritmo procede con l'estrarre all'i-esimo passo il minore degli n - i elementi rimasti. Si osserva dunque che la complessità dell'algoritmo è dato dalla seguente sommatoria.

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \implies \mathcal{O}(n^2)$$

Di seguito si riporta la struttura algoritmica del selection sort.

```
algorithm SelectionSort(Array A)
begin
    for k ← 1 to n − 1 do
        m ← k

        for j ← k + 1 to n do
            if (A[j] < A[m]) then m ← j
        switch A[m] and A[k]
end
```

Figura 4.1: Implementazione SelectionSort.

– 4.2.2 – Insertion sort.

Ricevuto l'input, all'i-esimo passo, l'i-esimo elemento e ordinato rispetto gli i - 1, elementi precedentemente ordinati. Si osserva pertanto che la complessità dell'algoritmo è dato dalla seguente sommatoria.

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \implies \mathcal{O}\left(n^2\right)$$

Di seguito si riporta la struttura algoritmica del selection sort.

```
algorithm InsertionSort(Array A)
begin
    for k ← 1 to n − 1 do
        x ← A[k]

        for j ← 1 to k + 1 do
            if (A[j] > x) then break

        if (j < k + 1) then
            for (l = k) down to j do
            A[l + 1] ← A[l]
            A[j] ← x
end
```

Figura 4.2: Implementazione InsertionSort

– 4.2.3 – Bubble sort.

Ricevuto l'input, all'i-esimo passo, si confronta un coppia adiacente di elementi, se non sono ordinati si procede a scambiarli. La complessità dell'algoritmo è dato dalla seguente sommatoria.

$$\sum_{i=1}^{n-1} n - 1 = (n-1)^2 \implies \mathcal{O}\left(n^2\right)$$

Di seguito si riporta la struttura algoritmica del selection sort.

```
algorithm BubbleSort(array A)
begin
    for i ← 1 to n − 1 do
        for j ← 2 to n − 1 do
        if (A[j −1 ] < A[j]) then
            swap A[j − 1] and A[j]
end
```

Figura 4.3: Implementazione BubbleSort.

– 4.3 – Ordinamenti ottimi.

In questa sezione si discuteranno gli algoritmi di ordinamento, che a meno di casi particolari, risultano essere pià efficenti. Si tratta di algoritmi la cui complessità è $\mathcal{O}(n \log n)$.

– 4.3.1 – Merge sort.

Ricevuto l'input, il merge sort procede applicando una strategia divide-and-conquer. Si fa in modo che un l'ordinamento di n elementi, è ridotto ricorsivamente, all'ordinamento di due sotto-problemi di taglia n/2. Giunti a problemi di taglia unitaria, questi vengono a risolti ordinando coppie distinte di elementi, ottenendo problemi di taglia 2, e cosi via.

Si osserva dunque che il costo dell'algoritmo è dato dalla seguente espressione ricorsiva.

$$C(n) = \begin{cases} 1, & \text{se } n = 1 \\ 2C(n/2) + \mathcal{O}(n), & \text{se } n > 1 \end{cases}$$

Applicando il *Teorema Master*, si osserva che si rientra nel secondo caso, dunque il costo complessivo è $\mathcal{O}(n \log n)$. Di seguito si riporta l'implementazione algoritmica del Merge sort.

```
procedure Merge(Array A, integer: right, middle, left)
    k ← right, i ← right, j ← middle + 1;
    while(i ≤ middle and j ≤ left)
        if(A[i] ≤ A[j]) then
            X[k] ← A[i];
            increase i and k
        else
            X[k] ← A[j];
            increase j and k

    if (i ≤ middle) then
        copy A[i : middle] at the end of X
    else
        copy A[middle + 1 : left] at the end of X
    copy X in A
```

```
algorithm MergeSort(Array A, integer right, left)
    if(right > left) break
    middle ← (right + left) / 2;
    MergeSort(A, right, middle);
    MergeSort(A, middle + 1, left);
    Merge(A, right, middle, left)
```

Figura 4.4: Procedura merge ed implementazione MergeSort.

– 4.3.2 – Heap sort.

L'heap sort è un algoritmo con approccio incrementale, che fa uso di una struttura dati per la ricerca del minimo: l'*heap*. Considerando l'heap in se, questi è implementato come albero con le seguenti proprietà.

1. Fino al penultimo livello, l'albero è completo.
2. Ciascun nodo contiene un elemento.
3. Il valore del nodo padre è maggiore o uguale a quello dei nodi figlio.

Per il modo in cui è implementato, l'heap ha le seguenti caratteristiche.

- Il massimo risiede nella radice.
- L'altezza è logaritmica rispetto il numero dei nodi.

Parlando dell'implementazione vera e propria, questa è effettuata grazie alle due procedure in *Figura 4.5*

```
procedure fixHeap(node v, heap H)
    begin
        if (v is a leaf) then return
        else
            let u be the son of v, with the highest value
            if (value(v) < value(u)) then
            swap v and u;
            fixHeap(u, H)
    end
```

```
procedure heapify(heap H)
    begin
        if (H is empty) the return;
        else
        heapify(left subtree of H);
        heapify(right subtree of H);
        fixHeap(root(H), H)
    end
```

Figura 4.5: Procedure per l'implementazione degli heap.

Analizzando la complessità delle due procedure: si osserva che fixHeap segue al più un cammino lungo quanto l'altezza dell'albero, dunque è $\mathcal{O}(\log n)$; circa la procedura heapify, questa è espressa dalla seguente relazione di ricorrenza

$$T(n) = 2T(n/2) + \mathcal{O}(\log n)$$

che per il *Master theorem* risulta essere $\mathcal{O}(n)$.

