Capitolo 2

Analisi di algoritmi

> *Begin at the beginning," the King said, gravely, "and go on till you come to an end; then stop.*
>
> Lewis Carroll, *Alice in Wonderland*, 1899

2.1 Introduzione

Il nostro obiettivo è stimare la complessità *in tempo* degli algoritmi. Dovremmo stimare anche quella in spazio, ma la complessità in spazio dipende da quella in tempo. Daremo delle definizioni, parleremo di modelli di calcolo, faremo qualche esempio di valutazione precisa e introdurremo una notazione. Faremo tutto questo per stimare il tempo per un dato input, per stimare il più grande input gestibile in tempi ragionevoli, per avere un metodo di misura per confrontare algoritmi diversi ed in particolare per ottimizzare le parti più importanti dell'algoritmo.

2.2 Definizione di complessità

La complessità viene definita come una funzione che, data la dimensione dell'input, restituisce il tempo, considerato come un valore intero.

Come definiamo quindi la dimensione dell'input? Come misuriamo il tempo?

2.3 Valutare la dimensione dell'input

Esistono due criteri per valutare la dimensione dell'input:

1. il criterio di **costo logaritmico**: dove la dimensione dell'input è il numero di bit necessari per rappresentarlo (un esempio è la moltiplicazione di numeri binari, lunghi n bit);
2. il criterio di **costo uniforme**: dove la dimensione dell'input è il numero di elementi di cui è costituito (un esempio è a ricerca del minimo in un vettote di n elementi).

Ad esempio consideriamo n interi rappresentati tramite 32 bit. Nel criterio di costo uniforme hanno un costo pari a n, mentre nel criterio di costo logaritmico hanno un costo pari a $32n$.

In molti casi, infatti, possiamo assumere che gli "elementi" siano rappresentati da un numero costante di bit e che le due misure coincidano a meno di una costante moltiplicativa.

Il criterio che abbiamo utilizzano fin'ora — e che useremo d'ora in poi — è il criterio del costo uniforme (in casi particolari utilizzeremo il criterio di costo logaritmico).

2.4 Misurare il tempo

Consideriamo un'istruzione come elementare se può essere eseguita in tempo "costante" dal processore. Facciamo qualche esempio:

- `a *= 2` effettua un'operazione di shift, è una singola operazione macchina;

- `Math.cos(d)` può essere considerata come un'operazione elementare;
- `min(A, n)` non può essere considerata un'operazione elementare poiché si richiede il minimo di un vettore *arbitrariamente lungo*.

Ma allora come possiamo distinguere in maniera precisa un'operazione elementare da una che non lo è? Per farlo abbiamo bisogno di un modello di calcolo, ossia una rappresentazione astratta di un calcolatore. Il quale deve:

1. permettere di nascondere i dettagli (tramite astrazione)
2. riflettere la situazione reale (realismo)
3. permettere di trarre conclusioni "formali" sul contesto.

La pagina di Wikipedia dei modelli di calcolo presenta centinaia di modelli di calcolo diversi. La macchina di turing ne è un esempio. È una macchina ideale che manipola – seconodo un insieme prefissato di regole – i dati contenuti su un nastro di lunghezza infinita. Ad ogni passo, la Macchina di Turing:

1. legge il simbolo sotto la testina;
2. modifica il proprio stato interno;
3. scrive il nuovo simbolo nella cella;
4. muove la testina a destra o a sinistra.

Nel corso di laurea magistrale è possibile approfondire questo aspetto, per i nostri scopi questa è una trattazione dell'argomento troppo a basso livello.

Noi utilizzeremo il modello di calcolo RAM, che sta per Random Access Machine, ossia una macchina che ha una quantità infinita di celle (di dimensione finita) e accesso in tempo costante indipendentemente dalla posizione (diversamente da ciò che avviene nei nastri); un singolo processore con un set di istruzioni simile a quelli reali i cui costi di esecuzione sono uniformi e ininfluenti ai fini della valutazione (faremo un esempio più avanti).

2.4.1 Calcolo della complessità

Proviamo a calcolare la complessità dell'algoritmo che ricerca il minimo.

Algoritmo 2.4.1: Calcolo della complessità della ricerca del minimo in un vettore

```
// calcola il minimo di un vettore arbitrariamente lungo
min(ITEM[] A, int n)
    ITEM min ← A[i]                          Costo   # Volte
    from i ← 2 until n do                    c2      n
        if A[i] < min then                   c3      n − 1
            min ← A[i]                       c4      n − 1
    return min                               c5      1
```

Ragionamento sul calcolo della complessità L'assegnazione del minimo viene eseguita solo una volta. Il ciclo viene eseguito n volte. Il controllo $A[i] < min$ viene eseguito $n-1$ volte in quanto dobbiamo guardare tutto il vettore. Consideriamo *il caso pessimo*, ovvero un vettore ordinato in modo decrescente. L'istruzione di ritorno viene eseguita una volta sola.

Bisogna tenere ben a mente che:

- ogni istruzione richiede un tempo costante per essere eseguita e

- viene eseguita un certo numero di volte, dipendente da n e
- la costante è potenzialmente diversa da istruzione a istruzione.

Sommando tutte le costanti il costo totale risultante è:

$$\begin{aligned} T(n) &= c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 \\ &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) \\ &= an + b \end{aligned}$$

raccogliamo / *semplifichiamo*

Possiamo quindi notare che le costanti vanno a semplificarsi nei parametri a e b.

Proviamo a calcolare la complessità dell'algoritmo che ricerca un numero intero all'interno di un vettore ordinato.

Algoritmo 2.4.2: Calcolo della complessità della ricerca di un numero intero in un vettore ordinato

```
// effettua una ricerca binaria su un vettore
binarySearch(ITEM[] A, ITEM v, int i, int j)
```

	Costo	# $i > j$	# $i \leqslant j$
if $i > j$ **then**	c_1	1	1
return 0	c_2	1	0
else			
int $m \leftarrow \lfloor \frac{(i+j)}{2} \rfloor$	c_3	0	1
if $A[m] == v$ **then**	c_4	0	1
return m	c_5	0	0
else if $A[m] < v$ **then**	c_6	0	1
return binarySearch$(A, v, m+1, j)$	$c_7 + T\left(\lfloor \frac{n-1}{2} \rfloor\right)$	0	0/1
else			
return binarySearch$(A, v, i, m-1)$	$c_7 + T(\lfloor n/2 \rfloor)$	0	1/0

Nota. Ci è permesso fare questo ragionamento poiché il vettore è ordinato in ordine decrescente.

Ragionamento sul calcolo della complessità Il vettore viene diviso due parti: la parte sinistra di dimensione $\lfloor \frac{n-1}{2} \rfloor$ e la parte destra di dimensione $\lfloor n/2 \rfloor$. Se n è pari allora il vettore viene diviso in due parti uguali, altrimenti il vettore "di destra" avrà un elemento in più. Si andrà cercare sulla metà sinistra o sulla metà destra a seconda che l'elemento cercato sia più grande o più piccolo rispettivamente. Anche in questo caso consideriamo il caso peggiore, ovvero il caso in cui l'elemento non sia presente. Non prendiamo in considerazione il caso fortunato in cui l'elemento che stiamo cercando sia l'elemento che guardiamo per primo. Nelle chiamate ricorsive dobbiamo considerare (nel costo complessivo) anche il costo delle sotto-chiamate ricorsive con dimensione dell'input pari alla dimesione del vettore passato.

