Esempi di esercizi di esame sulla complessità

Gabriele Frassi

A.A 2019-2020 - Secondo semestre

Regole (riprese dalla lezione dedicata)

Divide et impera

$$\begin{cases} T(n) = d & n \leq m \\ T(n) = hn^k + aT(n/b) & n > m \end{cases}$$

con $h > 0$

Teorema

- Se $a < b^k, T(n) \in O(n^k)$
- Se $a = b^k, T(n) \in O(n^k \log n)$
- Se $a > b^k, T(n) \in O(n^{\log_b a})$

Relazioni lineari

$$\begin{cases} T(0) = d \\ T(n) = bn^k + a_1T(n-1) + a_2T(n-2) + \cdots + a_rT(n-r) \end{cases}$$

Due chiamate ricorsive, complessità esponenziale:

$$a_1 = 1, a_i = 0 \; i > 1$$

Caso elementare

$$\begin{cases} T(0) = d \\ T(n) = bn^k + T(n-1) \end{cases}$$

$T(n) \in O(n^{k+1})$

Raccolta

Fattoriale ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

selectionSort ($O(n^2)$)

$$\begin{cases} T(1) = a \\ T(n) = bn + T(n-1) \end{cases}$$

quickSort ($O(n \log n)$)

$$\begin{cases} T(1) = a \\ T(n) = bn + T(k) + T(n-k) \end{cases}$$

Caso peggiore ($O(n)$)

$$\begin{cases} T(1) = a \\ T(n) = bn + a + T(n-1) \end{cases}$$

Caso migliore ($O(n \log n)$

$$\begin{cases} T(1) = a \\ T(n) = bn + 2T(n/2) \end{cases}$$

Ricerca lineare ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

Ricerca binaria ($O(\log n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n/2) \end{cases}$$

Ricerca pseudo-binaria ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + 2T(n/2) \end{cases}$$

Torre di Hanoi ($O(2^n)$)

$$\begin{cases} T(1) = a \\ T(n) = b + 2T(n-1) \end{cases}$$

Serie di Fibonacci

Fibonacci cattivo ($O(2^n)$)

$$\begin{cases} T(0) = T(1) = a \\ T(n) = b + T(n-1) + T(n-2) \end{cases}$$

Fibonacci buono ($O(n)$)

$$\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases}$$

mergeSort ($O(n \log n)$)

Funzione principale ($O(n \log n)$)

$$\begin{cases} T(0) = T(1) = d \\ T(n) = bn + T(n/2) \end{cases}$$

split ($O(n)$)

$$\begin{cases} T(0) = T(1) = d \\ T(n) = b + T(n-2) \end{cases}$$

merge ($O(n)$)

$$\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases}$$

Alberi

Visita di un albero ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n_s) + T(n_d) \ \ con\ n_s + n_d = n - 1\ con\ n > 0 \end{cases}$$

Forma da noi calcolabile valida per un albero bilanciato

$$\begin{cases} T(0) = a \\ T(n) = b + 2T((n-1)/2) \end{cases}$$

Visite in funzione dei livelli ($O(2^k)$)

$$\begin{cases} T(0) = a \\ T(k) = b + 2T(k-1) \end{cases}$$

Ricerca in un ABR $O(\log n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(k) \quad k < n \end{cases}$$

Caso peggiore ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

Caso migliore ($O(\log n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n/2) \end{cases}$$

Moltiplicazione veloce ($O(n^{\log_2 3})$)

$$\begin{cases} T(1) = d \\ T(n) = bn + 3T(n/2) \end{cases}$$

[Osservazione per esercizi] Somma dei primi n numeri

Negli esercizi possono capitare cicli del seguente tipo:

```
int a = 0;

for(int i = 0; i <= n; i++)
        a += i;
```

In questo caso abbiamo la somma dei primi n numeri. La cosa è rilevante se dobbiamo calcolare il risultato di una funzione. Troviamo che

$$O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2+n}{2}\right) = O(n^2)$$

[Esempio 1] Esercizio sulla complessità

Calcolare la complessità dell'istruzione

```
for (int i=0; i<=f(t); i++) cout << g(t->left) + g(t->right);
```

in funzione del numero di nodi di t. Indicare per esteso le relazioni di ricorrenza.
Supporre che t sia un albero binario bilanciato e le funzioni **f** e **g** siano definite come segue:

```
int f(Node* t) {
   if (! t)
     return 1;
  return g(t->left) + f(t->left) +
f(t->right);
}
```

```
int g(Node * t) {
  if (!t) return 0;
  int a = g(t->left);
  int b = g(t->right);
  return 1 +2a + 2b;
}
```

