Appunti Algoritmi e Strutture Dati

Luca Seggiani

13 Marzo 2024

1 Algoritmi di teoria dei numeri

Studiamo adesso una categoria di algoritmi la cui complessità è calcolata prendendo come misura il numero di cifre che compongono il numero. Ad esempio:

- L'addizione ha complessità $O(n)$;
- La moltiplicazione ha complessità $O(n^2)$.

Vediamo ad esempio:

Moltiplicazione veloce fra interi non negativi

Ogni numero A di n cifre può essere visto come:

$$A = A_s 10^{\frac{n}{2}} + A_d$$

A questo punto il prodotto AB sarà:

$$AB = A_s B_s 10^{\frac{n}{2}} + A_d B_d$$

che può essere riscritto come:

$$AB = A_s B_S 10^n + ((A_s + A_d)(B_s + B_d) - A_s B_s - A_d B_d) 10^{\frac{n}{2}} + A_d B_d$$

dove compaiono tre moltiplicazioni fra interi non con n ma con $\frac{n}{2}$ cifre. In codice, questo si tradurrà in:

```
int mult(int A, int B, int n) {
  if(n = 1) return A * B;
  else {
    As = parte sinistra di A; Ad = parte destra di A;
    Bs = parte sinistra di B, Bd = parte destra di B;
    int x_1 = As + Ad; int x_2 = Bs + Bd;
    int y_1 = mult(x_1, x_2, n / 2);
```

```
        int y_2 = mult(A, B, n / 2);
        int y_3 = mult(Ad, Bd, n / 2);
        int s_1 = left_shift(y_2, n);
        int s_2 = left_shift(t_1 - t_2- t_3, n / 2);
        return s_1 + s_2 + y_3;
    }
}
```

Dove l'operazione left_shift(h, n) scorre h di n posti a sinstra, introducento n 0 ($\times 10^n$). Abbiamo tre chiamate ricorsive alla funzione mult, ciascuna su una dimensione di istanza di $\frac{n}{2}$. Da qui la relazione di istanza:

$$T(1) = d, \quad T(n) = hn + 3T(\frac{n}{2})$$

Da cui otteniamo che $T \in O(n^{\log_2 3}) = O(n^{1.59})$.

Relazioni lineari

Relazioni lineari in forma:

$$T(0) = d, \quad T(n) = bn^k + a_1T(n-1) + a_2T(n-2) + a_rT(n-r)$$

sono polinomiali soltanto se esiste al più una sola chiamata ricorsiva ($a_i = 1$ e $a_j = 0$ con $j \neq i$). Applichiamo questo teorema al problema del calcolo della serie di Fibonacci. Ricordiamo la classica formulazione ricorsiva:

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad (0, 1, 3, 5, 8, 13...)$$

da cui si ricava:

$$\phi = \lim_{n \to +\infty} \frac{f_n}{f_{n-1}} = 1.618...$$

nota come sezione aurea (o rapporto aureo). Cerchiamo allora di implementare un algoritmo ricorsivo che restituisca n cifre della serie di Fibonacci, ad esempio come:

```
int fibonacci(int n) {
  if(n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

La relazione di ricorrenza di questo algoritmo è:

$$T(0) = T(1) = d, \quad T(n) = b + T(n-1) + T(n-2)$$

da cui chiaramente avremo complessità esponenziale (chiaramente poco). Cerchiamo allora di definire un algoritmo iterativo:

```
int fibonacci(int n) {
  int k; int j = 0; int f = 1;
  for(int i = 1; i <= n; ++i) {
    k = j; j = f; f = k + j;
  }
  return j;
}
```

visto che il nostro numero di passaggi $g(n)$ è n, avremo una complessità totale di $O(n)$. Questo però significa soltanto che la nostra implementazione ricorsiva di partenza era inefficiente. Troviamone un altra:

```
int fibonacci(int n; int a = 0; int b = 1) {
  if(n == 0)  return a;
  return fibonacci(n - 1, b, a + b);
}
```

adesso abbiamo una singola chaiamata ricorsiva su una dimensione di istanza di $(n-1)$, da cui ricaviamo complessità $O(n)$.

2 Mergesort

Il mergesort è un algoritmo di ordinamento basato sull'unione, cioè sulla suddivisione del vettore in sottoinsiemi di dimensione minore, l'ordinamento di quei sottoinsiemi e la loro successiva riunificazione in un unico vettore. In pseudocodice:

```
void mergeSort(sequenza S_1) {
if(S.size() <= 1) return;
else {
  //dividi S in 2 sottosequenze S_1 e S_2 di uguale lunghezza
  sequenza S_2;
  split(S_1, S_2);

  mergeSort(S_1);
  mergeSort(s_2);

  //fondi S_1 e S_2
  merge(S_1, S_2);
  }
}
```

proviamo un'implementazione basata sulle liste:

```
void mergeSort(elem*& s_1) {
  if(s_1 == NULL || s_1->next == NULL) return;
  elem* s_2 = NULL;
  split(s_1, s_2);
  mergeSort(s_1);
```

```
  mergeSort(s_2);
  merge(s_1, s_2);
}
```

sarà necessario, dalla ricorrenza:

$$T(n) = bn + 2T(\frac{n}{2})$$

che una singola chiamata ricorsiva non sia peggiore di $O(n \log n)$. Vediamo le implementazioni della split e della merge:

```
void split(elem*& s_1, elem*& s_2) {
  if(s_1 == NULL || s_1->next = NULL) return;
  elem* p = s_1->next;
  s_1->next = p->next;
  p->next = s_2;
  s_2 = p;
  split(s_1->next, s_2);
}
```

La funzione split() divide la lista in due "sottoliste", estraendo alternativamente un elemento (da mettere in una nuova lista) e lasciandone un altro nella lista di partenza.

```
void merge(elem*$ s_1, elem*& s_2) {
  if(s_2 == NULL) return;
  if(s_1 == NULL) {
    s_1 = s_2;
    return;
  }
  if(s_1->inf <= s_2->inf)
    merge(s_1->next, s_2);
  else {
    merge(s_2->next, s_1);
    s_1 = s_2;
  }
}
```

La funzione merge() invece scorre le due liste contemporaneamente confrontando di volta in volta i due elementi concorrenti in testa ad entrambe e scegliendo il minore.