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	#	#	⋯	#	#	i_k	#	c_k	#	#	⋯

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 $\mathcal{O}(n+k)$.