Indice

Capitolo 1

Esercizi Ricorsione

Esercizio 1.1. *Stimare utilizzando il Metodo dell'Esperto la complessità della seguente ricorrenza, esplicitando in quale caso cade:*
$T(n) = 2T(\frac{n}{2}) + n^2$
Soluzione

Il **Master Theorem** rappresenta un "ricettario" per risolvere le ricorrenze, ovvero data una certa ricorrenza $T(n)$ stimiamo il suo costo senza dover svolgere complessi calcoli. Infatti, individuati i 3 parametri che guidano la complessità di una ricorrenza, il M.T. ci indirizza in uno delle sue 3 casistiche. I 3 parametri chiave di una ricorrenza sono:

- **a**, che definisce il numero di sottoproblemi che abbiamo ogniqualvolta si ricorre;
- **b**, che indica la dimensione dei sottoproblemi (la frazione del problema originale);
- **f(n)**, che definisce il costo del passo combina, i.e., il costo delle operazioni che svolgiamo prima di ricorrere.

In questo contesto identifichiamo 3 diverse casistiche confrontando la crescita **asintotica** delle funzioni: $f(n)$ e $n^{\log_b a}$. In particolare, se:

1. $n^{\log_b a}$ cresce più velocemente asintoticamente di $f(n)$ cadiamo nel **primo caso**;
2. $n^{\log_b a}$ cresce asintoticamente come $f(n)$ cadiamo nel **secondo caso**;
3. $n^{\log_b a}$ cresce più lentamente asintoticamente di $f(n)$ cadiamo nel **terzo caso**;

Veniamo alla soluzione dell'esercizio. Come sottolineato nelle righe precedenti dobbiamo isolare i tre parametri chiave della nostra ricorrenza:

- a= 2; b= 2; e f(n)= n^2

Pertanto abbiamo che $n^{\log_b a} = n^{\log_2 2} = n \implies \mathcal{O}(n^2) > \mathcal{O}(n)$.
Pertanto verifichiamo se effettivamente $T(n)$ è riconducibile al terzo caso.

- $\exists \epsilon > 0$ t.c. $f(n) \in \Omega(n^{\log_b a + \epsilon})$

- Per $\epsilon = 1$ abbiamo che $n^2 \in \Omega(n^{\log_2 2+1}) = \Omega(n^2)$ **OK!**

- **Condizione di regolaritá:** $af(\frac{n}{b}) \leq cf(n)$
  - $2(\frac{n}{2})^2 \leq cn^2$; verificato per $c = \frac{1}{2}$ **OK!**

Pertanto cadiamo effettivamente nel terzo caso del M.T. $\implies T(n) \in \Theta(f(n)) = \Theta(n^2)$.

Esercizio 1.2. *Stimare utilizzando il Metodo dell'Esperto la complessità della seguente ricorrenza, esplicitando in quale caso cade:*
$T(n) = T(\frac{8n}{11}) + n$
Soluzione

Come fatto in precedenza isoliamo i tre parametri chiave della nostra ricorrenza:

- a= 1; b= $\frac{11}{8}$; e f(n)= n

Pertanto abbiamo che $n^{\log_b a} = n^{\log_{\frac{11}{8}} 1} = 1 \implies \mathcal{O}(n) > \mathcal{O}(1)$.
Pertanto verifichiamo se effettivamente $T(n)$ è riconducibile al terzo caso.

- $\exists \epsilon > 0$ t.c. $f(n) \in \Omega(n^{\log_b a+\epsilon})$
  - Per $\epsilon = 1$ abbiamo che $n \in \Omega(n^{\log_{\frac{11}{8}} 1+1}) = \Omega(n)$ **OK!**
- **Condizione di regolaritá:** $af(\frac{n}{b}) \leq cf(n)$
  - $(\frac{8n}{11}) \leq cn^2$; verificato per $c = \frac{8}{11}$ **OK!**

Pertanto cadiamo effettivamente nel terzo caso del M.T. $\implies T(n) \in \Theta(f(n)) = \Theta(n)$.

Esercizio 1.3. *Stimare utilizzando il Metodo dell'Esperto la complessità della seguente ricorrenza, esplicitando in quale caso cade:*
$T(n) = 16T(\frac{n}{4}) + n^2$
Soluzione

Come fatto in precedenza isoliamo i tre parametri chiave della nostra ricorrenza:

- a= 16; b= 4; e f(n)= $n^2 \implies n^{\log_b a} = n^{\log_4 16} = n^2$

Pertanto abbiamo che $\mathcal{O}(n^2) = \mathcal{O}(n^2)$.
Pertanto verifichiamo se effettivamente $T(n)$ è riconducibile al secondo caso.

- $\exists k \geq 0$ t.c. $f(n) \in \Theta(n^{\log_b a} \log^k n)$
  - Per $k = 0$ abbiamo che $n^2 \in \Theta(n^{\log_4 16} \log^0 n) = \Theta(n^2)$ **OK!**

Pertanto cadiamo effettivamente nel secondo caso del M.T. $\implies T(n) \in \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^2 \log n)$.

Esercizio 1.4. *Stimare utilizzando il Metodo dell'Esperto la complessità della seguente ricorrenza, esplicitando in quale caso cade:*
$T(n) = 4T(\frac{n}{2}) + n$
Soluzione

> Come fatto in precedenza isoliamo i tre parametri chiave della nostra ricorrenza:
>
> - a= 4; b= 2; e f(n)= $n \implies n^{\log_b a} = n^{\log_2 4} = n^2$
>
> Pertanto abbiamo che $\mathcal{O}(n) < \mathcal{O}(n^2)$.
> Pertanto verifichiamo se effettivamente $T(n)$ è riconducibile al primo caso.
>
> - $\exists \epsilon > 0$ t.c. $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$
>   - Per $\epsilon = 1$ abbiamo che $n \in \mathcal{O}(n^{\log_2 4 - 1}) = \mathcal{O}(n)$ **OK!**
>
> Pertanto cadiamo effettivamente nel primo caso del M.T. $\implies T(n) \in \Theta(n^{\log_b a}) = \Theta(n^2)$.