Esercizio Cerca nel vettore ordinato l'elemento 0 tramite la procedura binarySearch, calcolando di volta in volta la dimensione del vettore n.

1	2	3	4	5	6	7	8
8	7	6	5	4	3	2	1

Calcolo del caso pessimo Assumiamo per semplicità che

1. n sia una potenza di 2 ($n = 2^k$, $8 = 2^3$);
2. l'elemento cercato non sia precisato
3. ad ogni passo scegliamo il vettore di destra (di dimensione $n/2$).

Si hanno due casi:

- il caso base $\boxed{i > j}$, dove $n = 0$ e la relazione di ricorrenza è pari a $T(n) = c_1 + c_2 = c$ dove c è una costante;

- il caso ricorsivo $\boxed{i \leqslant j}$, dove $n > 0$ e dobbiamo tener conto di tutte le costanti moltiplicative $T(n) = T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7$, raccogliendo le costanti $T(n) = T(n/2) + d$, dove d è la costante che racchiude tutti i costi.

La relazione di ricorrenza che ne segue è la sequente:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n > 0 \end{cases}$$

Nota. Per calcolare la complessità di una funzione ricorsiva abbiamo bisogno di una funzione di ricorrenza anch'essa ricorsiva.

Le equazioni di ricorrenza così fatte $T(n) = d\log(n) + e$ sono dette in "forma chiusa" e rappresentano la complessità dell'algoritmo (non necessano quindi ulteriori sviluppi).

Risolviamo quindi l'equazione di ricorrenza *tramite espansione*:

$$\begin{aligned} T(n) &= T(n/2) + d && \big) \left(T\left(\tfrac{n}{2} \cdot \tfrac{1}{2}\right) + d\right) + d \\ &= T(n/4) + 2d && \big) \left(T\left(\tfrac{n}{4} \cdot \tfrac{1}{2}\right) + 2d\right) + d \\ &= T(n/8) + 3d \\ &\;\;\vdots && \big) \; n = 2^k \Rightarrow k = \log n \\ &= T(1) + kd && \big) \; T(0) = c \\ &= T(0) + (k+1)\,d \\ &= kd + (c+d) && \big) \; k = \log n \\ &= d\log n + e. \end{aligned}$$

2.5 Ordini di complessità

Per ora, abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due *funzioni di complessità*:

- Ricerca: $T(n) = d\log n + e$. Chiamiamo questa funzione **logaritmica**, utilizzando la notazione $\mathcal{O}(\log n)$;
- Minimo: $T(n) = a + b$. Chiamiamo questa funzione **lineare**, utilizzando la notazione $\mathcal{O}(n)$.

Abbiamo visto anche una terza funzione che deriva dall'algoritmo banale (*naïf*) per la ricerca del minimo:

- Minimo: $T(n) = fn^2 + gn + h$. Chiamiamo questa funzione **quadratica**, utilizzando la notazione $\mathcal{O}(n^2)$.

Tabella 2.1: Classi di complessità

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	logaritmico
$\sqrt{n}$	3	10	31	100	sublineare
n	10	100	1000	10 000	lineare
$n\log n$	30	664	9965	132 877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^{12}	cubico
2^n	1024	10^{30}	10^{300}	10^{3000}	esponenziale

2.6 Funzioni di costo, notazione asintotica

Ora andremo a formalizzare le nozioni sui limiti superiori ed inferiori che abbiamo accennato in maniera informale nelle lezioni precedenti.

Definizione (Funzione di costo)**.** Utilizziamo il termine "funzione di costo" per indicare una funzione $f: \mathbb{N} \to \mathbb{R}$ (dall'insieme dei numeri naturali ai reali).

Definizione (Notazione $\mathcal{O}$)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\mathcal{O}(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cg(n), \forall n \geqslant m$$

Nota. Eventuali fattori moltiplicativi non ci interessano.

La notazione si legge $f(n)$ è "O grande" di $g(n)$ e si scrive $f(n) = \mathcal{O}(g(n))$. Questo è un abuso di notazione, dovemmo scrivere $f(n) \in \mathcal{O}(g(n))$, in quanto $\mathcal{O}$ è un insieme (più precisamente una famiglia di funzioni). Questa notazione è però diventata d'uso comune poiché ci si può fare una specie di aritmetica sopra infatti è la notazione che troverete nella letteratura e sta a significare che $g(n)$ è un limite asintotico superiore per $f(n)$, ossia che $f(n)$ cresce al più (al massimo) come $g(n)$.

Definizione (Notazione Ω)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : f(n) \geqslant cg(n), \forall n \geqslant m$$

La notazione si legge $f(n)$ è "Omega grande" (nella letteratura big-O) di $g(n)$, si scrive $f(n) = \Omega(g(n))$ e sta a significare che $g(n)$ è un limite asintotico inferiore per $f(n)$, ossia che $f(n)$ cresce almeno quanto (non di meno) come $g(n)$.

Definizione (Notazione Θ)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : c_1 g(n) \leqslant f(n) \leqslant c_2 g(n), \forall n \geqslant m$$

La notazione si legge $f(n)$ è "Theta" di $g(n)$, si scrive $f(n) = \Theta(g(n))$ e sta a significare che $f(n)$ cresce *esattamente* come $g(n)$ al di là di fattori moltiplicativi. Nota che $f(n) = \Theta(g(n))$ avviene se e solo se $f(n) = \mathcal{O}(g(n))$ e $f(n) = \Theta(g(n))$.

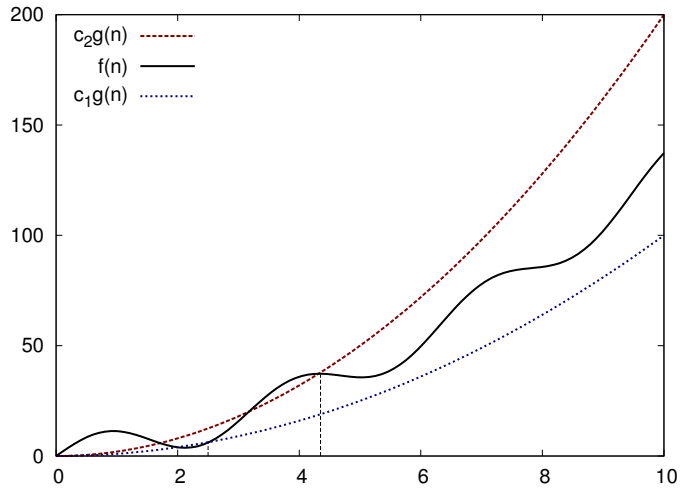

Figura 2.1: Notazione asintotica Θ

2.7 Complessità di un algoritmo e di un problema

Definizione (Complessità in tempo di un algoritmo)**.** La più grande quantità di tempo richiesta per un input di dimensione n.

