Appunti Algoritmi e Strutture Dati

Luca Seggiani

6 Marzo 2024

1 Algoritmo Quicksort

Riprendiamo la trattazione dell'algoritmo quicksort (codice negli appunti presi il 5 marzo 2024). Possiamo innanzitutto dire che la complessità della fase non ricorsiva dell'algoritmo (scelta del perno e successiva divisione in parti del vettore) non richede mai più passaggi di quanti sono gli elementi del vettore ed è quindi di O(n). Dovrò però a questo punto considerare le due possibili chiamate ricorsive successive. Posso definire la seguente relazione di ricorrenza:

$$T(1) = a$$

$$T(n) = bn + T(k) + T(n-k)$$

notiamo come la posizione del perno k sia fondamentale all'efficienza dell'algoritmo. Nel caso di un perno estremamente sbilanciato, e.g. k = 1, avremo:

$$T(1) = a$$

$$T(n) = bn + T(n-1)$$

posto invece (come vorremmo) $k = \frac{n}{2}$, avremo:

$$T(1) = a$$

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

che diventerà:

$$T(n) = (n \log n)b + na$$

potremo a questo punto dire che la complessità dell'algoritmo e di $O(n^2)$ nel suo caso peggiore, ma di O(nlogn) nel suo caso medio, cosa che lo rende molto più efficiente di altri algoritmi di ordinamento sui vettori.

2 Torre di Hanoi

La formulazione del problema della torre di Hanoi prevede 3 paletti, con 3 dischetti di diametro decrescente, impilati sul primo paletto. A questo punto si chiede di portare i 3 dischetti, nello stesso ordine, sull'ultimo paletto, con la limitazione fondamentale che non è possibile sposta più di un cerchio alla volta, né di mettere un cerchio più grande su uno più piccolo. Chiamati i 3 paletti A, B e C, definiamo una funzione "trasferisci" che sposta n cerchi dal paletto A al paletto C:

trasferisci una torre di n cerchi da A a C
se n = 1
sposta il cerchio da A a C
altrimenti
trasferisci la torre degli n - 1 cerchi più piccoli da A a B usando C come paletto ausiliario
sposta il cerchio più grande da A a C
trasferisci la torre degli n-1 cerchi più piccoli da B a C usando A come paletto ausiliario

in codice, sfruttando la ricorsione:

```
void hanoi(int n, pal A, pal B, pal C) {
  if(n == 1)
    sposta(A, C);
  else {
    hanoi(n - 1, A, C, B);
    sposta(A, C);
    hanoi(n-1, B, A, C);
  }
}
```

da cui la relazione di ricorrenza:

$$T(1) = a$$

$$T(n) = b + 2T(n-1)$$

che potremo sviluppare in:

$$T(n) = (2^{(n-1)} - 1)b + 2^{(n-1)}a$$

ovvero un T(n) di $O(2^n)$!

3 Altri algoritmi ricorsivi

Ricerca lineare ricorsiva
Poniamo un algoritmo ricorsivo che implementi la ricerca lineare:

```
int RlinearSearch (int A[], int x, int m, int i = 0) {
  if(i == m) return 0;
  if(A[i] == x) return 1;
  return RlinearSearch(A, x, m, i+1);
}
```

dove la chiamata ricorsiva è fatta su una dimensione di istanza di n - 1. La relazione per ricorrenza di questo algoritmo si risolve e decade in una complessità di O(n).

Ricerca binara ricorsiva
Ripetiamo quanto fatto prima sull'algoritmo di ricerca binaria:

```
int binSearch(int A[], int x, int i = 0, int j = m - 1) {
  if(i > j) return 0;
  int k = (i + j) / 2;
  if (x == A[k]) return 1;
  if (x < A[k])
    return binSearch(A, x, i, k - 1);
  else
    return binSearech(A, x, k + 1, j);
}
```

notiamo che le ultime due chiamate ricorsive, simili a quelle del quicksort, sono in questo caso mutualmente esclusive, e la complessità dell'algoritmo decade quindi in $O(\log n)$. Ricordiamo che la ricerca binaria vale solo su vettori ordinati.

Ricerca ricorsiva
Esiste un algoritmo simile alla ricerca binaria ma applicabile a vettori non ordinati:

```
int Search(int A[], int x, int i = 0, int j = n - 1) {
  if(i > j) return 0;
  int k = (i + j) / 2;
  if(x == A[k])
    return 1;
  return Search(A, x, i, k - 1) || Search(A, x, k + 1, j);
}
```

la complessità dell'algoritmo, data la relazione di ricorrenza:

$$T(0) = a$$

$$T(n) = b + 2T(n/2)$$

sarà identica alla sua controparte iterativa, ovvero di O(n).

4 Classificazione di alcuni relazioni di ricorrenza

Diamo adesso una classificazione di alcune delle relazioni di ricorrenza che potremmo incontrare. La maggior parte degli algoritmi che risultano in tali relazioni sono i cosiddetti "divide et impera", ovvero dividi e comanda, cieè algoritmi che dividono l'istanza del problema su cui operano in sottoproblemi più semplici da risolvere separatamente.

```
void divideEtImpera(S) {
  if(|S| <= m)
    //risolvi direttamente il problema
  else
  //dividi in due istanze separate
    divideEtImpera(S_1)
    ...
    divideEtImpera(S_2)
    //combina le due istanze
}
```

classifichiamo quindi alcune delle relazioni di ricorrenza che potremo ottenere da algoritmi simili:

$$T(0) = d$$
$$T(n) = c + T(\frac{n}{2})$$

Con complessità $O(\log n)$,

$$T(0) = d$$
$$T(n) = c + \frac{2T}{n/2}$$

Con complessità $O(n)$,

$$T(0) = d$$
$$T(n) = cn + 2T(\frac{n}{2})$$

Con complessità $O(n \log n)$.
In generale, se abbiamo:

$$T(n) = d, \quad n = 1$$
$$T(n) = c + aT(\frac{n}{b}), \quad n > 1$$

possiamo allora dire che:

$$T(n) \in O(\log n), \quad a = 1$$

$$T(n) \in O(n^{\log_b a})$$

e se abbiamo:

$$T(n) = d, \quad n <= m$$

$$T(n) = hn^k + aT\frac{n}{b}, \quad n > m$$

diremo:

$$T(n) \in O(n^k), \quad a < b^k$$

$$T(n) \in O(n^k \log n), \quad a = b^k$$

$$T(n) \in O(n^{log_b a}), \quad a > b^k$$

Ulteriori generalizzazioni si possono trovare nell'enunciato del Master Theorem, trovato ad esempio nel volume del Cormen.

5 Introduzione agli algoritmi di teoria dei numeri

Prendiamo adesso agloritmi la cui complessità è calcolata sulla base del numero di cifre che compongono il numero stesso. Ad esempio:

- La somma ha complessità lineare
- La moltiplicazione ha complessità quadratica

Moltiplicazione veloce fra interi non negativi
Notiamo la seguente proprietà dei naturali:

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

dove A_s e A_d sono due parti del numero A diviso (rispetto alle cifre) a metà.