Circa l'heap sort in se: data una sequenza di elementi da ordinare, questi sono posti su un heap, ad ogni passo da quest'ultimo è estratto il massimo; estrazione alla quale segue una chiamata a fixHeap. Poiché l'estrazione del massimo richiede $\mathcal{O}(\log n)$, e dato che ciò avviene n volte, il costo totale è $\mathcal{O}(n \log n)$.

– 4.3.3 – Quick sort.

Il quick sort è un algoritmo con complessità $\mathcal{O}(n^2)$, quindi la classificazione come algoritmo ottimo appare erronea. Nonostante non rientri nella definizione classica di algoritmi ottimi, il quick sort si dimostra essere superiore al merge e all'heap sort nel caso medio.

Parlando dell'quick sort in se: questi si basa sul principio di divide-and-conquer, secondo la seguente "struttura".

- Seleziona un elemento pivot x della sequenza. Separa gli elementi della sequenza in elementi minori o uguali e elementi maggiori di x.
- Procede ricorsivamente sulle due sotto sequenze.
- Restituisce la concatenazione delle due sotto-sequenze ordinate.

Nota: Di grande importanza è la scelta del pivot, difatti l'efficienza dell'algoritmo dipende da ciò.

In *Figura 4.6* è riportata un'implementazione del quick sort, ne esiste un'altra che procede con effettuando scambi in loco, sfruttando la procedura partition in *Figura 4.7*.

```
algorithm QuickSort(Array A)
    begin
        choose an element x in A;
        procede partitioning A respect to x
        let A_1 = {y ∈ A : y ≤ x}, A_2 = set y ∈ A y > x
        if (norm A_1 > 1) then quickSort(A_1)
        if (norm A_2 > 1) then quickSort(A_2)
        concatenate A_1 and A_2, copy the result on a
    end
```

Figura 4.6: Implementazione QuickSort.

```
procedure partition(Array A, integer: left, right) → integer
    begin
        x ← A[left], inf ← left, sup ← right + 1
        do
            do {inf ← inf + 1} while (A[inf] < x)
            do {sup ← sup − 1} while (A[sup] > x)
            if (inf < sup) swap A[inf] and A[sup];
            inf ← inf − 1, sup ← sup − 1
        while (inf < sup)
        swap A[left] and A[sup]
        return sup
    end
```

Figura 4.7: Procedura partition, per l'implementazione di QuickSort in loco.

– 4.4 – Ordinamenti lineari.

Nel caso di sequenze da ordinare con particolari proprietà, esistono algoritmi che restituiscono la sequenza corretta in tempo proporzionale alla taglia della sequenza. Questi algoritmi sono detti di *ordinamento lineare*.

– 4.4.1 – Integer sorting.

Data una sequenza da ordinare di n interi tutti compresi tra $[1,k]$, questa è ordinabile in tempo lineare tramite diversi algoritmi, quali *counting sort, bucket sort*.

Partendo con il counting sort: sia X la sequenza da ordinare, e Y un array di supporto di taglia k. Ad ogni passo, se $x \in X$ allora $Y[x] = Y[x]+1$. Completata la lettura di X, procede a copiare x in X, tante volte quante indicate da $Y[x]$. Da quanto descritto e dall'implementazione algoritmica di *Figura 4.8*, si osserva che il counting sort ha complessità $\mathcal{O}(n+k)$.

```
algorithm countingSort(Array A, integer k)
    begin
        let COUNT be an array of k elements
        for i ← 1 to n do COUNT[i] ← 0
        for j ← 1 to n do COUNT[A[j]] ← COUNT[A[j]] + 1
        j ← 1
        for i ← 1 to k do
            while (COUNT[i] > 0) do
            A[j] ← i;
            j ← j + 1;
            COUNT[i] ← COUNT[i] − 1
    end
```

Figura 4.8: Implementazione CountingSort.

Considerando ora il bucket sort: si supponga di voler ordinare n record con chiave intera appartenenti a $[1,k]$, prendendo spunto dal counting sort, sia Y un array tale che $Y[i]$ sia una lista di elementi con chiave i. Da ciò segue che l'ordinamento degli n record, è dato dalla concatenazione delle liste in Y. É facile osservare, proprio per la similarità al counting sort, che il bucket sort ha complessità $\mathcal{O}(n+k)$.

– 5 – Strutture dati per la manipolazione di insiemi.

Capita spesso che quando si ha a che fare con un problema, questi è descrivibile in termini di strutture matematiche elementari, ad esempio gli insiemi. Per tale ragione in questa sezione si analizzeranno le operazioni elementari sugli insiemi, e sulle strutture adatte alla loro implementazione.

Circa le operazioni sugli insiemi, di interesse al corso sono le seguenti.

1. MEMBER(a, S): verifica se a appartiene o meno ad S.
2. INSERT(a, S): aggiunge a ad S, se a non è già presente.
3. DELETE(a, S): rimuove a da S, se a è presente.
4. UNION(S, T, U): gli insiemi S e T sono uniti, e la loro unione è assegnata ad U.

– 5.1 – Hashing.

Sia S un insieme di elementi: come gestire efficientemente i cambiamenti dello stesso? Se S è formato da elementi che possono appartenere ad un universo molto ampio, allora questo può essere gestito tramite l'*hashing*. Sebbene esistano variazioni all'hashing, nel seguito si considera l'idea alla base. Partendo col definire la funzione di hashing h, questa è definita dall'insieme S ad un sottoinsieme dei naturali. Cioè

$$h : S \to \mathbb{M} \subseteq \mathbb{N}$$

Sia A un array di taglia m, i cui elementi sono puntatori a lista di elementi di S, ossia: $A[i] = \{a \in S : h(a) = i\}$. Si assuma che $\forall a \in S, h(a) \in \{0, 1, \ldots, m-1\}$, e che ciò richieda tempo costante.