Spiegazione

- Troviamo le relazioni di ricorrenza di g (necessaria per trovare la relazione di f):
  - Complessità:
    - Il caso base (0 elementi, albero vuoto) ha complessità costante $O(1)$
    - Ogni volta ho due chiamate di funzione: una per il sottoalbero sinistro e una per il sottoalbero destro. Considerando che l'albero è bilanciato ogni chiamata coinvolge metà degli n elementi.
    - Oltre a casi base e chiamate ricorsive non ho altre istruzioni rilevanti che potrebbero aumentare la complessità
    - Ottengo

      $$\begin{cases} T_g(0) = d \\ T_g(n) = c + 2T_g(n/2) \end{cases}$$

      Segue $T_g \in O(n)$ con le regole che conosciamo.
  - Risultato:
    - Con il caso base (0 elementi, albero vuoto) ho risultato 0.
    - Analizzo il return: ogni volta restituisco una somma dove sono coinvolti i risultati di due chiamate ricorsive. Entrambi i risultati, nella somma restituita, sono moltiplicati per due. Entrambi i risultati sono ottenuti coinvolgendo metà degli n elementi.

    * Ottengo

$$\begin{cases} R_g(0) = 0 \\ R_g(n) = c + 4R_g(n/2) \end{cases}$$

    Segue $R_g \in O(n^2)$ con le regole che conosciamo.

- Troviamo le relazioni di ricorrenza di f (adesso abbiamo i dati di g che ci servono):
  - Complessità:
    * Il caso base (0 elementi, albero vuoto) ha complessità costante $O(1)$
    * Ho due chiamate ricorsive, ciascuna per un sottoalbero. Considerando che l'albero è bilanciato ciascuna delle chiamate coinvolge metà degli n elementi.
    * Oltre alle due richiamate ricorsive ho una chiamata della funzione f. Sappiamo che $T_g \in O(n)$.
    * Ottengo

$$\begin{cases} T_f(0) = d \\ T_f(n) = cn + 2T_f(n/2) \end{cases}$$

    Segue $T_f \in O(n \log n)$ con le regole che conosciamo.
  - Risultato:
    * Con il caso base (0 elementi, albero vuoto) restituisco 1
    * Analizzo il return: ho due risultati ottenuti da chiamate ricorsive. Entrambi i risultati, nella somma restituita, sono ottenuti coinvolgendo metà degli n elementi.
    * Nella somma ho un ulteriore elemento: un risultato ottenuto dalla funzione g. Abbiamo visto che $R_g \in O(n^2)$.
    * Ottengo:

$$\begin{cases} R_f(0) = 1 \\ R_f(n) = cn^2 + 2R_f(n/2) \end{cases}$$

    Segue $R_f(n) \in O(n^2)$ con le regole che conosciamo.

- Riprendiamo il frammento iniziale:
  - Il numero di iterazioni del for è $O(n^2)$
  - La complessità di una singola iterazione è:

$$O(n) + O(n) + O(n \log n) = O(n \log n)$$

    Ho considerato anche la complessità di f (ogni volta che verifico se la condizione del for è true eseguo la funzione f)
  - Complessità del for:

$$O(n \log n) * O(n^2) = O(n^3 \log n)$$

[Esempio 2] Esercizio sulla complessità

Calcolare la complessità del blocco (indicando le relazioni di ricorrenza di tempo e risultato per ogni funzione) in funzione di n:

```
{
  int a = 0;
  for (int i=0; i <= g(n)/n; i++)
     a += f(n);
}
```

con le funzioni **f** e **g** definite come segue:

```
int f(int x) {
   if (x<=1)
     return 1;
   cout << f(x/3) + f(x/3);
   return f(x/3) + 1;
}
```

```
int g(int x) {
  int a=0;
  for (int i=0; i <= f(x); i++)
     a++;
  for (i=0; i <= 2*f(x); i++)
     a+=i;
  return a;
}
```

Indicare per esteso le relazioni di ricorrenza e, per ogni comando ripetitivo, il numero di iterazioni e la complessità dell'iterazione singola.