Esercizio 1.5. *Risolvere la seguente equazione di ricorrenza:*
$T(n) = 4T(\frac{n}{2}) + \mathcal{O}(n^3)$
Soluzione

> Seppure l'esercizio non ci vincoli ad utilizzare alcuna tecnica per risolvere la ricorrenza, sembrebbe che $T(n)$ sia attacabile sfruttando il M.T..
>
> Proviamo:
>
> - a= 4, b= 2, f(n)= $n^3 \implies n^{\log_b a} = n^{\log_2 4} = n^2$
>
> $f(n) = \mathcal{O}(n^3) > \mathcal{O}(n^2)$, verifichiamo se cadiamo nel terzo caso.
>
> - $\exists \epsilon > 0$ t.c. $f(n) \in \Omega(n^{\log_b a + \epsilon})$
>   - Per $\epsilon = 1$ abbiamo che $n^3 \in \Omega(n^{\log_2 4 + 1}) = \Omega(n^3)$ **OK!**
> - **Condizione di regolaritá:** $af(\frac{n}{b}) \leq cf(n)$
>   - $4(\frac{n}{2})^3 \leq cn^3$; verificato per $c = \frac{1}{2}$ **OK!**
>
> $\implies T(n) \in \Theta(f(n)) = \Theta(n^3)$.

Esercizio 1.6. *Stimare la complessitá in tempo della seguente ricorrenza:*
$$T(n) = \begin{cases} \mathcal{O}(1), & n \leq 2 \\ 8T(\frac{n}{2}) + \sum_{i=1}^{n} \frac{n}{i}, & n > 2 \end{cases}$$

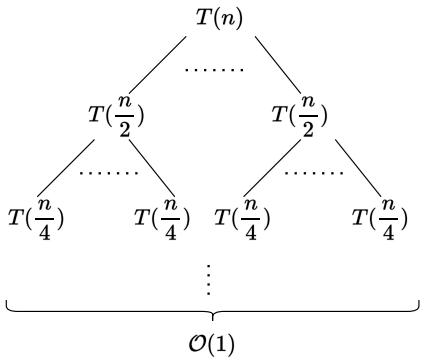
Soluzione

> Per stimare il costo della ricorrenza $T(n)$ dobbiamo innanzitutto stimare il valore di $\sum_{i=1}^{n} \frac{n}{i}$:
> $\sum_{i=1}^{n} \frac{n}{i} = n \sum_{i=1}^{n} \frac{1}{i} = \mathcal{O}(n \log n)$

Pertanto possiamo riscrivere la ricorrenza nel seguente modo:

$$T(n) = \begin{cases} \mathcal{O}(1), & n \leq 2 \\ 8T(\frac{n}{2}) + \mathcal{O}(n \log n), & n > 2 \end{cases}$$

Proviamo ora ad utilizzare l'albero della ricorsione per stimare una complessità per $T(n)$:



Nodo generico: $T(n/2^i)$

Altezza: $\frac{n}{2^i} \leq 2 \Rightarrow i > \log_2 \frac{n}{2} = \log_2 n - 1$

Costo livello generico: $8^i(\frac{n}{2^i} \log \frac{n}{2^i})$

Numero foglie: $8^{\log_2 n} = n^{\log_2 8} = n^3$

Arrivati a questo punto possiamo stimare il costo complessivo dell'albero costruito:

$\sum_{i=0}^{\log_2 n - 1}(8^i(\frac{n}{2^i} \log_2 \frac{n}{2^i})) + n^3 \leq \sum_{i=0}^{\log_2 n - 1}(8^i(\frac{n}{2^i} \log_2 n)) + n^3 = n \log_2 n \sum_{i=0}^{\log_2 n - 1}(4^i) + n^3 =$
$= n \log_2 n(\frac{4^{\log_2 n} - 1}{4-1}) + n^3 \leq n^3 \log_2 n + n^3 \implies \mathcal{O}(n^3 \log_2 n)$

Proviamo per induzione il limite appena stimato:

Caso base:

$T(1) = \mathcal{O}(1), T(2) = \mathcal{O}(1) \implies$ **OK!**

Dobbiamo ora verificare che $T(n) \in \mathcal{O}(n^3 \log_2 n)$, cioè $T(n) \leq cn^3 \log_2 n$ per qualche $c > 0$ e per qualche $n_0 > 0$.

Ipotesi induttiva:

$T(\frac{n}{2}) \leq c(\frac{n}{2})^3 \log_2(\frac{n}{2})$

Allora

$T(n) \leq 8c(\frac{n}{2})^3 \log_2(\frac{n}{2}) + c_1 n \log_2 n$
$= cn^3 \log_2(\frac{n}{2}) + c_1 n \log_2 n$
$= cn^3 \log_2(n) \underbrace{-cn^3 + c_1 n \log_2 n}_{\leq 0}$

Osserviamo che $-cn^3 + c_1 n \log_2 n \leq 0$ già a partire da un $n_0 = 1$ e da $c = c_1$, pertanto

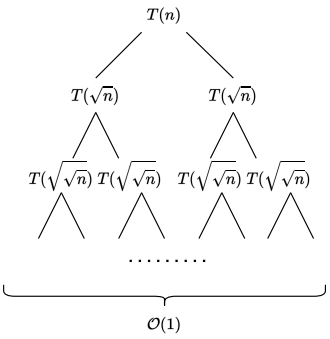
$\leq cn^3 \log_2(n) \implies$ **OK!**

Esercizio 1.7. *Stimare la complessitá in tempo della seguente ricorrenza:*

$$T(n) = \begin{cases} \mathcal{O}(1), & n \leq 2 \\ 2T(\sqrt{n}) + \log_2 n, & n > 2 \end{cases}$$

Soluzione

Ci avvaliamo dell'albero della ricorsione per stimare la complessitá della nostra ricorrenza.