Considerando le operazioni: l'esecuzione di INSERT computa $h(a)$, verificando nella lista $A[h(a)]$ l'assenza di a, nel cui caso lo accoda; l'esecuzione di DELETE procede inversamente, in fine MEMBER scannerizza $A[h(a)]$ e fornisce una risposta.

Circa la complessità: se si assume che la funzione di hashing abbia la stessa probabilità di assegnare un valore compreso tra 0 e $m-1$, e al più $n = m$, risulta che il tempo richiesto è $\mathcal{O}(n)$.
Risulta ovvio che procedere così non è sempre possibile, specie perché n non è sempre noto. Segue pertanto, che necessario dover possibilmente costruire più tabelle di hash $T_0, T_1, \ldots$. Si procede allora creando una prima tabella di hash T_0 di una taglia m adeguata, si continua ad utilizzate T_0 finché gli elementi non superano m. Raggiunto tale limite, si applica una nuova funzione di hash per copiare gli elementi di T_0 in una nuova tabella T_1 di taglia $2m$. Si continua ripetendo lo stesso processo, ogni volta che si raggiunge la dimensione massima.

– 5.2 – Optimal BST.

Sia $a_1, a_2, \ldots, a_n$ una sequenza di elementi di un insieme S. Siano inoltre p_i la probabilità di ricercare l'i-esimo elemento, q_0 la probabilità di ricercare $a < a_1$, q_i la probabilità di ricercare $a_i < a < a_{i+1}$ e q_n la probabilità di ricercare l'ultimo elemento.

Per definire il costo di un *BST*, si osserva quanto segue:

1. l'elemento a ricercato è l'etichetta di un qualche vertice v_i, da cui per ricercare a si sono visitati DEPTH(v) + 1 vertici;
2. l'elemento $a \notin S$.

Da ciò, il costo di un BST risulta essere

$$\sum_{i=1}^{n} p_i(DEPTH(a_i)+1) + \sum_{i=0}^{n} a_i DEPTH(i)$$

Sorge ora una domanda: come trovare il BST di costo minimo? Si osserva che un approccio divide-and-conquer non è applicabile, poiché anche se si potrebbe procedere costruendo i sotto-alberi, rimane il problema di costruire la radice. Cio suggerisce però che si considerano 2n sotto-problemi, due per ogni radice possibile, segue che una soluzione prevede la programmazione dinamica.

Siano $T_{ij}, 1 \leq i \leq j \leq n$ gli alberi di costo minimo per gli elementi $a_{i+1}, \ldots, a_j$. Sia c_{ij} il costo di T_{ij}, e r_{ij} la sua radice. Sia infine w_{ij} il peso di T_{ij}, definito come $q_i + (p_i + q_{i+1}) + \ldots + (p_i + q_j)$.

Si può quindi considerare ogni T_{ij} come l'albero con radice a_k e sotto-alberi $T_{i,k-1}$ e $T_{k,j}$. Sia per convenzione T_{ii} l'albero vuoto, per cui $c_{ii} = 0$.
Considerando $C_{ij}, i < j$ si ha

$$\begin{aligned} c_{ij} &= w_{ik} + p_k + w_{ij} + c_{i,k-1} + c_{kj} \\ &= w_{ij} + c_{i,k-1} + c_{kj} \end{aligned}$$

Da cui T_{ij} ottimo ha costo $c_{ij} = MIN(w_{ij} + c_{i,k-1} + c_{kj})$.

– 5.3 – Problema della Union-Find con struttura ad albero.

Il problema della union-find può essere semplicemente descritto come segue: dati degli insiemi, come gestire efficientemente operazioni di unione e ricerca?

Nota: Esistono soluzioni che sfruttano gli array, ma per quanto efficienti risultano lente se comparate a quelle di seguito proposte.

Ogni insieme A può essere rappresentato come un albero non orientato, i cui vertici sono elementi di A.

Considerando l'istruzione UNION, se T_A e T_B rappresentato, rispettivamente, gli insiemi A e B, segue che questa richiede tempo costante. Si osserva infatti che basta aggiungere un puntatore tra i due alberi. Circa l'istruzione FIND, questa è effettuata localizzando il vertice ricercato e percorrendo il cammino dallo stesso, verso la radice. Risulta ovvio che questo richiede al più $\mathcal{O}(n)$[7] operazioni.

Si osserva che il costo dell'istruzione FIND può essere ridotto, se l'albero è bilanciato. Come ottenere tale bilanciamento? Sfruttando il seguente lemma.

Lemma 5.1.
Se nell'eseguire un'istruzione di UNION, si fa in modo che l'albero di altezza minore punti a quello di altezza maggiore, non esisterà albero nella foresta[8] *di altezza h, a meno che questi non abbia almeno 2^h vertici.*

Dimostrazione: La dimostrazione procede per induzione su h.

Base: Sia $h = 0$, l'ipotesi è ovviamente vera, poiché ogni albero ha almeno un nodo.

Induzione: Si supponga che quanto detto sia vero per ogni valore minore di $h \geq 1$. Sia T un albero di altezza h col numero minore di foglie. Allora T è ottenuto come UNION di T_1 e T_2, con $\|T_1\| = h-1$ e T_1 ha al più lo stesso numero di vertici di T_2. Per ipotesi induttiva T_1 e T_2 hanno almeno 2^{h-1} vertici, per cui T ne ha almeno 2^h.

□

Una modifica all'algoritmo prevede la *path compression*. Si è visto che il costo maggiore dell'intero algoritmo è dato dall'istruzione FIND, pertanto ciò che è di interesse è ridurre tale costo. Si è detto che un'istruzione FIND(i) comporta ol dover percorrere un cammino del tipo, $i, v_1, \ldots, v_k, r$. Se si rende ciascuno di questi vertici figlio di r, si dimostra che le FIND successive risultano più efficienti.

[7] Questo perché potrebbe capitare che l'albero risulti sbilanciato.
[8] Con foresta in questo caso ci si riferisce ad altri possibili insiemi su cui si deve eseguire una UNION

– 5.4 – Alberi bilanciati: alberi 2-3.