Spiegazione

- Troviamo le relazioni di ricorrenza di f (necessario dopo per g):
  - Complessità:
    - Il caso base si ha con $n \leq 1$: tenendo conto che i valori possibili sono maggiori o uguali a 0 individuo che il caso base si ha con $0, 1$.
    - Il cout ha complessità $O(1)$
    - Ho tre chiamate ricorsive, ciascuna con argomento $x/3$
    - Ottengo
    $$\begin{cases} T_f(0) = T_f(1) = d \\ T_f(n) = c + 3T_f(n/3) \end{cases}$$
    Segue $T_f(n) \in O(n)$ con le regole che conosciamo.
  - Risultato:
    - Il caso base si ha con $n \leq 1$: il valore restituito è 1. Ciò avviene coi valori $0, 1$
    - Analizzo il return: ogni volta restituisco la somma tra 1 e il risultato di una chiamata ricorsiva (con argomento $n/3$)
    - Ottengo
    $$\begin{cases} R_f(0) = R_f(1) = 1 \\ R_f(n) = 1 + R_f(n/3) \end{cases}$$
- Troviamo complessità e risultato di g (funzione non ricorsiva):
  - Complessità:
    - Gli unici elementi rilevanti sono i due for.
    - Il post-incremento ha complessità $O(1)$, ma devo tenere conto che ogni volta viene chiamata la funzione f, con complessità $O(n)$. Il numero di iterazioni è il risultato di f, cioè $O(\log n)$. Segue che la prima parte avrà complessità $O(n \log n)$.

- * Il secondo for consiste nella somma dei primi $2\log n$ numeri[1]. La somma ha complessità $O(1)$, ma anche qua f viene eseguita ogni volta con complessità $O(n)$. Il numero di iterazioni è $O(2\log n) = O(\log n)$. Segue che il secondo for avrà complessità $O(n\log n)$.
  * Ottengo

$$T_g \in O(n\log n)$$

- Risultato:
  * La funzione restituisce l'intero a, inizialmente uguale a 0.
  * Nel primo for incremento a $\log n$ volte.
  * Nel secondo for sommo a quanto già trovato i primi $2\log n$ numeri.
  * Ottengo

$$O(\log n)+\left(\frac{2\log n(2\log n+1)}{2}\right) = O(\log n)+O\left(\frac{4(\log n)^2+2\log n}{2}\right) = O(\log n)+O(2(\log n)^2+\log n) = O((\log n)^2)$$

Quindi $R_g \in O((\log n)^2)$

- Concludiamo col for del blocco:
  - Il numero di iterazioni consiste nel risultato di g fratto n, cioè $\frac{g(n)}{n} = \frac{(\log n)^2}{n}$
  - La singola iterazione ha la seguente complessità

$$O(n) + O(n\log n) = O(n\log n)$$

considero sia la somma ad a che l'esecuzione, ogni volta della funzione g

  - La complessità totale sarà

$$O(n\log n) * O\left(\frac{(\log n)^2}{n}\right) = O((\log n)^3)$$

[1] Non sono d'accordo che il for consiste in 2 volte la somma dei primi $\log n$ come scritto dalla prof.

$$2\frac{\log n(\log n+1)}{2} = \frac{2\log n(2\log n+1)}{2}$$

I membri non diventano uguali semplificando. Il risultato di R_g, comunque sia, non cambia.

[Esempio 3] Esercizio sulla complessità

Calcolare la complessità del blocco (indicando le relazioni di ricorrenza di tempo e risultato per ogni funzione) in funzione del numero di nodi dell'albero t,

a) supponendo che t sia quasi bilanciato.
b) supponendo che t sia completamente sbilanciato

Indicare per esteso numero di iterazioni e complessità di ogni iterazione per i comandi ripetitivi.

```
{
  int a = 0;
  for (int i=0; i <= g(t)*f(t); i++)
     a += Nodes(t);
}
```

Le funzioni **f** e **g** sono definite come segue:

`int f(Node* tree) {`
`  if (!tree) return 1;`
`  int x=0;`
`  for (int i=1;i<= Nodes(tree);i++)`
`  x+=i;`
`  int b = 4*f(tree->left);`
`  return x+b;`
`}`
	`int g(Node * tree) {`
	`  if (!tree) return 1;`
	`  int a=0;`
	`  for (int i=1;i<=f(tree)/Nodes(tree);i++)`
	`    { a++;`
	`      cout << a;`
	`    }`
	`  return 3;`
	`  }`