- $\mathcal{O}(f(n))$: per tutti gli input, l'algoritmo costa al più $f(n)$;
- $\Omega(f(n))$: per tutti gli input, l'algoritmo costa almeno $f(n)$;
- $\Theta(f(n))$: l'algoritmo richiede $\Theta(f(n))$ per tutti gli input.

Definizione (Complessità in tempo di un problema computazionale)**.** La complessità in tempo relativa a tutte le possibili soluzioni.

- $\mathcal{O}(f(n))$: complessità del miglior algoritmo che risolve il problema;
- $\Omega(f(n))$: dimostrare che nessun algoritmo può risolvere il problema in tempo inferiore a $\Omega(f(n))$;
- $\Theta(f(n))$: abbiamo trovato l'algoritmo ottimo.

2.8 Esercizi

Iniziamo con gli esercizi banali che ci permettono di introdurre delle tecniche che utilizzeremo con le ricorrenze. In particolare ci servono a renderci conto che non stiamo dimostrando equazioni, ma disequazioni.

$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 \\
&\leqslant 10n^3 + 2n^3 + 7 && \forall n \geqslant 1 \\
&\leqslant 10n^3 + 2n^3 + 7n^3 && \textit{sommiamo i termini} \\
&= 19n^3 && \textit{esiste una certa costante } c \\
&\stackrel{?}{\leqslant} cn^3 && \textit{per la quale } f(n) \leqslant cn^3 \ ? \\
19n^3 &\leqslant cn^3 && \textit{metto a confronto} \\
19\cancel{n^3} &\leqslant c\cancel{n^3} && \textit{semplifico}
\end{aligned}$$

che è vera per ogni $c \geqslant 19$ (abbiamo così trovato la costante moltiplicativa) e per ogni $n \geqslant 1$ (introdotta nei calcoli), quindi $m = 1$.

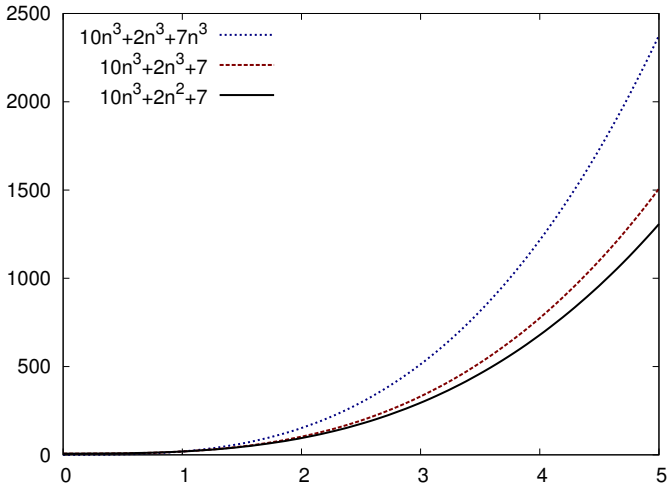

Figura 2.2: Risoluzione grafica dell'esercizio

Nota. In generale noi considereremo solo valori di n positivi, in quanto le funzioni di costo sono definite sull'insieme dei numeri naturali, non ha alcun senso definire una funzione di costo su una dimensione dell'input negativa.

Dato lo stesso esercizio posso esserci passaggi risolutivi diversi. Risolviamo quindi l'esercizio precedente diversamente.

$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$
\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 && \big) \ \forall n \geqslant 1 \\
&\leqslant 10n^3 + 2n^3 + 7 && \big) \ \forall n \geqslant \sqrt[3]{7} \\
&\leqslant 10n^3 + 2n^3 + n^3 && \big) \ \textit{sommiamo i termini} \\
&= 13n^3 && \big) \ \textit{esiste una certa costante } c \\
&\overset{?}{\leqslant} cn^3 && \big) \ \textit{per la quale } f(n) \leqslant cn^3 \ ? \\
13n^3 &\leqslant cn^3 && \big) \ \textit{metto a confronto} \\
13\cancel{n^3} &\leqslant c\cancel{n^3} && \big) \ \textit{semplifico}
\end{aligned}
$$

che è vera per ogni $c \geqslant 13$ e per ogni $n \geqslant \sqrt[3]{7}$ (ad esempio con $n = 2$ abbiamo $n^3 = 2^3 = 8$ che soddisfa la nostra condizione), quindi usiamo $m = 2$ (abbiamo semplificato, sarebbe $m = \sqrt[3]{7}$, ma possiamo prendere un qualunque valore si trovi dopo n in modo totalmente arbitrario).

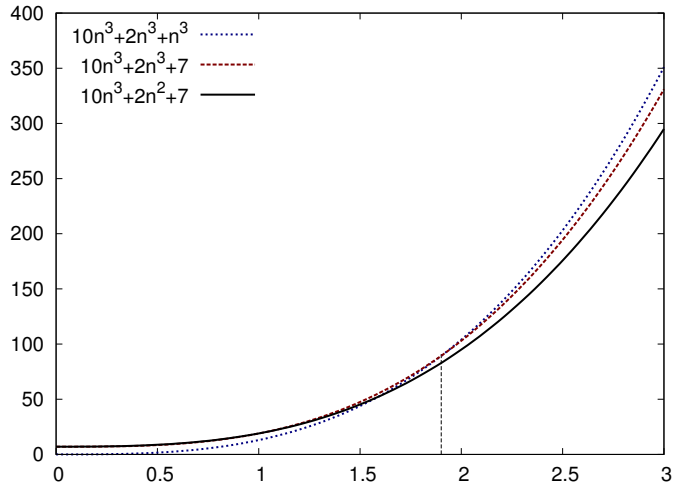

Figura 2.3: Risoluzione grafica dell'esercizio

$$\boxed{f(n) = 3n^2 + 7n \overset{?}{=} \Theta(n^2)}$$

Limite inferiore: Dobbiamo dimostrare che $\exists c_1 > 0, \exists m_1 \geqslant 0 : f(n) \geqslant c_1 n^2, \forall n \geqslant m_1$

$$
\begin{aligned}
f(n) &= 3n^2 + 7n && \big) \ \forall n \geqslant 0 \\
&\geqslant 3n^2 && \big) \ \textit{esiste una certa costante } c \\
&\overset{?}{\geqslant} c_1 n^2 && \big) \ \textit{per la quale } f(n) \leqslant c_1 n^2 \ ? \\
3n^2 &\leqslant c_1 n^2 && \big) \ \textit{metto a confronto} \\
3\cancel{n^2} &\leqslant c_1 \cancel{n^2} && \big) \ \textit{semplifico}
\end{aligned}
$$

che è vera per ogni $c_1 \leqslant 3$ e per ogni $n \geqslant 0$ (introdotta nei calcoli), quindi $m_1 = 0$.
Nota. Abbiamo dimostrato quindi che $f(n) = \Omega(n^2)$.