Come anticipato nella precedente sezione, se un albero è bilanciato, le operazioni che lo coinvolgono risultano più efficienti.

Si è finora visto una situazione statica cioè: gli insiemi su cui svolgere le operazioni, era già noto. Da adesso si procede col considerare il caso dinamico.
Per far ciò si considererà come struttura dati un albero bilanciato: alberi la cui altezza è logaritmica rispetto al numero di nodi.

Risulta banale che la costruzione iniziale di un albero bilanciato sia semplice, il problema sorge quando su di esso si eseguono operazioni di INSERT e DELETE.

Nota: Esistono varie tecniche per mantenere un albero bilanciato, di maggiore importanza sono gli *AVL tree* e gli *alberi 2-3*. Saranno considerati solo quest'ultimi.

Definizione: Un albero 2-3 è un albero in cui ciascun vertice non foglia, ha 2 o 3 figli. Inoltre, ogni cammino dalla radice a una qualsiasi foglia deve avere la medesima lunghezza.

Lemma 5.2.
Sia T un albero 2-3 di altezza h. Allora T ha un numero di vertici compreso tra $2^{h+1}-1$ e $3^{h+1}-1$, il numero di foglie è invece compreso tra 2^h e 3^h.

Nelle successive sezioni si dimostrerà come implementare *dizionari, code con priorità, mergeable heap* e *code concatenabili*, per mezzo di alberi 2-3.

– 5.5 – Dizionari e code con priorità.

Operazioni elementari comuni ad un dizionari e a una coda con priorità sono: INSERT, DELETE; operazioni specifiche risultano essere rispettivamente: MEMBER per i dizionari e MIN per le code con priorità.

Considerando l'istruzione INSERT (in *Figura 5.1* se ne riporta l'implementazione algoritmica): sia a un elemento da inserire. Per prima cosa si ricerca la posizione per una foglia l che conterrà a. Assunto che l'albero abbia più di un elemento, la ricerca prosegue fin quando si trova un vertice f con due o tre figli. Se f ha solo due figli, si supponga l_1 e l_2, si fa in modo che l diventi figlio di f.
Definendo $E[x]$ il valore contenuto nel vertice x, $L[x]$ il massimo valore contenuto nel sotto-albero con radice il figlio sinistro di x, ed $M[x]$ il massimo valore contenuto nel sotto-albero con radice il figlio centrale di x; si presentano tre casi.

1. $a < E[l_1]$, allora si rende l figlio sinistro di f. Inoltre, $L[f] = a$ e $M[f] = l_1$.
2. $E[l_1] < a < E[l_2]$, si rende l figlio centrale di f. Inoltre $M[f] = a$.
3. $E[l_2] < a$, si rende l il terzo figlio di f.

Supponendo che f abbia tre figli, si supponga l_1, l_2, l_3, segue che dover rendere l figlio di f, comporta che questi abbia quattro figli, violando così la proprietà di albero 2-3. Sia g un nuovo vertice, al fine di mantenere la proprietà di albero 2-3, si rendono i due figli più a destra di f figli di g. In fine, si rendono f e g figli dello stesso padre, eventualmente ripetendo ricorsivamente la procedura.
Caso particolare è quello della radice. Se la radice ha quattro figli, si crea una nuova radice con due figli, ciascuno dei quali a loro volta avrà due dei figli della precedente radice.

```
procedure AddSon(vertex v)
    begin
        create a new vertex v';
        make the two rightest sons of v, left and right sons of v'
        if (v has no father) then
            create a new root r;
            make v and v', respectively, left and right sons of r
        else
            let f be father of v;
            make v' right brother of v
        if (f has four sons) then AddSon(f)
```

Figura 5.1: Procedura AddSon, per la creazione di alberi 2-3

Teorema 5.1.
La procedura di Figura 5.1*, inserisce un nuovo elemento in* $\mathcal{O}(\log n)$*. In più mantiene l'ordine originale e conserva la struttura di albero 2-3.*

Analizzando ora l'operazione di DELETE, sia x il vertice da eliminare, si osservano le seguenti possibilità.

1. Il vertice x è la radice: si procede eliminandola[9].
2. Il vertice x è figlio di un vertice con tre figli: lo si elimina.
3. Il vertice x è figlio di un vertice f con due figli, si hanno i seguenti casi.
   (a) f è la radice: si eliminano sia x che la radice; l'ultimo figlio è reso la nuova radice.
   (b) Il padre non è la radice. Si supponga che f abbia una fratello g con due figli, si rende il vertice da non eliminare figlio di g. Procedendo con l'eliminare f ed x. Se g ha tre figli, si rende il figlio più a destra figlio sinistro di f; si procede con eliminare x.

Considerando in fine l'operazione di MIN: si osserva che per costruzione, il minimo risiede nella foglia più a sinistra, richiedendo dunque tempo $\mathcal{O}(\log n)$.

In conclusione, gli alberi due tre permettono di implementare dizionari e code con priorità in $\mathcal{O}(n \log n)$.

[9] Caso banale: si ha un'unico vertice.

– 5.6 – Mergeable heaps.

Un mergeable heap è una struttura dati che permette almeno le istruzioni di INSERT, DELETE, MIN, UNION, tutte con costo $\mathcal{O}(\log n)$.
Sia T un albero 2-3 che rappresenta elementi di un insieme S. Ogni $x \in S$ appare come etichetta di una qualche foglia di T: foglie che su cui non è posto nessun ordine. Per ogni vertice interno a T, sia SMALLEST[v] l'elemento con valore minimo nel sotto-albero di radice v.