Spiegazione

- Consideriamo, in entrambi i casi, una funzione *Nodes* con complessità lineare. Tutte le volte scorro l'albero per individuare il numero dei suoi nodi.
- Albero t quasi bilanciato:
  - Un albero quasi bilanciato presenta le proprietà di un albero bilanciato fino al penultimo livello: abbiamo una situazione di sostanziale equilibrio che ci permette di gestire questi alberi come se fossero bilanciati.
  - Troviamo le relazioni di ricorrenza per f:
    * Complessità:
      · Il caso base ha complessità $O(1)$
      · L'iterazione del for ha complessità $O(n)$: l'istruzione di somma è $O(1)$ ma devo tener conto della chiamata della funzione *Nodes*. Il numero di iterazioni è il risultato di *Nodes*, cioè $O(n)$. Segue che il for avrà complessità $O(n^2)$.
      · Ho una chiamata ricorsiva di f, che coinvolge la metà degli n elementi.
      · Ottengo
      $$\begin{cases} T_f(0) = d \\ T_f(n) = dn^2 + T_f(n/2) \end{cases}$$
      Segue $T_f(n) \in O(n^2)$ con le regole che conosciamo.
    * Risultato:
      · Il caso base restituisce 1
      · Il for aggiorna il valore di x, che inizialmente è uguale a 0. Sommo i primi n numeri (nella condizione del for abbiamo *Nodes*, che restituisce n).
      $$O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2+n}{2}\right) = O(n^2)$$
      · Nel return ho anche b, che consiste nel risultato di una chiamata ricorsiva di f moltiplicato per quattro. Coinvolgo, nel calcolo del risultato, la metà di n elementi.

    - Ottengo
      $$\begin{cases} R_f(0) = 1 \\ R_f(n) = dn^2 + 4R_f(n/2) \end{cases}$$
      Segue $R_f(n) \in O(n^2 \log n)$ con le regole che conosciamo.
  - Troviamo complessità e risultato di g (funzione non ricorsiva):
    * Complessità:
      - Se l'albero è vuoto ho complessità $O(1)$
      - L'unico elemento che determina la complessità è il for.
      - Le istruzioni nel blocco del for hanno complessità $O(1)$. Nel calcolo della complessità dell'iterazione devo tenere conto anche delle chiamate di f e *Nodes* nella condizione del for. Ottengo
        $$O(1) + O(n) + O(n^2) = O(n^2)$$
      - Il numero di iterazioni del for si ottiene dividendo il risultato di f per il risultato di *Nodes*, cioè
        $$O\left(\frac{n^2 \log n}{n}\right) = O(n \log n)$$
      - Segue la complessità del for
        $$O(n \log n) * O(n^2) = O(n^3 \log n)$$
        quindi $T_g(n) \in O(n^3 \log n)$
    * Risultato:
      - Il risultato del caso base è 1
      - Il for aggiorna il valore di a, tuttavia è assolutamente ininfluente nel valore restituito dalla funzione.
      - La funzione restituisce ogni volta 3.
      - Ottengo $R_g \in O(1)$
  - Concludo col for del blocco:
    * Il numero di iterazioni del blocco si ottiene moltiplicando il risultato di f per il risultato di g
      $$O(1) * O(n^2 \log n) = O(n^2 \log n)$$
    * La complessità dell'iterazione è data dalla chiamata di *Nodes* nel blocco e dalle chiamate di g ed f nella condizione
      $$O(n) + O(n^2) + O(n^3 \log n) = O(n^3 \log n)$$
    * La complessità del for si ottiene moltiplicando quanto detto prima
      $$O(n^2 \log n) * O(n^3 \log n) = O(n^5 (\log n)^2)$$

- Albero t completamente sbilanciato:
  - L'albero assume i tratti di una lista e la complessità della visita viene degradata diventando lineare. I ragionamenti sono simili a quelli fatti prima: cambiano gli argomenti delle chiamate ricorsive e i risultati delle funzioni.
  - In riferimento a f: abbiamo le stesse relazioni di ricorrenza, cambiano solo gli argomenti ($n-1$ invece di $n/2$)

- * Complessità:

$$\begin{cases} T_f(0) = d \\ T_f(n) = dn^2 + T_f(n-1) \end{cases}$$

  Segue $T_f(n) \in O(n^3)$ con le regole che conosciamo.

- * Risultato:

$$\begin{cases} R_f(0) = 1 \\ R_f(n) = dn^2 + 4R_f(n-1) \end{cases}$$

  Qui addirittura abbiamo $R_f(n) \in 4^n$.

– In riferimento a g:
  - * Complessità:
    - · Cambia il numero di iterazioni del for (è cambiato il risultato della funzione f): $O(4^n/n)$.
    - · La complessità dell'iterazione sarà la seguente

$$O(1) + O(n^3) + O(n) = O(n^3)$$

    - · Segue la complessità del for, che è R_g

$$O(n^3) * O(4^n/n) = O(n^2 4^n)$$

  - * Risultato: $O(1)$ come prima.