Nodo generico: $T(n^{\frac{1}{2}^i})$

Altezza: $n^{\frac{1}{2}^i} \leq 2 \implies \frac{1}{2}^i \log_2 n \leq \log_2 2 \implies i \geq \log_2 \log_2 n$

Costo livello generico: $2^i \log_2(n^{\frac{1}{2}^i})$

Numero foglie: $2^{\log_2 \log_2 n} \implies \log_2 n$

Possiamo dunque calcolarci il costo totale del nostro albero:

$\sum_{i=0}^{\log_2 \log_2 n} 2^i \log_2(n^{\frac{1}{2}^i}) + \log_2 n = \sum_{i=0}^{\log_2 \log_2 n} \log_2 n + \log_2 n \implies \in \mathcal{O}(\log n \log\log n)$.

Proviamo il bound stimato per induzione per accertarci che sia corretto:

$\exists c > 0, \exists n_0 > 0$ t.c. $\forall n \geq n_0, T(n) \leq c \log n \log\log n$

Il caso base è chiaramente valido, pertanto formuliamo la nostra ipotesi induttiva.

Hp: $T(\sqrt{n}) \leq c \log \sqrt{n} \log\log \sqrt{n}$
$T(n) \leq 2c \log_2 \sqrt{n} \log_2 \log_2 \sqrt{n} + \log_2 n$
$= c \log_2 n \log_2(\frac{1}{2} \log_2 n) + \log_2 n$
$= c \log_2 n (\log_2 \frac{1}{2} + \log_2 \log_2 n) + \log_2 n$
$= c \log_2 n \log_2 \log_2 n \underbrace{-c \log_2 n + \log_2 n}_{\leq 0}$

Abbiamo che la nostra prova è verificata quando:

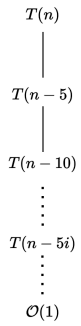
$-c \log_2 n + \log_2 n \leq 0 \implies c \geq 1$
$\implies \leq c \log_2 n \log_2 \log_2 n$ **OK!**

Esercizio 1.8. *Stimare la complessitá in tempo della seguente ricorrenza:*

$$T(n) = \begin{cases} T(n-5) + \mathcal{O}(1), & n > 5 \\ \mathcal{O}(1), & n \leq 5 \end{cases}$$

Soluzione

Ci avvaliamo dell'albero della ricorsione per stimare la complessitá della nostra ricorrenza.



Nodo generico: $T(n-5i)$

Altezza: $n - 5i \leq 5 \implies i \geq \frac{n}{5} - 1$

Costo livello generico: $\mathcal{O}(1)$

Numero foglie: 1

Possiamo dunque calcolarci il costo totale del nostro albero:

$\sum_{i=0}^{\frac{n}{5}-1} \mathcal{O}(1) + 1 = \frac{n}{5} - 1 + 1 \implies \Theta(n)$.

Proviamo il bound stimato per induzione per accertarci che sia corretto:

$\exists c > 0, \exists n_0 > 0$ t.c. $\forall n \geq n_0, T(n) \leq cn$

Hp: $T(n-5) \leq c(n-5)$
$T(n) \leq c(n-5) + \mathcal{O}(1)$
$= cn \underbrace{-5c + c_1}_{\leq 0}$

Abbiamo che la nostra prova è verificata quando:

$-5c + c_1 \leq 0 \implies c \geq \frac{c_1}{5}$
$\implies \leq cn$ **OK!**

Esercizio 1.9. *Stimare la complessitá in tempo della seguente ricorrenza:*

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n, & n > 4 \\ \mathcal{O}(1), & n \leq 4 \end{cases}$$

Soluzione

Per stimare il costo di questa ricorrenza possiamo avvelerci del Master Theorem, pertanto:

- $a = 2$
- $b = 2$
- $f(n) = n$

$\implies$ #foglie $= n^{\log_2 2} = n$

Osserviamo che che $f(n)$ e #foglie sono funzioni che asintoticamente uguali, quindi proviamo a verificare il **secondo caso** del M.T..

$\exists k \geq 0$ t.c. $f(n) \in \Theta(n^{\log_b a} \log^k n)$, allora $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$

Questo è vero con $k = 0$, infatti è chiaramente vero che:

$n \in \Theta(n \log^0 n) \implies T(n) \in \Theta(n \log n)$.

Esercizio 1.10. *Risolvere con l'albero della ricorsione l'equazione di ricorrenza che rappresenta la complessità in tempo del seguente algoritmo:*

```
Procedure Pippo (n)
begin
    k = n, i = 1
    while k ≥ 5 do
        a = 0
        for j=1; j≤ 7; j++ do
            a = a + 1
        k = k − 12 + a
    if n ≥ 10 then
        Pippo = 2*Pippo(n/2) + √n
    else
        Pippo = 100
```

end

Soluzione

L'esercizio proposto risulta essere analogo a quelli visti nel Capitolo **??**, dove era necessario individuare le parti di codice più "pesanti" dal punto di vista computazionale al fine di stimare un limite alla complessità della funzione. L'unica differenza che osserviamo è la **chiamata ricorsiva** nel ramo vero del nostro IF. Questo ci suggerisce che dovremo derivare per la procedura PIPPO una equazione di ricorrenza, che studieremo per identificare la complessità della stessa.