L'operazione di MIN risulta $order \log n$ poiché basta seguire il cammino indicato dai vertici con SMALLEST minore. Circa DELETE: se gli elementi di S possono essere indicati dagli interi $1, 2, \ldots, n$, è possibile allora indicizzare le foglie direttamente. Se ciò non è possibile, è necessario far uso di una struttura di supporto, che contenga puntatori alle foglie di T.

Siano S_1 e S_2 due insiemi di cui T_1 e T_2 sono la rappresentazione tramite alberi 2-3. L'esecuzione di un'istruzione UNION è effettuata con un chiamata alla procedura Implant, di *Figura 5.2*.

```
procedure Implant(2-3 tree: T1, T2)
    begin
        if (HIGHT(T1) == HIGHT(T2)) then
            create a new root, make T1 and T2, respectively,
            left and right sons of said root
        else // assuming HIGHT(T1) > HIGHT(T2)
            let v be the vertex on the rightest path of T1,
            such that HIGHT(v) == HIGHT(T1) - HIGHT(T2),
            make ROOT(T2) right brother of v
        if (f has four sons) then AddSon(f)
    end
```

Figura 5.2: Procedura Implant, per l'unione di alberi 2-3.

Analizzando la procedura: sia h_1 l'altezza di T_1 e h_2 l'altezza di T_2. Si ricerca nel cammino più a destra di T_1 una foglia v, tale che questa abbia altezza h_2 e si rende la radice di T_2 fratello destro di v. Si hanno due possibilità a seguito della procedura, posto f padre di v, quali

1. f ha tre figli, non sorgono problemi.
2. f ha quattro figli, è necessario ripristinare la proprietà di albero 2-3, si effettua una chiamata ad AddSon.

– 5.7 – Code concatenabili.

Un coda concatenabile, è una struttura dati che permette operazioni di INSERT, DELETE, MEMBER, CONCATENATE, SPLIT.

Osservazione: Le operazioni di INSERT, DELETE e MEMBER, risultano analoghe a quelle dei dizionari.

Analizzando le operazioni di CONCATENATE e SPLIT, entrambe richiedono tempo $\mathcal{O}(\log n)$. Considerando per prima CONCATENATE: siano S_1 e S_2 due sequenze di elementi, tali che ogni elemento in S_1 sia minore di ogni elemento in S_2; allora CONCATENATE restituisce $S_1 S_2$. Supponendo che T_1 e T_2 siano due alberi 2-3 che rappresentano i due insiemi, l'operazione di CONCATENATE si risolve con un chiamata a Implant.

Analizzando ora SPLIT: questa fa si che l'insieme venga suddiviso in due metà, una prima contenente gli elementi minori o uguali a un certo a, la seconda contenente gli elementi con valore maggiore.
Considerando l'implementazione con alberi 2-3, ciò è effettuato con la procedura DIVIDE. Questa fa si che ricevuti a un valore e T un albero 2-3, questa divide T in due sotto-alberi T_1 e T_2, rispettivamente, con elementi minori o uguali ad a, ed elementi maggiori di a.

```
procedure DIVI DE(a, T):
    begin
        on the path from ROOT[T] to the leaf labeled a remove all vertices
        except the leaf;
        while (there is more than one tree in the left forest) do
            begin
                let T' and T" be the two rightmost trees in the left forest;
                IMPLANT(T', T")
            end;
        while (there is more than one tree in the right forest) do
            begin
                let T' and T" be the two leftmost trees in the right forest:
            IMPLANT(T', T")
            end
    end
```

Generalizzando dato T un albero 2-3 si segue il cammino dalla radice ad a, Tale cammino divide l'albero in una foresta di sotto-alberi, le cui radici non esse stesse nel cammino. Con tale suddivisione i sotto-alberi nel cammino sinistro, uniti ad a, sono combinati componendo T_1. Analogo discorso per i sotto-alberi nel cammino destro.
Per un teorema che qui non si riporta, DIVIDE richiede $\mathcal{O}(HEIGHT(T))$, dunque $\mathcal{O}(\log n)$.

– 6 – Algoritmi su grafi.

Molti problemi scientifici possono essere formulati in termini di grafi, orientati e non. Si presentano in questa sezione diversi problemi legati ai grafi, problemi che come sarà dimostrato avranno una soluzione al più polinomiale.

– 6.1 – Spanning tree di costo minimo.

Definizione: Sia $G = (V, E)$ un grafo non orientato. Si definisce uno *Spanning-Tree* di G, un albero non orientato $S = (V, T)$, ottenuto a partire da G. Si definisce analogamente *Spanning Forest* di G l'insieme degli Spanning-Trees $\{(V_1, T_1), \ldots, (V_k, T_k)\}$, tali che ciascuna delle $V_i, i = 1, \ldots, k,$ sia una partizione di V e gli $T_i, i = 1, \ldots, k$ siano sottoinsiemi, possibilmente vuoti di E.

Posto che il costo di uno Spanning-Tree è dato dalla somma dei costi dei suoi archi, ciò che risulta interessante è la ricerca dello Spanning-Tree di costo minimo. Si riportano a seguito due lemmi; sui di essi sono basati gran parte degli algoritmi sugli Spanning-Tree .

Lemma 6.1.
Sia $G = (V, E)$ un grafo connesso (esiste un cammino tra ogni coppia di vertici) e non orientato, sia $S = (V, T)$ un suo Spanning-Tree. Allora

1. *per ogni $u, v \in V$ esiste un unico cammino da u a v in S;*
2. *se un'arco in $E \setminus T$ è aggiunto ad S, si viene ad avere un ciclo.*

Dimostrazione: Dimostrare il primo punto è banale: se vi fosse più di un cammino ne risulterebbe un ciclo. La dimostrazione del secondo punto è altrettanto banale, poiché vi deve necessariamente gia essere una cammino tra gli estremi dell'arco aggiunto. □

Lemma 6.2.
Sia $G = (V, E)$ un grafo connesso e non orientato, sia inoltre c una funzione di costo sugl'archi di G. Sia $\{(V_1, T_1), \ldots, (V_k, T_k)\}, k > 1$ Spanning Forest di G, con $T = \bigcup_{i=1}^{k} T_i$. Supposto $e = (v, w)$ un arco di costo minimo in $E \setminus T$, $v \in V_1$, $w \notin V_1$, allora esiste uno Spanning-Tree di G che include $T \cup \{e\}$, con costo equivalente a qualsiasi altro Spanning-Tree che includa T.