– Concludiamo col for:
  - * Il numero di iterazioni del for è il prodotto tra i risultati di g ed f

$$O(4^n) * O(1) = O(4^n)$$

  - * La complessità dell'iterazione sarà

$$O(n) + O(n^3) + O(n^2 4^n) = O(n^2 4^n)$$

  - * Segue la complessità del for del blocco

$$O(4^n) * O(n^2 4^n) = O(n^2 (4^n)^2)$$

[Esempio 4] Esercizio sulla complessità

Calcolare la complessità in funzione di **n**>0 dell'istruzione

`y=g(f(n));`

con le funzioni **f** e **g** definite come segue:

```
int f(int x) {
   if (x<=1) return 1;
   int b=0, i, j, c;
   for (i=1; i<=x; i++) b+=i;
   c = b*b;
   for (j=1; j<=c; j++) b+=j;
   return b + f(x-1);
}
```

```
int g(int x) {
  if (x<=1) return 10;
  int a=0;
  for (int i=0; i<f(x); i++)
      a++;
   return a+2*g(x/2);
}
```

Indicare le eventuali relazioni di ricorrenza e spiegare brevemente il calcolo della complessità dei cicli.

Spiegazione

- Dobbiamo trovare la complessità di una funzione composta, precisamente $y = g(f(n))$. Dovremo considerare la complessità di f e di g, ma anche il risultato di f, che costituirà argomento della funzione g!
- Troviamo le relazioni di ricorrenza per f:
  - Complessità:
    - Abbiamo due for che provocano un incremento della complessità
    - Il primo somma a b, inizialmente uguale a 0, i primi n numeri. Avrò complessità $O(n)$ (numero di iterazioni) e risultato $O\left(\frac{n(n+1)}{2}\right) = O(n^2)$.
    - SuccessivameNTE moltiplico il risultato con se stesso, ottengo

      $$O(n^2) * O(n^2) = O(n^4)$$

    - Il secondo for somma a b, già uguale alla somma dei primi n numeri, la somma dei primi n^4 numeri. La complessità del for è $O(n^4)$.
    - Ho una chiamata ricorsiva lineare $(n - 1)$
    - Ottengo la seguente relazione

      $$\begin{cases} T_f(0) = 0 \\ T_f(n) = n^4 + T_f(n-1) \end{cases}$$

      con le regole che conosciamo otteniamo $O(n^{4+1}) = O(n^5)$
  - Risultato:
    - Considero che restituiamo la somma dei risultati dei due for più il risultato restituito dalla funzione con argomento $n - 1$
    - Nel primo for sommiamo i primi n numeri. Come già detto il risultato di questo for è

      $$O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

    - Nel secondo for sommiamo i primi n^4 numeri, quindi otteniamo come risultato

      $$O\left(\frac{n^4(n^4+1)}{2}\right) = O(n^8)$$

    - Il risultato dei for sarà $O(n^2) + O(n^8) = O(n^8)$

        * Ottengo la seguente relazione

$$\begin{cases} R_f(0) = 1 \\ R_f(n) = n^8 + R_f(n-1) \end{cases}$$

con le regole che conosciamo otteniamo $O(n^{8+1}) = O(n^9)$

- Troviamo le relazioni di ricorrenza per g:
    - Complessità:
        * Abbiamo un for che incrementa la complessità.
        * Il body del for ha complessità costante $O(1)$, tuttavia nella condizione troviamo una chiamata alla funzione f.
        * Sappiamo che la funzione f ha complessità $O(n^5)$ (viene eseguita ogni volta che si verifica la condizione) e restituisce risultato $O(n^9)$.
        * Segue che avrò $O(n^9)$ iterazioni, quindi il for avrà complessità

$$O(n^9) * O(n^5) = O(n^{14})$$

        * Ottengo la seguente relazione

$$\begin{cases} T_g(0) = a \\ T_g(n) = n^{14} + T_g(n/2) \end{cases}$$

con le regole che conosciamo otteniamo $O(n^{14})$

    - Il risultato non ci interessa
- Concludiamo:
    - Sommo la complessità di f e quella di g, ponendo come argomento di g n^9, cioè il risultato di f.

$$T_f(n) + T_g(n^9) = O(n^5) + O({n^9}^{14}) = O(n^{126})$$