Cominciamo quindi con il capire il costo dei due cicli innestati. Osserviamo che il FOR più interno esegue sempre un numero iterazioni pari a 7, e al suo interno abbiamo solo operazioni aritmetiche con costo costante. Come conseguenza notiamo anche che il valore di a sarà sempre uguale a 4, e pertanto il passo con cui decresce k può essere riscritto come $k = k - 5$.

Quindi possiamo stimare il costo del WHILE costruendo la nostra solita tabella:

step: 0	1,	2,	3,	...	i
k: n	$n-5$,	$n-10$,	$n-15$,	...	$n-5i$

Dunque quanti passi eseguiamo prima di fermarci? Lo scopriamo studiando la seguente disequazione:

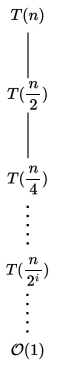
$n - 5i < 5 \implies i > \frac{n}{5} - 1$

Il costo dei due cicli è desumibile da $\sum_{j=0}^{\frac{n}{5}-1} \mathcal{O}(1) = \frac{n}{5} - 1 \in \Theta(n)$.

La nostra ricorrenza sarà dunque:

$$T(n) = \begin{cases} T(\frac{n}{2}) + \Theta(n), & n \geq 10 \\ \Theta(1), & n < 10 \end{cases}$$

Per studiare la ricorrenza potremmo utilizzare il M.T. cadendo nel **terzo caso**, ma a scopo dimostrativo utilizziamo ancora una volta l'albero della ricorrenza:



Nodo generico: $T(\frac{n}{2^i})$

Altezza: $\frac{n}{2^i} < 10 \implies i > \log_2 \frac{n}{10}$

Costo livello generico: $\frac{n}{2^i}$

Numero foglie: 1

Possiamo dunque calcolarci il costo totale del nostro albero:

$\sum_{i=0}^{\log_2 \frac{n}{10}} \frac{n}{2^i} + 1 = n \sum_{i=0}^{\log_2 \frac{1}{10}} (\frac{1}{2})^i + 1 \leq n \sum_{i=0}^{+\infty} (\frac{1}{2})^i + 1 = n(\frac{1}{1-\frac{1}{2}}) \in \Theta(n)$.

Proviamo il bound stimato per induzione per accertarci che sia corretto:

$\exists c > 0, \exists n_0 > 0$ t.c. $\forall n \geq n_0, T(n) \leq cn$

Hp: $T(\frac{n}{2}) \leq c\frac{n}{2}$
$T(n) \leq c\frac{n}{2} + \Theta(n)$
$= c\frac{n}{2} + c_1 n$

Ci stiamo chiendo quindi quando è che:

$c\frac{n}{2} + c_1 n \leq cn$

Abbiamo che la nostra prova è verificata quando:

$c \geq 2c_1$
$\implies \leq cn$ **OK!**

Esercizio 1.11. *Risolvere con l'albero della ricorsione l'equazione di ricorrenza che rappresenta la complessità in tempo del seguente algoritmo:*

```
Procedure Pippo (n)
begin
    if n ≤ 5 then
        return 3
    else
        i = 1
        while i ≤ n do
            for j=1; j ≤ n; j++ do
                i = i+3
        Pippo = Pippo(n/5) + Pippo(n/7)
end
```

Soluzione

Studiamo la complessità del ciclo FOR interno, studiando il valore di $\sum_{j=0}^{n-1} \mathcal{O}(1) \in \Theta(n)$. Oltre al costo del ciclo, ci interessa però sapere il valore che assume la variabile i, la quale viene incrementata ad ogni iterazione di 3 all'interno del ciclo FOR.

$\implies$ avremo che il valore della variabile i sarà $1 + 3n$. Da questo segue che il ciclo WHILE verrà eseguito una sola volta, ovvero dopo una singola iterazione avremo soddisfatto la condizione di arresto. Dunque il costo complessivo dei due cicli è $\Theta(n)$.

Possiamo ora scrivere l'equazione di ricorrenza:

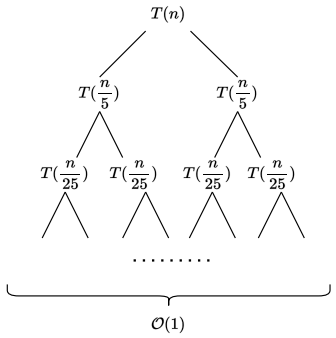
$$T(n) = \begin{cases} T(\frac{n}{5}) + T(\frac{n}{7}) + \Theta(n), & n > 5 \\ \Theta(1), & n \leq 5 \end{cases}$$

L'equazione risultante ci suggerisce che il nostro albero della ricorsione sarà sbilanciato, ovvero i suoi rami avranno dimensioni "irregolari".

hint: per semplificare lo studio della ricorrenza potremmo pensare di maggiorarla definendo una ricorrenza che definisce a sua volta un albero bilanciato.

Definiamo quindi:

$$T'(n) = \begin{cases} 2T'(\frac{n}{5}) + \Theta(n), & n > 5 \\ \Theta(1), & n \leq 5 \end{cases}$$



Nodo generico: $T(\frac{n}{5^i})$

Altezza: $\frac{n}{5^i} \leq 5 \implies i \geq \log_5 \frac{n}{5}$

Costo livello generico: $2^i \frac{n}{5^i}$

Numero foglie: $2^{\log_5 \frac{n}{5}} = n^{\log_5 \frac{2}{5}}$

Possiamo dunque calcolarci il costo totale del nostro albero:

$\sum_{i=0}^{\log_5 \frac{n}{5}} 2^i \frac{n}{5^i} + n^{\log_5 \frac{2}{5}} = n \sum_{i=0}^{\log_5 \frac{n}{5}} (\frac{2}{5})^i + n^{\log_5 \frac{2}{5}} \leq n \sum_{i=0}^{+\infty} (\frac{2}{5})^i + n^{\log_5 \frac{2}{5}} = n(\frac{1}{1-\frac{2}{5}}) + n^{\log_5 \frac{2}{5}}$
$\implies T'(n) \in \Theta(n)$.