Dimostrazione: Si supponga per assurdo che $S' = (V, T')$ sia uno Spanning-Tree di G, tale che T' contenga T ma non e. Si supponga inoltre che S' sia lo Spanning-Tree con costo minore tra tutti gli altri Spanning-Tree di G che includono $T \cup \{e\}$. Per il *Lemma 6.1*, aggiungere e ad S crea un ciclo. Se ciò avviene esiste allora $e' = (v', w')$, $v' \in V$ e $w' \in V$ oltre e.

Sia S lo Spanning-Tree ottenuto aggiungendo e ad S' e rimuovendo e'. Per ipotesi $c(e) \leq c(e')$, quindi S non è più costoso di S', ma poiché S contiene sia T che e ciò contraddice la minimalità di S'. □

La costruzione dello Spanning-Tree minimo può essere effettuata con vari algoritmi, tra i quali quello di *Figura 6.1*.

```
algorithm Kruskal(graph G = (V, E))
    begin
        T ← ∅;
        VS ← ∅;
        construct a priority queue Q, containing all the edges in E
        for each vertex v ∈ V do add {v} to VS
        while ||VS|| > 1 do
            begin
                choose(u, v) in Q with of lowest cost;
                delete (u, v)
                if (v and w are in different sets W1 and W2) then
                    begin
                        replace W1 and W2 by their union in VS
                        add (u, v) to T
                    end
            end
    end
```

Figura 6.1: Algoritmo di Kruskal.

Si dimostra che la complessità risulta essere $\mathcal{O}(m \log n)$ con $m = \|E\|$ e $n = \|V\|$.

– 6.2 – Depth-first search.

Dato un grafo G effettuare una depth-first search, rappresenta il seguente processo.

1. Si sceglie un vertice x e lo si visita.
2. Si seleziona un vertice y discendente di x, e lo si visita.

Più in generale: supposto x il vertice visitato più di recente, la ricerca prosegue selezionando un arco inesplorato (x,y). Se y è stato precedentemente visitato, si ricerca un'altro arco. Viceversa, si visita y è si riprende la visita a partire da esso. Terminate le visite su tutti i cammini con radice y, si ritorna ad x. La DFS termina quando tutti i vertici risultano esplorati.

Nel caso di grafi non orientati, segue che la DFS suddivide gli archi in due insiemi T e B, rispettivamente, i *tree edges* e i *back edges*. Il sottografo (V,T) risulta essere una foresta non orientata detta *depth-first spanning forest*.

La visita in DFS è effettuata con il seguente algoritmo.

```
procedure SEARCH(vertex v)
    begin
        mark v ''old''
        for each vertex on L[v] do
        if (w is marked ''new'') then
            add (v, w) to T;
            SEARCH(w)
    end
```

```
algorithm DFS(graph G = (V, E))
begin
    T ← ∅
    for all v in V do mark v ''new''
    while (there exists v in v marked ''new'') do
    SEARCH(v)
end
```

Figura 6.2: Algoritmo per la visita in DFS.

– 6.3 – Biconnettività.

Si considera ora un'applicazione della DFS: la ricerca di componenti biconnesse di un grafo non orientato.

Definizione: Sia $G = (V, E)$ un grafo connesso e non orientato. Dicasi che un vertice a è punto di articolazione se esistono due vertici v e w, tali che ogni cammino tra i due passa per a.

Definizione: Un grafo G è detto *biconnesso* se per ogni tripla v, w, a, esiste un cammino tra v, w che non passa per a.

Si definisce ora una relazione di equivalenza su gli archi di un grafo G. Per tale relazione due archi e_1, e_2 sono in relazione se $e_1 = e_2$, oppure esiste un ciclo che li contiene entrambi. Per $1 \leq i \leq k$, sia V_i l'insieme dei vertici degl'archi in E_i. Ogni sottografo $G_i = (V_i, E_i)$ è detta *componente biconnessa* di G.

Lemma 6.3.
Per $1 \leq \imath \leq k$, sia $G_i = (V_i, E_i)$ una componente biconnessa di un grafo G. Allora

1. *G_i è biconnesso per ogni $i, 1 \leq i \leq k$;*
2. *per ogni $i \neq j, \left\| V_i \cap V_j \right\| \leq 1$;*
3. *un vertice a è punto di articolazione per G se e solo se $a \in V_i \cap V_j, i \neq j$.*

Dimostrazione:

1. Si suppongano $v, w, a \in V_i$ tali che ogni cammino da v a w passi per a. Dunque $(v, w) \notin E_i \implies \exists (v, v'), (w, w') \in E_i$ e vi è un ciclo che li contiene entrambi. Per definizione stessa di componente biconnessa ogni arco e vertice nel ciclo, appartengono a E_i e V_i rispettivamente. Esistono dunque due cammini uno solo dei quali passa per a, il che è una contraddizione.

2. Si suppongano v e w due vertici distinti appartenenti a $V_i \cap V_j$. Esiste quindi un ciclo C_1 in G_i che li contiene entrambi, e un ciclo C_2 in G_j che li contiene entrambi. Ora, poiché $E_i \cap E_j = \emptyset$, necessariamente $C_1 \cap C_2 = \emptyset$. Ma si può costruire un ciclo che contiene sia vertici di C_1 che C_2, esiste quindi un vertice in E_i equivalente a un vertice in E_j. Ma se ciò fosse vero E_i ed E_j non sarebbero classi di equivalenza, come si è invece supposto.