Limite superiore: Dobbiamo dimostrare che $\exists c_2 > 0, \exists m_2 \geqslant 0 : f(n) \leqslant c_2 n^2, \forall n \geqslant m_2$

$$
\begin{aligned}
f(n) &= 3n^2 + 2n^2 + 7n && \big) \ \forall n \geqslant 1 \\
&\leqslant 3n^2 + 7n^2 && \big) \ \textit{raccogliamo} \\
&\leqslant 10n^2 && \big) \ \textit{esiste una certa costante } c \\
&\overset{?}{\leqslant} c_2 n^2 && \big) \ \textit{per la quale } f(n) \leqslant c_2 n^2 \ ? \\
10n^2 &\leqslant c_2 n^2 && \big) \ \textit{metto a confronto} \\
10\cancel{n^2} &\leqslant c_2 \cancel{n^2} && \big) \ \textit{semplifico}
\end{aligned}
$$

che è vera per ogni $c_2 \geqslant 10$ e per ogni $n \geqslant 1$, quindi $m_2 = 1$.

Nota. Abbiamo dimostrato quindi che $f(n) = \mathcal{O}(n^2)$.

Notazione Θ: $\exists c_1 > 0, \exists c_2 > 0, \exists m \geqslant 0 : c_1 n^2 \leqslant f(n) \leqslant c_2 n^2, \forall n \geqslant m$.

Con questi paramentri:

- $c_1 = 3$
- $c_1 = 10$
- $m = max\{m_1, m_2\} = max\{0, 1\} = 1$, ossia un valore dopo il quale la nostra proprietà è provata

Nota. Abbiamo dimostrato quindi che $f(n) = \Theta(n^2)$

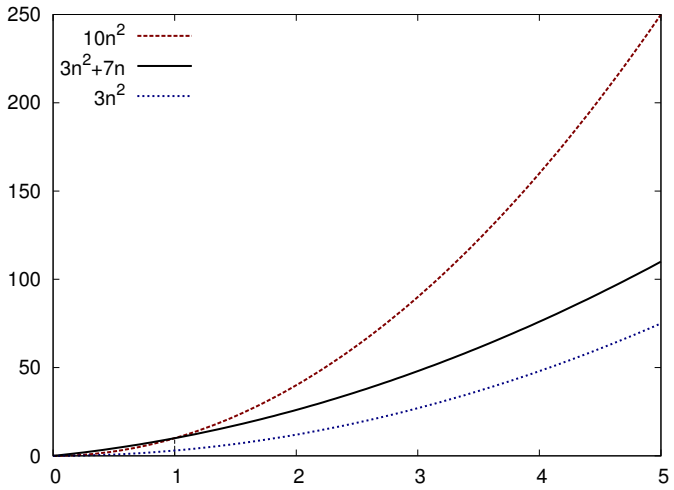

Figura 2.4: Risoluzione grafica dell'esercizio

Errori comuni durante la risoluzione degli esercizi

$$\boxed{f(n) = n^2 \stackrel{?}{=} \mathcal{O}(n)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : n^2 \geqslant cn, \forall n \geqslant m$.

Otteniamo che $n^2 \leqslant cn \Leftrightarrow c \geqslant n$, questo significa che c cresce con il crescere di n, ovvero che non possiamo scegliere una costante c.

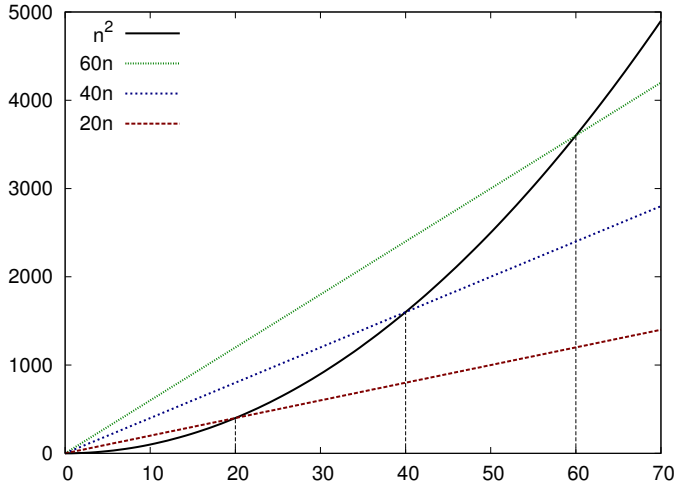

Figura 2.5: Per qualunque fattore c scelto (ossia la pendenza della retta) la curva quadratica crescerà sempre più velocemente da un certo punto in poi

$$\boxed{f(n) = n^2 \stackrel{?}{=} \Omega(n^3)}$$

Limite inferiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0, n^2 \geqslant cn^3, \forall n \geqslant m$.

Otteniamo che $n^2 \geqslant cn^3 \Leftrightarrow c \leqslant \frac{1}{n}$, questo significa che c diminuisce al crescere di n, ovvero che non possiamo scegliere una costante c che verifichi la proprietà.

2.9 Complessità degli algoritmi e dei problemi a confronto

In questa sezione ragioneremo su alcuni algoritmi risolutivi che ci sono stati insegnati, in alcuni casi si può migliorare la complessità, in altri è impossibile fare di meglio.

> Qual è il rapporto fra un problema computazionale e l'algoritmo?

2.9.1 Moltiplicare numeri complessi

La moltiplicazione fra numeri complessi avviene nel seguente modo: $(a+bi)(c+di) = [ac-db]+[ad+bc]i$. Abbiamo in input a, b, c, d e dobbiamo restituire in output $ac-bd$ e $ad+bc$.

Consideriamo un modello di calcolo dove la moltiplicazione costa 1 e le addizioni e sottrazioni costano 0,01.

1. Quanto costa l'algoritmo dettato dalla definizione?
2. Riesci a fare meglio di così?
3. Qual è il ruolo del modello di calcolo?

L'algoritmo banale dettato dalla definizione costa 4,02, in quanto bisogna fare 4 moltiplicazioni, 1 somma ed una sottrazione.

La seguente è la soluzione di Gauss al problema, datata 1805.

Input: a, b, c, d, Output: $A1 = ac - bd$, $A2 = ad + bc$

$$\begin{aligned} m_1 &= a \times c \\ m_2 &= b \times d \\ A_1 &= m_1 - m_2 \\ m_3 &= (a+b)\cdot(c+d) = ac + ad + bc + bd \\ A_2 &= m_3 - m_1 - m_2 = ad + bc \end{aligned}$$

calcolo i valori intermedi

evito una moltiplicazione

Il costo totale è 3,05.

> Si può fare ancora meglio di così? Oppure, è possibile dimostrare che non si può fare di meglio?

2.9.2 Sommare numeri binari

Nota. In questo caso usiamo il criterio del costo logaritmico.

L'algoritmo elementare della somma richiede di esaminare tutti gli n bit, il costo totale risulta cn, dove c è il costo per sommare due bit e generare il riporto.

> Esiste un metodo più efficiente?

È dimostrabile per assurdo che *non è possibile fare di meglio* di una soluzione lineare, poiché non è possibile sommare due numeri binari senza esaminare tutti gli n bit.