Proviamo il bound stimato per induzione per accertarci che sia corretto:

$\exists c > 0, \exists n_0 > 0$ t.c. $\forall n \geq n_0, T(n) \leq cn$

Hp: $T(\frac{n}{5}) \leq c\frac{n}{5}$ and Hp: $T(\frac{n}{7}) \leq c\frac{n}{7}$
$T(n) \leq c\frac{n}{5} + c\frac{n}{7} + \mathcal{O}(n)$
$\leq c\frac{n}{5} + c\frac{n}{7} + c_1 n$
$= \frac{12}{35}cn + c_1 n$

Ci stiamo chiendo quindi quando è che:

$\frac{12}{35}cn + c_1 n \leq cn$

Abbiamo che la nostra prova è verificata quando:

$c \geq \frac{35}{23}c_1$
$\implies \leq cn$ **OK!**

Esercizio 1.12. *Risolvere con l'albero della ricorsione l'equazione di ricorrenza che rappresenta la complessità in tempo del seguente algoritmo:*

```
Procedure Pippo (n)
begin
    if n ≤ 5 then
        return 3
    else
        i = 1; j = 1; a = 0
        while i ≤ n do
            j = 1
            while j ≤ i do
                a = a+1
                j = j+1
            i = 2*i
        Pippo = 2*Pippo(n/3)
end
```

Soluzione

Cominciamo col calcolare il numero di iterazioni che il ciclo WHILE più esterno esegue:

step: 0	1,	2,	3,	...	k
i: 1	2,	4,	8,	...	2^k

Pertanto che abbiamo che:

$2^k > n \implies k > \log_2 n$

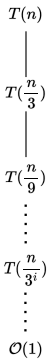
Quindi il costo dei due cicli sarà dato da:

$\sum_{i=1}^{\log_2 n} \sum_{j=1}^{i} \mathcal{O}(1) = \sum_{i=0}^{\log_2 n} i = \frac{\log_2 n(\log_2 +1)}{2} \in \Theta(\log^2 n)$.

Possiamo scrivere la nostra equazione di ricorrenza:

$$T(n) = \begin{cases} T(\frac{n}{3}) + \Theta(\log^2 n), & n > 5 \\ \Theta(1), & n \leq 5 \end{cases}$$

Costruiamo il nostro albero della ricorsione per stimare la complessità:



Nodo generico: $T(\frac{n}{3^i})$

Altezza: $\frac{n}{3^i} \leq 5 \implies i \geq \log_3 \frac{n}{5}$

Costo livello generico: $\log_3^2 \frac{n}{3^i}$

Numero foglie: 1

Possiamo dunque calcolarci il costo totale del nostro albero:

$\sum_{i=0}^{\log_3 \frac{n}{5}} \log_3^2 \frac{n}{3^i} + 1 \leq \sum_{i=0}^{\log_3 \frac{n}{5}} \log_3^2 n + 1 = \log_3^2 n \sum_{i=0}^{\log_3 \frac{n}{5}} \mathcal{O}(1) + 1 \implies T(n) \in \Theta(\log^3 n)$.

Proviamo il bound stimato per induzione per accertarci che sia corretto:

$\exists c > 0, \exists n_0 > 0$ t.c. $\forall n \geq n_0, T(n) \leq c \log^3 n$

HP: $T(\frac{n}{3}) \leq c \log_3^3 \frac{n}{3}$

$$\begin{aligned} T(n) &\leq c \log_3^3 \tfrac{n}{3} + \Theta(\log^2 n) \\ &= c \log_3^3 \tfrac{n}{3} + c_1 \log_3^2 n \\ &= c(\log_3 n - \log_3 3)^3 + c_1 \log_3^2 n \\ &= c(\log_3^3 n - 3\log_3^2 n + 3\log_3 n - 1) + c_1 \log_3^2 n \\ &= c \log_3^3 n \underbrace{-3c \log_3^2 n + 3c \log_3 n - c + c_1 \log_3^2 n}_{\leq 0} \end{aligned}$$

Abbiamo che la nostra prova è verificata quando:

$-3c \log_3^2 n + 3c \log_3 n - c + c_1 \log_3^2 n \leq 0$

$c \geq \frac{c_1 \log_3^2 n}{3 \log_3^2 n + 3 \log_3 - 1} \implies \leq c \log_3^3 n$ **OK!**

Esercizio 1.13. *Risolvere con l'albero della ricorsione l'equazione di ricorrenza che rappre-*

senta la complessità in tempo del seguente algoritmo:

```
Procedure Pippo(n)
begin
    if n ≤ 3 then
        return 3
    else
        for int i=1; i ≤ n − 1; i++ do
            for int j=1; j ≤ n; j++ do
                if i + j ≤ n + 1 then
                    A[i, j] = 0
        Pippo = 4*Pippo(n/2) + Pippo(n/3)

end
```

Soluzione

Cominciamo con l'analizzare il costo computazionale dei due cicli FOR. Abbiamo dunque che:

$\sum_{i=0}^{n-1} \sum_{j=0}^{n} \mathcal{O}(1) = n \sum_{i=0}^{n-1} \mathcal{O}(1) = n(n-1) \in \Theta(n^2)$.