3. Si suppongano a punto di articolazione per G. Esistono quindi due vertici v, w tali che v, w, a siano distinti, ed ogni cammino tra v e w contiene a. Siano (x, a) e (y, a) due archi in un cammino tra v e w incidenti[10] su a. Se esiste un ciclo contenente questi archi, esiste pertanto un cammino tra v e w che non passa per a. Dunque (x, a) e (y, a) sono in due componenti biconnesse diverse, e a è punto di intersezione dei loro insiemi. Il viceversa è banale.

□

[10] Due archi sono incidenti se contengono almeno un vertice comune.

Lemma 6.4.
Sia $G = (V, E)$ un grafo connesso e non orientato, sia $S = (V, T)$ un suo Spanning-Tree. Un vertice a è punto di articolazione per G, se e solo se

1. *a è radice di S, ed ha più di un figlio; oppure*
2. *a non è la radice, per qualche figlio s di a non vi è un back-edge tra un discendente di s e un'antenato di a.*

Dimostrazione:

1. Risulta banale.
2. Sia f padre di a. Dalla definizione stessa di back-edge, un discendente v di s va ad un antenato di v. Ciò implica che v vada in a o ad un discendente di s. Ma pertanto ogni cammino da s ad f passa per a, quindi segue a punto di articolazione.

   Viceversa, sia supposto a punto di articolazione, ma non radice. Siano x e y due vertici distinti oltre a, tali che ogni cammino tra i due passi per a. Almeno uno dei due è discendente proprio di a, si supponga x. Sia s figlio di a tale che x sia discendente di s. Allora o non vi è un back-edge tra v discendente di s e un'antenato w di a, nel cui caso l'ipotesi è immediatamente vera, o esiste tale back-edge. In quest'ultimo caso seguono due possibilità.

   (a) Si supponga che y non discenda da a. Esiste a questo punto un cammino $x \to v \to w \to y$, che eviti a, ma ciò è una contraddizione.

   (b) Si supponga che y sia discendente di a. Sicuramente y non è discendente di s. Sia allora s' figlio di a tale che y discenda da s'. In questo caso non vi è un back-edge tra un discendente v' di s' e un'antenato proprio w' di a , da cui l'ipotesi è immediatamente vera, oppure esiste un cammino $x \to v \to w \to w' \to y$ che evita a, ma ciò è una contraddizione.

□

– 6.4 – DFS di grafi orientati.

Nel caso in cui il grafo su cui si deve effettuare la DFS sia un grafo orientato, se si applica opportune modifiche, si può sfruttare l'algoritmo sin ora visto. Nel caso di grafi orientati infatti, la DFS produce una Spanning-Forest i cui archi sono di quattro tipi.

- Tree edge: collegano vertici ancora non visitati.
- Back edge: collegano discendente e antenato di un vertice.
- Forward edge: collegano l'antenato e il discendente di un vertice, se questi non è Tree edge.
- Cross edge: archi che collegano vertici tra i quali non vie relazione di discendenza.

Lemma 6.5.
Se (v,w) *è un Cross edge, allora* $DFNUMBER[v] > DFNUMBER[w]$.

– 6.5 – Connettività forte.

Definizione: Sia $G = (V, E)$ un grafo diretto. Siano $V_i, 1 \leq i \leq k$ partizioni di V, tali che presi due vertici v, w, i due sono in relazione se e solo se esiste un cammino dall'uno all'altro e viceversa. Siano $E_i, 1 \leq i \leq k$ le partizioni di E che collegano vertici di V_i.
Il sottografo $G_i = (V_i, E_i)$ è detta *componente fortemente connessa* di G. Un grafo è detto fortemente connesso se e solo se esiste un'unica componente fortemente connessa al suo interno.

Si analizza ora la relazione tra componenti fortemente connesse e DFS.

Lemma 6.6.
Sia $G_i = (V_i, E_i)$ una componente fortemente connessa di un certo grafo orientato G. Sia $S = (V, T)$ Spanning-Forest per G. Allora i vertici di G_i e gli archi di $E_i \cap T$ formano un albero.

Dimostrazione: Siano v, w vertici in V_i. Senza perdita di generalità, si assuma $v < w$. Poiché v, w sono nella stessa componente fortemente connessa, esiste un cammino p tra i due. Sia x il vertice di p minore. Giunti ad un discendente di x tramite p, accade che non si può uscire dal sotto-albero di discendenti di x. Dunque w è discendente di x. Poiché ciascun vertice è numerato in pre-ordine, ogni vertice compreso tra x, w è discendente di x. Segue che poiché $x \leq v < w, v$ è discendente di x.

Sia r antenato comune dei vertici in G_i, col DFNUMBER minore. Se v è in G_i, ogni vertice nel cammino tra r, v è in G_i. □

Lemma 6.7.
Per ogni $1 \leq i \leq j, G_i$ si compone di tutti i vertici discendenti da r_i, che non sono in $G_1, \ldots, G_{i-1}$.

Dimostrazione: La radice r_j , per $j > i$ non può discendere da r_i. Infatti la chiamata a $SEARCH(r_i)$ termina prima di quella a $SEARCH(r_j)$. □

Al fine di semplificare la ricerca delle radici, sia definita la seguente funzione.

$LOWLINK[v] = MIN(\{v\} \cup \{w :$ esista un cross edge, o un back edge tra v e w.
Inoltre la radice contente w è antenato di $v\}$)

Lemma 6.8.
Sia G un grafo orientato. Un vertice v è radice di una componente fortemente connessa se e solo se $LOWLINK[v] = v$.