Limiti alla complessità di un problema

Definizione (Limite superiore, $\mathcal{O}(f(n))$)**.** Un problema ha complessità $\mathcal{O}(f(n))$ *se esiste almeno un algoritmo* che ha complessità $\mathcal{O}(f(n))$.

Nota. Il problema della somma dei numeri binari ha complessità $\mathcal{O}(n)$.

Definizione (Limite inferiore, $\Omega(f(n))$)**.** Un problema ha complessità $\Omega(f(n))$ *se tutti i possibili algoritmi* che lo risolvono hanno complessità $\Omega(f(n))$.

Nota. Il problema della somma dei numeri binari ha complessità $\Omega(n)$.

2.9.3 Moltiplicare numeri binari

L'algoritmo elementare del prodotto richiede di moltiplicare ogni bit con ogni altro bit, per un costo totale di cn^2.

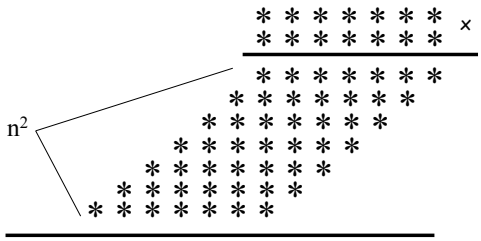

Figura 2.6: Moltiplicazione di due numeri binari

Si potrebbe concludere che il problema della moltiplicazione è molto più costoso del problema dell'addizione: ne è conferma la nostra esperienza.

Nota. Per provare che il problema del prodotto è più costoso del problema della somma, dobbiamo provare che non esiste una soluzione in tempo lineare al problema del prodotto.

Abbiamo infatti erroneamente confrontato gli algoritmi, non i problemi! Sappiamo solo che l'algoritmo della somma che ci hanno insegnato è più efficiente di quello della moltiplicazione.

Nel 1960, Kolmogorov enunciò che la moltiplicazione avesse limite inferiore pari a $\Omega(n^2)$, una settimana dopo un suo studente Karatsuba riuscì a provare il contrario. Osserviamo la sua soluzione.

Moltiplicazione di Karatsuba

Karatsuba adottò un approccio divide-et-impera.

Definizione 2.9.1 (Approccio divide-et-impera)**.** Si svolge in tre parti:

- **Divide**: dividi il problema in sottoproblemi di dimensione inferiore;
- **Impera**: risolvi i sottoproblemi in maniera ricorsiva;
- (**Combina**): unisci le soluzioni dei sottoproblemi in modo da ottenere la risposta del problema principale.

$$X = a \cdot 2^{n/2} + b$$
$$Y = c \cdot 2^{n/2} + d$$
$$XY = ac \cdot 2^{n} + (ad + bc) \cdot 2^{n/2} + db$$

Algoritmo 2.9.1: Algoritmo della moltiplicazione di due numeri binari

```
// moltiplica due numeri binari
bool[ ] pdi(bool[ ] X, bool[ ] Y, int n)
    // X: numero binario
    // Y: numero binario
    // n: numero di bit contenuti
    if n == 1 then
        return X[1] · Y[1] // eseguo la moltiplicazione di due bit
    else
        spezza X in a;b e Y in c;d
        return pdi(a, c, n/2) · 2^n + (pdi(a, d, n/2) + pdi(b, c, n/2)) · 2^(n/2) + pdi(b, d, n/2)
```

Complessità Moltiplicare per 2^t è equivalente ad eseguire uno shift di t posizioni, in tempo lineare, quindi l'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Nota. Non sappiamo ancora trattare questo genere di problemi, quindi facciamo solo degli accenni.

Analisi della ricorsione Al primo passo la chiamata ricorsiva avviene su una dimesione n, al secondo passo vengono effettuate 4 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $4^2 = 16$ chiamate ricorsive su una dimesione $n/2^2$... al livello i-esimo vengono effettuate 4^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $4^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $4^{\log n} = n^{\log 4} = n^2$, le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^2)$. È possibile ridurre ulteriormente la complessità.

$$\begin{aligned} A_1 &= a \times c \\ A_2 &= b \times d \\ m &= (a+b) \times (c+d) = ac + ad + bc + bd \\ A_3 &= m - A_1 - A_2 = ad + bc \end{aligned}$$

calcolo un valore intermedio

evito una moltiplicazione

Principio Effettuo un'unica moltiplicazione che mi permette di calcolare un valore intemedio che contiene la somma di tutte le combinazioni (m), e ricavo $ad + bc$ tramite due sottrazioni (nello stesso modo in cui lavorava Gauss), evitando così una moltiplicazione.

Algoritmo 2.9.2: Algoritmo della moltiplicazione di Karatsuba

```
bool[ ] karatsuba(bool[ ] X, bool[ ] Y, int n)
    if n == 1 then
        return X[1] · Y[1] // rimane invariato
    else
        spezza X in a;b e Y in c;d
        bool[ ] A1 = karatsuba(a, c, n/2)
        bool[ ] A3 = karatsuba(b, d, n/2)
        bool[ ] m = karatsuba(a+b, c+d, n/2) // potrebbe essere n/2 + 1
        bool[ ] A2 = m − A1 − A3 // ottengo A2 tramite sottrazione
        return A1 · 2^n + A2 · 2^(n/2) + A3 // effettuo degli shift
```

Complessità L'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} c_1 & n = 1 \\ 3T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Analisi della ricorsione Al primo passo la chiamata ricorsiva avviene su una dimesione n, al secondo passo vengono effettuate 3 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $3^2 = 9$ chiamate ricorsive su una dimesione $n/3^2$... al livello i-esimo vengono effettuate 3^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $3^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $3^{\log n} = n^{\log 3} = n^{1.58\cdots}$, le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^{1.58\cdots})$.

Nota. L'algoritmo ingenuo (naïf) non è sempre il migliore a meno che non sia possibile dimostrare il contrario.

Negli anni sono stati proposti diversi algoritmi, che il limite inferiore al problema della moltiplicazione sia $\Omega(n \log n)$ è una congettura. Una congettura è un'affermazione o un giudizio fondato sull'intuito, ritenuto probabilmente vero, ma non ancora rigorosamente dimostrato, cioè dunque relegato solamente a rango di ipotesi.

Nella GNU Multiple Precision Arithmetic Library vengono utilizzati diversi algoritmi al crescere di n, il valore soglia per cui si predilige un algoritmo rispetto ad un altro dipende dal tipo di architettura.

2.10 Algoritmi di ordinamento

In questa lezione impareremo a capire quando è meglio utilizzare un algoritmo di ordinamento rispetto ad un altro.

In alcuni casi, gli algoritmi si comportano diversamente a seconda delle caratteristiche dell'input. Conoscere in anticipo tali caratteristiche permette di scegliere l'algoritmo migliore in quella determinata situazione.

Tipologia di analisi

Esistono tre tipi di analisi:

1. Analisi del *caso pessimo*: è la tipologia più importante, il tempo di esecuzione nel caso peggiore è il limite superiore al tempo di esecuzione per qualsiasi input. Per alcuni algoritmi il caso peggiore si verifica molto spesso (ad esempio nella ricerca di dati non presenti nel database);
2. Analisi del *caso medio*: è difficile da definire (cosa si intende per "medio"?), dobbiamo avere una conoscenza pregressa sulle distribuzioni;
3. Analisi del *caso ottimo*: può avere senso se si conoscono informazioni particolari sull'input.