Possiamo ora definire l'equazione di ricorrenza della nostra funzione:

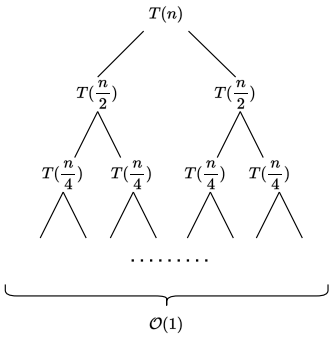
$$T(n) = \begin{cases} T(\frac{n}{2}) + T(\frac{n}{3}) + \Theta(n^2), & n > 3 \\ \Theta(1), & n \leq 3 \end{cases}$$

L'equazione risultante ci suggerisce che il nostro albero della ricorsione sarà sbilanciato, ovvero i suoi rami avranno dimensioni "irregolari".

hint: per semplificare lo studio della ricorrenza potremmo pensare di maggiorarla definendo una ricorrenza che definisce a sua volta un albero bilanciato.

Definiamo quindi:

$$T'(n) = \begin{cases} 2T'(\frac{n}{2}) + \Theta(n^2), & n > 3 \\ \Theta(1), & n \leq 3 \end{cases}$$



Nodo generico: $T(\frac{n}{2^i})$

Altezza: $\frac{n}{2^i} \leq 3 \implies i \geq \log_2 \frac{n}{3}$

Costo livello generico: $2^i(\frac{n}{2^i})^2$

Numero foglie: $2^{\log_2 \frac{n}{3}} = n^{\log_2 \frac{2}{3}}$

Possiamo dunque calcolarci il costo totale del nostro albero:

$\sum_{i=0}^{\log_2 \frac{n}{3}} 2^i(\frac{n}{2^i})^2 + n^{\log_2 \frac{2}{3}} = n^2 \sum_{i=0}^{\log_2 \frac{n}{3}} (\frac{1}{2})^i + n^{\log_2 \frac{2}{3}} \leq n^2 \sum_{i=0}^{+\infty} (\frac{1}{2})^i + n^{\log_2 \frac{2}{3}} = n^2(\frac{1}{1-\frac{1}{2}}) + n^{\log_2 \frac{2}{3}} \implies T'(n) \in \Theta(n^2)$.

Proviamo il bound stimato per induzione per accertarci che sia corretto:

$\exists c > 0, \exists n_0 > 0$ t.c. $\forall n \geq n_0, T(n) \leq cn^2$

Hp: $T(\frac{n}{2}) \leq c(\frac{n}{2})^2$ and Hp: $T(\frac{n}{3}) \leq c(\frac{n}{3})^2$
$T(n) \leq c(\frac{n}{2})^2 + c(\frac{n}{3})^2 + \Theta(n^2)$
$\leq c\frac{n^2}{4} + c\frac{n^2}{9} + c_1 n^2$
$= \frac{13}{36}cn^2 + c_1 n^2$

Ci stiamo chiendo quindi quando è che:

$\frac{13}{36}cn^2 + c_1 n^2 \leq cn^2$

Abbiamo che la nostra prova è verificata quando:

$c \geq \frac{36}{23}c_1$
$\implies \leq cn^2$ **OK!**

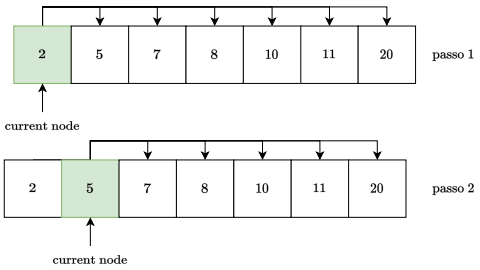
Capitolo 2

Divide et Impera

Esercizio 2.1. *Sia A un array di n interi ordinato e sia dato un intero k. Determinare se esistono due interi in A tali che il loro prodotto sia k. Dare lo pseudocodice dell'algoritmo. Studiare la complessità in tempo della soluzione proposta, e dire se la soluzione proposta è ottima.*

Soluzione

Vediamo come affrontare il problema. La prima idea è quella di utilizzare una strategia **forza bruta**. Questo significa scandire in modo esaustivo tutte le coppie distinte. L'immagine seguente illustra il procedimento mostra alcune iterazioni dell'algoritmo. In verde abbiamo l'elemento corrente puntato dall'indice j, mentre le frecce ci indicano con chi verrà moltiplicato il valore $A[j]$.



Scriviamolo lo pseudocodice:

```
Procedure brute-force (Array A, Int k)
begin
    for (j := 0; j++; j ≤ A.length − 2)
    {
        for (z := j + 1; z++; z + 1 ≤ A.length − 1)
        {
            if (A[i] ∗ A[j] = k)
                return 1
        }
    }
    return 0
end
```

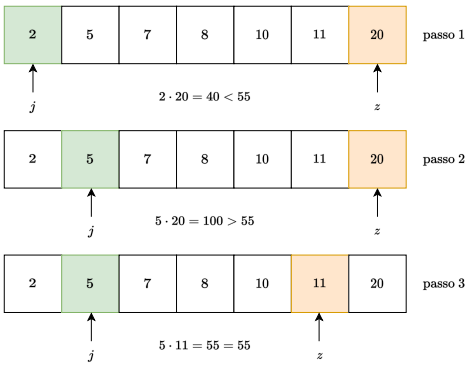
Analizziamo la complessità dell'algoritmo. L'elemento 1 viene moltiplicato con gli elementi $2, 3, \ldots, n$ ($n-1$ confronti). L'elemento 2 viene moltiplicato con gli elementi $3, 4 \ldots, n$ ($n-2$

confronti). Il caso peggiore è quando la coppia di elementi non esiste all'interno dell'array. In questo caso l'algoritmo effettua $(n-1)+(n-2)+(n-3)...+1 \in \mathcal{O}(n^2)$ confronti. Per quanto riguarda lo spazio occupato, non abbiamo utilizzato alcuna struttura dati supporto e quindi abbiamo consumato uno spazio che appartiene ad $\mathcal{O}(1)$.