Dimostrazione: Sia $LOWLINK[v] = v$. Se v non è la radice di una componente fortemente connessa contente v stesso, allora un antenato proprio r di v lo è. Esiste quindi un cammino p tra v, r. Si considera il primo arco in p da un discendente di v ad un certo vertice w, che non discenda da v. Segue che (v, w) è un back edge o un cross edge. comune $w < v$. In quanto, r è antenato di v, segue che esiste un cammino da v a w. Si conclude che w, r sono nella stessa componente fortemente connessa; se però ciò fosse vero $LOWLINK[v] \leq w < v$: una contraddizione.

Sia viceversa v radice di una componente fortemente connessa di un grafo G. Per definizione stessa $LOWLINK[v] \leq v$. Si supponga $LOWLINK[v] < v$. Segue che esistono w, r tali che

1. w sia raggiunto da un discendente di v tramite back edge o cross edge;
2. r è radice della componente fortemente connessa contente w;
3. r è antenato di v;
4. $w < v$.

Per il secondo punto $r \leq w$. Per il quarto $r < v$, implicando, per il punto tre, che r sia antenato proprio di v. Segue però che, poiché in G esiste un cammino da v a r passante per w, quest'ultimi sono nella stessa componente fortemente connessa; concludendo pertanto che v non sia radice: una contraddizione. □

La ricerca delle componenti fortemente connesse è riportata di seguito.

```
procedure SEARCH(v):
    begin
        ... // look at the first four instruction of the DFS algorithm
        push v on STACK;
        for each vertex w on L [v] do
            if (w is marked "new") then
                begin
                    SEARCHC(w);
                    LOWLINK[v] ← MIN(LOWLINK[v], LOWLINK[w])
                end
            else if (DFNUMBER[w] < DFNUMBER[v] and w is on STACK) then
                LOWLINK[v] ← MIN(DFNUMBER[w],LOWLINK[v]);
        if LOWLINK[v] = DFNUMBER[v] then
        begin
            repeat
                begin
                    pop x from top of STACK;
                    print x
                end
            until x = v;
            print "end of strongly connected component"
        end
    end
```

– 6.6 – Problema del path-finding.

Definizione: Dato un grafo pesato e orientato G, dicasi costo di un cammino p, la somma dei costi degli archi in p stesso.

Legati al costo di un cammino vi sono due problemi, quali

- chiusura transitiva;
- single-source problem.

– 6.6.1 – Chiusura transitiva.

Sia $G* = (V, E*)$ un grafo ottenuto da un grafo orientato $G = (V, E)$, tale che $(v, w) \in E*$ se e solo se esiste in G un cammino da v a w.

Definizione: Un sistema $(S, +, \cdot, 0, 1)$ è detto *semi-anello chiuso*, con S insieme di elementi, $+, \cdot$ operazioni binarie tali da soddisfare le seguenti proprietà.

- $(S, +, 0)$ è un monoide, ossia:
  - $\forall a, b \in S, a + b \in S$;
  - $\forall a, b, c \in S, a + (b + c) = (a + b) + c$;
  - $\forall a \in S, a + 0 = 0 + a = a$.

  discorso analogo per $(S, \cdot, 1)$.
- $+$ è commutativo;
- $\cdot$ è distributivo rispetto $+$;
- se $a_1, \ldots, a_i, \ldots$ è una sequenza di elementi in S, allora $a_1 + \ldots + a_i + \ldots$ esiste ed è unico in S.
- $\cdot$ distribuisce su somme infinite numerabili, come su quelle finite.

Dalle proprietà quattro e cinque, segue

$$\left(\sum_i a_i\right) \cdot \left(\sum_j b_j\right) = \sum_i \left(\sum_j (a_i \cdot b_j)\right)$$

Esempio: Sia $S_1 = (\{0, 1\}, +, \cdot, 0, 1)$, con $+, \cdot$ definite come segue.

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Le prime tre proprietà sono dimostrate banalmente, circa ultime due si osserva che una somma è 0 se tutti gli elementi sono 0.

Esempio: Sia $S_2 = (\mathbb{R}^+, MIN, +, +\infty, 0)$. Risulta ovvio che $+\infty$ sia elemento neutro per MIN e 0 per $+$.

Esempio: Sia $S_3 = (F_\Sigma, \cup, \cdot, \emptyset, \{\varepsilon\})$ dove F_Σ è la famiglia di stringhe su un alfabeto Σ. Le prime tre proprietà sono banalmente dimostrate, le ultime due sono definite se si definisce

$$x \in (A_1 \cup \ldots \cup A_i \cup \ldots) \iff x \in A_i, \text{ per qualche } i$$

Se $(S, +, \cdot, 0, 1)$ è un semi-anello chiuso, inoltre $a \in S$, si definisce $*$ come l'operazione di chiusura, nel seguente modo.

$$\sum_{i=0}^{+\infty} a^i$$

Sia $G = (V, E)$ un grafo orientato, in cui ogni arco è etichettato da un elemento di un semi-anello $(S, +, \cdot, 0, 1)$. Per ogni (v, w) si definisce $c((v, w))$: la somma delle etichette di tutti i cammini tra v, w. La computazione di $c((v, w))$ è effettuata col seguente algoritmo.

```
begin
    for i ← 1 to n do C^0_ii ← 1 + l(v_i, v_i)
    for 1 ≤ i, j ≤ n and i ≠ j do C^0_ij ← l(v_i, v_j)
    for k ← 1 to n do
        for 1 ≤ i, j ≤ n do
            C^k_ij ← C^{k-1}_ij + C^{k-1}_ik · (C^{k-1}_kk)* · C^{k-1}_kj
    for 1 ≤ i, j ≤ n do c(v_i, v_j) ← C^n_ij
end
```

si osservi che $l(v_i, v_j)$ è 0 se non esiste un arco $(v_i, v_j) \in E$, 1 altrimenti. Inoltre se il semi-anello considerato è S_1, l'operazione di chiusura $(C^{k-1}_{kk})*$ può essere omessa.