Problema dell'ordinamento

Data una sequenza $A = a_1, a_2, \ldots, a_n$ di n valori in input, il problema dell'ordinamento consiste nel restituire in output una sequenza $B = b_1, b_2, \ldots, b_n$ che sia una permutazione di A, tale per cui $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$ (ovvero che ci sia un ordinamento *totale*).

Un approccio "demente" è quello di generare tutte le possibili permutazioni (complessità $n!$) fino a quando non se ne trova una ordinata.

2.10.1 Selection sort

Un approccio banale (*naïf*) è quello di cercare il minimo e metterlo nella posizione corretta, riducendo il problema agli $n-1$ valori rimanenti.

Algoritmo 2.10.1: Algoritmo selectionSort

```
// effettua l'ordinamento di un vettore
selectionSort(ITEM[ ] A, int n)
    from int i ← 1 until n − 1 do // l'ultimo elemento è ordinato
        int j ← min(A, i, n) // ricerca il nuovo minimo
        swap(A[i], A[j]) // lo metto nella posizione corretta

// cerca l'indice dell'elemento più piccolo
int min(ITEM[ ] A, int i, int j)
    int min ← i // posizione del minimo parziale
    from int j ← i + 1 until n do
        if A[h] < A[min] then // ho trovato un nuovo minimo
            min ← h // nuovo minimo parziale
    return min // restituisco l'indice dell'elemento più piccolo
```

Suggerimento. Provalo su carta! Un algoritmo dev'essere provato per essere capito davvero!

Analisi della complessità Il ciclo effettua n chiamate della funzione min (una per ciascuna iterazione). Ad ogni iterazione il vettore su cui viene calcolato il minimo risulta più piccolo di un elemento.

$$\begin{aligned}
&\sum_{i=1}^{n-1}(n-1) \\
&= \sum_{i=1}^{n-1} i && \text{(} 10+9+\cdots+1 \Leftrightarrow 1+2+\cdots+10 \text{)} \\
&= \frac{n(n-1)}{2} \\
&= n^2 - \frac{n}{2} && \text{(svolgo i calcoli)} \\
&= \Theta(n^2)
\end{aligned}$$

Posso dire che è $\Theta(n^2)$, e non solo che $\Omega(n^2)$, perché indipendentemente dall'ordine dei numeri ci metterà sempre lo stesso tempo. In altre parole, il caso migliore, peggiore e medio coincidono.

2.1.2 Insertion sort

Un algoritmo che si basa sul principio di ordinamento di una "mano" di carte da gioco è il seguente:

Algoritmo 2.1.2: Algoritmo insertionSort

```
// effettua l'ordinamento di un vettore
insertionSort(ITEM[] A, int n)
    from int i = 2 until n do // il 1° elemento verrà ordinato in seguito
        ITEM temp ← A[i] // elemento da ordinare
        int j ← i
        while j > 1 and A[j − 1] > temp do
            A[j] ← A[j − 1] // copio l'elemento
            j++ // mi sposto
        A[j] ← temp
```

Questo è un algoritmo molto efficiente per ordinare piccoli insiemi di elementi.

Analisi della complessità Il costo di esecuzione di questo algoritmo non dipende solo dalla dimensione del vettore, ma anche dalla distribuzione dei dati in ingresso. Nel caso in cui il vettore sia *già ordinato* il costo è $\mathcal{O}(n)$, in quanto non si entra mai nel secondo ciclo dato che la condizione risulta falsa. Nel caso in cui il vettore sia *ordinato in ordine inverso* è $\Omega(n^2)$. In media (informalmente) possiamo assumere che metà dei valori sia ordinata rispetto la loro disposizione finale e quindi metà di loro dovrannno fare n passi per arrivare alla destinazione per una complessità di $n \cdot n/2 = \mathcal{O}(n^2)$.

Infine quando sappiamo che i valori sono quasi ordinati o che n è molto piccolo (nell'ordine di 16 o 32 elementi) allora questo algoritmo risulta efficiente.

2.1.3 Merge sort

MergeSort è basato sulla tecnica divide-et-impera vista in precendenza. Ma come la utilizza?

Definizione 2.1.1 (Approccio divide-et-impera di MergeSort)**.** Si svolge in tre parti:

- **Divide**: spezza il vettore di n elementi in 2 sottovettori di $\frac{n}{2}$ elementi;
- **Impera**: chiama mergeSort ricorsivamente sui due sottovettori (ottenendo due metà ordinate);
- (**Combina**): unisce le due sequenze ordinate (`merge`).

L'idea alla base di questo algoritmo sfrutta il fatto che è possibile unire due sottovettori ordinati in un vettore ordinato in tempo lineare.

Algoritmo 2.1.3: merge

```
// ordina i sottovettori
mergeSort(ITEM[] A, int primo, int ultimo)
    if primo < ultimo then // devono esistere almeno due elementi
        int mezzo ← ⌊(primo+ultimo)/2⌋
        mergeSort(A, primo, mezzo)
        mergeSort(A, mezzo+1, ultimo)
        merge(A, primo, ultimo, mezzo)  // unisce le soluzioni

// effettua l'ordinamento dei sotto-vettori
merge(ITEM A, int primo, int ultimo, int mezzo)
    int i, j, k, h

    // inizializzo i puntatori
    i ← primo
    j ← mezzo
    k ← primo
    // k:  indica la prossima posizione di scrittura

    while i ⩽ mezzo and j ⩽ ultimo do
        // B è il vettore di appoggio in cui memorizzo la porzione di vettore già ordinata
        if A[i] ⩽ A[j] then
            // l'elemento è gia ordinato
            B[k] ← A[i]
            i++
        else
            B[k] ← A[j]
            j++

        // in entrambi i casi ho inserito un valore
        k++

    // se uno dei due vettori finisce ricopio la parte ordinata alla fine del vettore d'appoggio
    j ← ultimo
    from h ← mezzo until i do
        A[j] ← A[h]
        j++

    // ricopio il vettore d'appoggio del vettore originale
    from j ← primo until k − 1 do
        A[j] ← B[j]
```

Analisi della complessità Assumiamo (per semplicità) che $n = 2^k$ (ovvero che l'altezza dell'albero di suddivisioni sia esattamente $k = \log_2 n$) e che tutti i sottovettori abbiano dimensioni che sono potenze

esatte di 2. L'equazione di ricorrenza risultante è la seguente:

$$T = \begin{cases} \Theta(1) & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + \Theta(n) & n > 1 \end{cases}$$
$$= \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

Qual è il costo computazionale di mergeSort?