Ovviamente questa soluzione **non è ottima** perchè lo spazio delle soluzioni è quadratico ma l'array è ordinato e alcuni confronti espliciti si possono evitare. Possiamo realizzare una funzione che produce lo stesso risultato ma con una complessità computazionale inferiore **sfruttando il fatto che array è ordinato**. L'intuizione è la seguente:

- Fissiamo due indici, il primo (j) che punta al primo elemento dell'array, mentre il secondo (z) che punta all'ultimo elemento
- Moltiplichiamo il valore che sta in posizione j con quello che sta in posizione z.
  - Se $A[j] \times A[z] = k$, allora abbiamo finito e ritorniamo TRUE
  - se $A[j] \times A[z] > k$, allora decrementiamo l'indice z ($z = z - 1$). Ossia, possiamo escludere $A[z]$ perchè nessun altro elemento moltiplicato per $A[z]$ sarà pari a k. Infatti, qualunque altro elemento $A[j']$ con $j' > j$ è maggiore o uguale a $A[j]$ e quindi $A[j'] \times A[z] > A[j] \times A[z] > k$.
  - se $A[j] \times A[z] < k$, allora incrementiamo l'indice j ($j = j + 1$). Ossia, possiamo escludere $A[j]$ perchè nessun altro elemento moltiplicato per $A[j]$ sarà pari a k. Infatti, qualunque altro elemento $A[z']$ con $z' < z$ è minore o uguale a $A[z]$ e quindi $A[z'] \times A[j] < A[z'] \times A[j] < k$.
- Ripetiamo finchè $j = z$.

La figura seguente illustra un esempio in cui dato l'array $A[1, \ldots, n]$ bisogna trovare i valori il cui prodotto è $k = 55$.



Scriviamo lo pseudocodice:

```
Procedure Indici (Array A, Int k)
begin
    z = 0, j = A.length − 1\\
    while (j < z)
    {
        if (A[j] ∗ A[z] = k)
            return 1
        else
        {
```

```
            if (A[j] * A[z] < k)
                j = j + 1
            else
                z = z - 1
        }
    }
    return 0
end
```

Per determinare il costo dell'algoritmo dobbiamo studiare il caso pessimo, ovvero quando l'array $A[\ldots]$ non contiene la coppia il cui prodotto è uguale a k In questo caso, i due indici convergeranno sulla stessa cella. Osserviamo che ad ogni iterazione i due indici puntano celle differenti, e solo durante l'ultima iterazione si sovrappongono. Ad ogni iterazione muoviamo uno solo dei due indici. Segue che saranno effettuate al più $\Theta(n)$ spostamenti prima di terminare. **Ottimalità:** dato che l'array ha dimensione n e l'algoritmo converge con un costo $\Theta(n)$, questo sancisce l'ottimalità dell'algoritmo per la dimensione dell'input del problema.

Esercizio 2.2. *Data una matrice M di dimensione $m \times m$ i cui elementi sono uguali a 0 o 1, ed ordinata in modo che visitando la matrice per righe con indice da 1 a n, e ogni riga da sinistra verso destra, si incontrano prima tutti gli 0's e poi gli 1's. Dare un algoritmo per trovare la posizione del primo 1 in M.*

- *Descrivere l'algoritmo a parole*
- *Dare lo pseudocodice dell'algoritmo*
- *Studiare la complessità dell'algoritmo proposto e discutere l'ottimalita*

Esempio: Input; M=4 x 4

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Output: $(3, 2)$

Soluzione

Studio della Complessità Intrinseca del Problema

Nella matrice vi sono $m \times m = m^2$ elementi, quindi esiste una soluzione (Upper Bound) con complessità $O(m^2)$; infatti leggendoli tutti dal primo all'ultimo, e fermandoci al primo 1 risolviamo il problema. Nel caso pessimo, il primo 1 è nell'ultima posizione, e sono richieste m^2 operazioni per trovarlo. Tuttavia la scansione di tutti i dati in input non è necessaria perchè i dati sono ordinati. Se visitando a caso una posizione trovo uno 0 so, senza visitarli, che tutti gli elementi che precedono sono 0. Similmente se visitando a caso una posizione trovo 1 so, senza visitarli, che tutti gli elementi che seguono sono 1. Pertanto non possiamo dire che per decidere se una posizione è soluzione essa debba essere visitata. Quindi la tecnica dell'input/output, che darebbe un lower bound di m^2 non si applica in questo caso che gli elementi sono ordinati.

Il fatto che gli elementi sono ordinati però ci dà un vantaggio e ci permette di ottenere la complessità in tempo derivata dall'albero delle decisioni. Infatti, lo spazio delle soluzioni,

ossia le posizioni candidate a contenere il primo 1 sono m^2, ossia tutte le celle della matrice. L'albero delle decisioni deve quindi scegliere fra una di queste posizioni e deve perciò avere m^2 foglie. Poichè l'albero delle decisioni di altezza minima che alloca m^2 foglie ha altezza $\log_2 m^2 = 2\log_2 m$, abbiamo una complessità intrinseca del problema pari a $\Omega(\log_2 m)$.

Avendo chiaro il range di complessità in cui cercare un algoritmo di risoluzione, cerchiamolo.

Ricerca di un Algoritmo Risolutivo

Oss: Nel cercare la soluzione dobbiamo tenere conto che la matrice è $m \times m$ e che vi è un ordinamento degli elementi al suo interno.

Algoritmo a Forza Bruta: Consiste nel leggere gli elementi da quello in alto a sinistra a quello in basso a destra procedendo da sinistra verso destra una riga alla volta. Come detto nello studio della complessità, avendo m^2 elementi, avrebbe costo $O(m^2)$; quindi non ottimo; infatti non sfrutta l'ordinamento degli elementi, né il fatto che la matrice in cui sono memorizzati sia $m \times m$.

Algoritmo a Balzi: Si legge l'elemento centrale della matrice, se è uno 0, si prende in considerazione la parte della matrice inferiore a quella posizione, ripetiamo quindi la procedura. Se l'elemento centrale della sequenza è un 1, si prende in considerazione la parte della matrice superiore a quella posizione, altrimenti se è 0 la matrice inferiore.

```
Procedure RabbitAlgorithm (Matrix M, int StartRow, int EndRow, int StartCol,
    int EndCol, int dim)
begin
    int mRow = ⌊(EndRow+StartRow)/2⌋ // Troviamo le coordinate del valore centrale della sequenza di valori
    int mCol = ⌊(EndCol+StartCol)/2⌋
    // Caso Base
    if (StartRow == EndRow \And EndCol − StartCol + 1 ≤ 3)
    {
        //Procediamo in avanti dall'alto
        for (i = StartCol; i ≤ EndCol; i++)
        {
            if (M[StartRow, i] == 0 && M[StartRow, i + 1] == 1)
                return (StartRow, i+1)
        }
        if (StartRow == EndRow − 1 && M[StartRow, dim] == 0)
            return EndRow, RicercaBinaria ({M[Endrow, ∗], 1, EndColumn, 1})
        else
            if (StartRow == EndRow − 1 && M[StartRow, dim] == 1)
                return StartRow, RicercaBinaria(M[StartRow, ∗], StartColumn, dim, 1)
        //Caso Ricorsivo
        if (StartRow < EndRow − 1)
        {
            if (M [mRow, mCol] == 0)
                //Aggiorniamo le coordinate iniziali con quelle medie
                return RabbitAlgorithm (M, mRow, EndRow, mCol, EndCol, dim)

            else
                //Abbiamo un 1, aggiorniamo le coordinate finali con quelle medie
                return{RabbitAlgorithm (M, StartRow, mRow, StartCol, mCol, dim)}
        }
end
```

Nella chiamata, passeremo all'algoritmo i valori $(1, 1)$ ed (m, m) come coordinate iniziali. Quindi $StartRow = 1$, $StartCol = 1$, $EndRow = m$, $EndCol = m$; metteremo anche

$dim = m$.
La procedura *RicercaBinaria*(A,StartArray,EndArray,key) ricerca nell'array ordinato A di dimensioni $EndArray - StartArray + 1$ la chiave key.

Studio Complessità dell'Algoritmo Proposto

L'andare a "dimezzare" la dimensione della matrice ad ogni passo fino ad arrivare a trovare le due righe in cui si trova il primo 1 ci costa $O(\log m^2)$. Inoltre si esegue una ricerca binaria su un array che corrisponde ad una porzione di una riga, ed ha perciò complessità $O(\log_2 m)$.

Abbiamo quindi un costo finale di $O(\log m)$ che, sulla base dello studio della complessità effettuato è ottimo e pertanto la sua complessità è $\Theta(logm)$ perchè upper e lower bound coincidono in ordine di grandezza.

Soluzione informata Esiste una soluzione ottima che si scrive in due righe di pesudocodice. Notiamo che la colonna m della matrice è ordinata ed ogni riga della matrice è ordinata. La riga in cui si trova il primo 1 si trova applicando la ricerca binaria alla colonna m della matrice. Trovata la riga, si applica una ricerca binaria nella riga stessa per trovare il primo 1.

Pseudocodice

```
Procedure fast (Matrix M, int StartRow, int EndRow, int StartCol, int EndCol
    , int dim)
begin
    OutRow = RicercaBinaria(M[*,Endcolumn],1,dim,1)
    OutColumn = RicercaBinaria(M[OutRow,*],1,dim,1)
    return (OutRow,OutColumn)
end
```

Al fine di non copiare la colonna m e la riga OutRow in due array è necessario riscrivere il codice della RicercaBinaria leggendo le opportune posizioni della matrice M. Tuttavia per brevità e per facilitare la comprensione dell'algoritmo presentiamo l'algoritmo smart come l'invocazione di due RicercaBinarie. Entrambi gli algoritmi RabbitAlgorithm e SmartAlgorithm sono ottimi in tempo.

Esercizio 2.3. *Dati due array A e B ordinati in senso crescente di dimensione n_1 and n_2 rispettivamente, si scriva una procedura per costruire l'array C di dimensione $n_1 + n_2$ ordinato in senso crescente. Dare lo pseudocodice dell'algoritmo. Studiare la complessità in tempo dell'algoritmo proposto. Discutere l'ottimalità della soluzione proposta.*

Soluzione

Si scorrono i due array ordinati in senso crescente e si mette ogni volta in quello risultante il valore minore trovato per poi aggiornare gli indici; in questo modo C è ancora in senso crescente. Se uno dei due array è terminato si continuano a scrivere solo i valori dell'altro.

```
Procedure merge (Array A,  Array B, Array C)
begin
    // Abbiamo i 3 array con rispettive dimensioni, C é giá creato
    i_a := 1 // inizializziamo gli indici degli array ad 1
    i_b := 1
    i_c := 1

    while(i_c ≤ (n_1 + n_2))
    {
        if(i_a ≤ n_1 && i_b ≤ n_2)
```

```
        {
            if (A[i_a] < B[i_b])
                C[i_c] := A[i_a], i_c++, i_a++
            else
                C[i_c] := B[i_b], i_c++, i_b++
        }
        else
        {
            if (i_a > n_1)
                C[i_c] := B[i_b], i_c++, i_b++
            else
                C[i_c] := A[i_a], i_c++, i_a++
        }
    }
end
```

Complessità: Supponendo $n = n_1 + n_2$, vediamo come il ciclo while esegue n operazioni ed al suo interno vi sono solo operazioni di costo $O(1)$, quindi l'algoritmo ha costo $O(n)$. Questo costo è ottimo, infatti rispecchia quella che è la dimensione dell'input e dell'output; non è possibile avere costo inferiore a tali dimensioni.