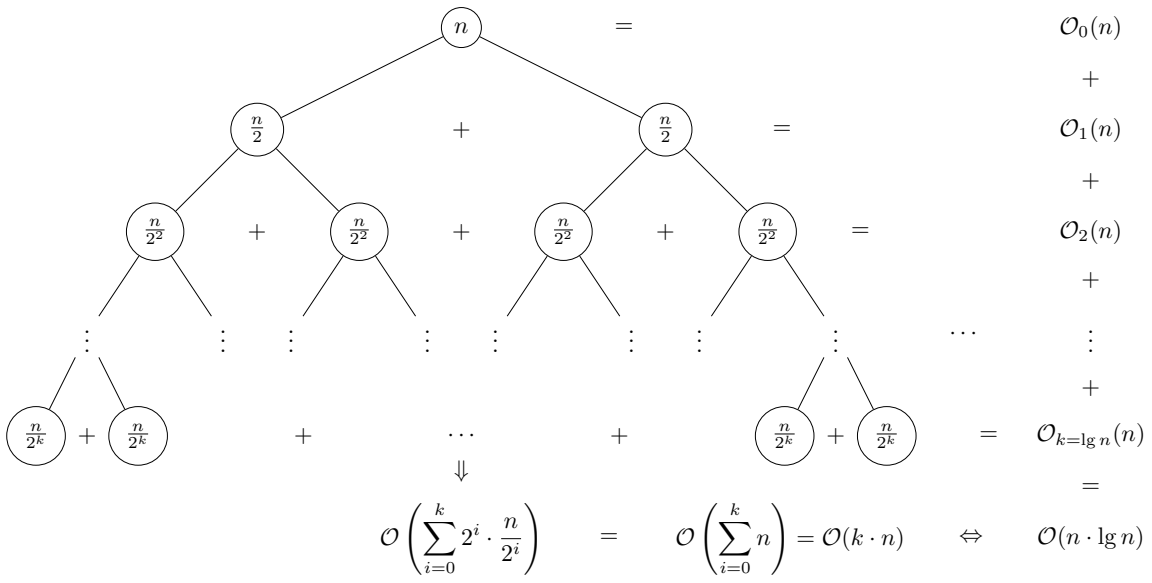

Figure 2.7: Analisi per livelli

L'analisi per livelli è la seguente:

$$\begin{aligned} &\mathcal{O}\left(\textstyle\sum_{i=0}^{k} 2^i \frac{n}{2^i}\right) \\ &= \mathcal{O}\left(\textstyle\sum_{i=0}^{k} n\right) && \text{semplifico} \\ &= \mathcal{O}((k+1) \cdot n) && \text{equivalente} \\ &= \mathcal{O}(k \cdot n) && k+1 \text{ elementi, semplico perché è una costante} \\ &= \mathcal{O}(n \log n) && k = \log n \end{aligned}$$

$\mathcal{O}(n \log n)$ è asintoticamente migliore di $\mathcal{O}(n^2)$. Questo algoritmo è preferibile — per grandi dimensioni di n — al selectionSort e all'insertionSort.

Modifiche richieste

12.2 Hateville

Descrizione del problema Hateville è un villaggio particolare, composto da n case, numerate da 1 a n lungo una singola strada. Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati. Quindi il vicino i odia i vicini $i-1$ e $i+1$ (se esistenti). Hateville vuole organizzare una sagra e vi ha affidato il compito di raccogliere i fondi. Ogni abitante i ha intenzione di donare una quantità $D[i]$, ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini.

Dobbiamo scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta.

Consegne del problema I problemi che possono esserci posti sono due.

1. scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta;
2. scrivere un algoritmo che restituisca il sottoinsieme di indici $S \subseteq \{1, \ldots, n\}$ tale per cui la donazione totale $T = \sum_{i \in S} D[i]$ è massimale.

se risolviamo il primo siamo ad un passo dalla soluzione del secondo, mentre se risolviamo il primo abbiamo risolto necessariamente il primo.

Esempi di esecuzione Con un vettore di donazioni $D = [4, 3, 6]$ la raccolta massima è 10, dato dall'insieme di indici $\{1, 3\}$. Mentre con un vettore di donazioni $D = [10, 5, 5, 10]$ la raccolta massima è 20, dato dall'insieme di indici $\{1, 4\}$.

La domanda che dobbiamo porci è la seguente: è possibile ridefinire una formula ricorsiva che ci permetta di calcolare il sottoinsieme di case che, se selezionate, dà origine alla maggior quantità di donazioni?

Ridefiniamo il problema caratterizzandolo matematicamente. Definiamo $HV(i)$ uno dei possibili insiemi di indici da selezionare per ottenere una donazione ottimale delle prime i case di Hateville, numerate $1, \ldots, n$. $HV(n)$ è la soluzione del problema originale.

Andiamo per passi. Consideriamo il vicino i-esimo. Cosa succede se non accetto la sua donazione? Lo scarto. Proviamo ad esprimerlo in funzione dei problemi precedenti (dopotutto il problema viene risolto in maniera ricorsiva):

$$HV(i) = HV(i-1)$$

Cosa succede se accetto la sua donazione? Accetto la donazione i-esima e scarto i vicini.

$$HV(i) = \{i\} \cup HV(i-2)$$

A questo punto come faccio a decidere quale delle due opzioni scegliere? Semplicemente prendo quello che mi dà un guadagno maggiore. In simboli:

$$HV(i) = highest(HV(i-1), \{i\} \cup HV(i-2))$$

La funzione *highest* restituisce l'insieme di valore massimo.

12.2.1 Sottostruttura ottima

[EN 1]: aiuto con l'italiano Quando voglio provare che una soluzione di programmazione dinamica è corretta devo riuscire a dimostrare che le mie possibili scelte sono quelle giuste. Per dimostrarlo utilizziamo il teorema di sottostruttura ottima, che dice sostanzialmente che il modo in cui ho applicato la ricorsione è corretto. Proviamo quindi a dimostrare le scelte fatte.

Il problema dato dalle prime i case è indicato con $HV_p(i)$ e una (ce ne può essere più di una) soluzione ottima per questo problema è indicato con $HV_s(i)$. [EN 2]: aiuto con l'italiano Se abbiamo la soluzione ottima, allora possiamo dimostrare che abbiamo la soluzione ottima per i rispettivi sottoproblemi (da qui sottostruttura).

Quindi se $i \in HV_s i$ allora $HV_s(i) = HV_s(i-1)$ (ossia la soluzione per il problema $HV_s i$ è la soluzione per il problema $HV_s(i-1)$), altrimenti se $i \notin HV_s i$ allora $HV_s(i) = HV_s(i-2) \cup \{i\}$.

Nota. Nella maggior parte dei casi non è necessaria una dimostrazione della soluzione, ma basta un'intuizione.

Dimostrazione

Indichiamo con $|HV_s(i)|$ l'ammontare di donazioni per la soluzione ottima $HV_s(i)$.

Nota. Sono due casi separati poiché sono mutualmente esclusivi: o prendiamo la donazione i-esima o non la prendiamo non ci sono altre possibilità.

Dimostrazione primo caso: $\boxed{i \notin HV_s(i)}$. Vogliamo dimostrare che $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$.

Se così non fosse esisterebbe una soluzione (migliore) $HV_s^{'}(i-1)$ per il problema $HV_p(i-1)$ tale che $|HV_s^{'}(i-1)| > |HV_s(i)|$.

Ma allora $HV_s^{'}(i-1)$ sarebbe una soluzione per $HV_p(i)$ (tale che $|HV_s^{'}(i-1)| > |HV_s(i)|$), che è assurdo. Quindi $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$. □

Dimostrazione secondo caso: $\boxed{i \in HV_s(i)}$. $i-1 \notin HV_s(i)$ (il precedente non appartiene alla soluzione ottima), altrimenti non sarebbe una soluzione ammissibile. Quindi, $HV_s(i) - \{i\}$ è una soluzione ottima per $HV_p(i-2)$.

Se così non fosse, esisterebbe una soluzione $HV_s^{'}(i-2)$ per il problema $HV_p(i-2)$ tale che $|HV_s^{'}(i-2)| > |HV_s(i) - \{i\}|$.

Ma allora $HV_s^{'}(i-2) \cup \{i\}$ sarebbe una soluzione per $HV_p(i)$ tale che $|HV_s^{'}(i-2) \cup \{i\}| > |HV_s(i)|$, il che è assurdo. □

12.2.2 Completare la ricorsione

Ragioniamo sui casi base. Se ho 0 case il mio guadagno è zero: $HV(0) = \varnothing$; se ho una casa prendo semplicemente la sua donazione: $HV(1) = \{1\}$.

Possiamo quindi scrivre una formula per calcolare la somma massima date i case:

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ highest(HV(i-1), HV(i-2) \cup \{i\}) & i \geqslant 2 \end{cases}$$

Non vale la pena scrivere un algoritmo ricorsivo, basato su divide-etimpera, per risolvere il problema di Hateville poiché si risolverebbero molti sottoproblemi più volte.

12.2.3 Memorizzare una tabella

Facciamo qualche esempio di esecuzione. Nel primo il vettore delle donazioni è $D = [10,5,5,8,4,7,12]$, mentre nel secondo è $D = [10,1,1,10,1,1,10]$. Convincersi che gli insiemi risultanti sono corretti.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
HV	$\varnothing$	{1}	{1}	{1,3}	{1,4}	{1,3,5}	{1,4,6}	{1,3,5,7}

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
HV	$\varnothing$	{1}	{1}	{1,3}	{1,4}	{1,4}	{1,4,6}	{1,4,7}

A questo punto dobbiamo risolvere ancora due problemi:

1. dobbiamo definire la funzione *highest* ma è banale;
2. dobbiamo memorizzare gli insiemi nella tabella, ma è costoso quindi lo non faremo, infatti noi andremo a costruire il valore della soluzione e non la soluzione e ci permetterà di ricostruire la soluzione a posteriori.

Tabella di programmazione dinamica

Indichiamo con $DP[i]$ il *valore* della massima quantità di donazioni che possiamo ottenere dalle prime i case di Hateville, e con $DP[n]$ il valore della soluzione ottima.

Possiamo quindi riempire la tabella di programmazione dinamica nel seguente modo:

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ \max(DP[i-1], DP[i-2] + DP[i]) & i \geqslant 2 \end{cases}$$

Nota. Non memorizziamo più insiemi, ma valori. Infatti non effettuiamo più l'unione di insiemi ma la somma fra i valori contenuti all'interno della tabella.

Di seguito vediamo un algoritmo iterativo che risolve questo particolare problema, nel caso volessimo implementare un algoritmo ricorsivo allora dovremmo utilizzare la tecnica della memoization che vedremo più avanti.

Algoritmo 12.2.1: Algoritmo iterativo che risolve il problema Hateville

```
int hateville(int[ ] D, int n)
    // creo la tabella
    int[ ] DP ← new int[ ]0n
    // inserisco i casi base
    DP[0] ← 0
    DP[1] ← DP[1]
    // calcolo il valore i-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i − 1], DP[i − 2] + D[i])
    // restituisco il valore n-esimo
    return DP[n]
```

Stiamo calcolando la soluzione per ogni possibile sottoproblema $(n+1)$ qual è il valore massimo della soluzione. Questa soluzione ha complessità $\Theta(n)$ in quanto dobbiamo fare $\Theta(n)$ somme da fare per ottenere il risultato.

12.2.4 Soluzione con linguaggi di programmazione

Vediamo un paio di implementazioni con linguaggi di programmazione. Notiamo che gli indici iniziano da 1. Gli indici differiscono dalla notazione matematica.

Codice 12.1: Implementazione della soluzione in Java

```java
public int hateville(int[] D, int n) {
    int[] DP = new int[n+1];
    DP[0] = 0;
    DP[1] = D[0];
    for (int i=2; i <= n; i++) {
        DP[i] = max(DP[i-1], DP[i-2] + D[i-1]);
    }

    return DP[n];
}
```

Codice 12.2: Implementazione della soluzione in Python

```python
def hateville(D):
        DP = [ 0, D[0] ]

        for i in range(1,len(D)):
                DP.append( max(DP[-1], DP[-2] + D[i]) )

        return DP[-1]
```

12.2.5 Ricostruire la soluzione originale

Questi sono i possibili risultati che possiamo ottenere applicando l'algoritmo.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
DP	0	10	10	15	18	19	25	31

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
DP	0	10	10	11	20	20	21	30

A questo punto abbiamo il valore della soluzione massimale, ma non abbiamo la soluzione (l'insieme degli indici)!

Per riscostruire la soluzione guardiamo l'elemento i-esimo presente nella tabella nella posizione $DP[i]$, se la casa i-esima non è stata selezionata allora il valore di $DP[i]$ deriva da $DP[i-1]$, altrimenti (se la casa è stata selezionata) il suo valore deriva da $DP[i-2]+D[i]$. Utilizziamo quindi questa informazione per ricostruire la soluzione in modo ricorsivo: per ricostruire la soluzione fino ad i calcoliamo i valori fino a $i-1$ senza aggiungere nulla se la casa non è stata selezionata, altrimenti li calcoliamo fino a $i-2$ e aggiungiamo i.

Algoritmo 12.2.2: Ricostruire la soluzione generale di Hateville

```
int hateville(int[ ] D, int n)
    // creo la tabella
    int[ ] DP ← new int[ ]0n
    // inserisco i casi base
    DP[0] ← 0
    DP[1] ← DP[1]
    // calcolo il valore i-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i − 1], DP[i − 2] + D[i])
    // Restituisco il valore n-esimo
    return solution(DP, D, n)
int solution(int[ ] DP, int[ ] D, int i)
    // i:  indice di scorrimento
    if i == 0 then // caso base
        return ∅
    else if i == 1 then// caso base
        return {1}
    else if DP[i] == DP[i − 1] then// non seleziono la casa
        return solution(DP, D, i − 1)
    else // seleziono la casa
        Set sol = solution(DP, D, i − 2)
        sol.insert(i)
        return sol
```

Analisi della complessità La complessità computazionale di solution è $T(n) = \Theta(n)$, quella spaziale è $S(n) = \Theta(n)$.

Nota. Non è possibile migliorare la complessità spaziale di hateville poiché è necessario ricostruire la soluzione.