Appunti Algoritmi e Strutture Dati

Luca Seggiani

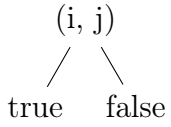
17 Aprile 2024

1 Limiti inferiori

Un problema si dice di ordine $\Omega(f(n))$ se non è possibile trovare un'algoritmo che lo risolva con complessità minore di $f(n)$. Tutti gli algoritmi che lo risolvono saranno quindi $O(n)$. Cercheremo adesso di individuare un limite inferiore per la complessità dell'algoritmo, ovvero un limite massimo di ottimizzazione possibile. Per fare ciò avremo bisogno di un albero decisione.

Alberi di decisione

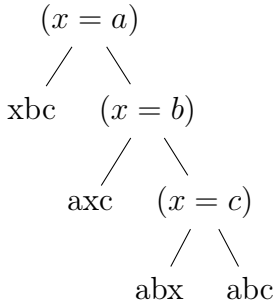
Un'albero di decisione si applica solamente agli algoritmi basati sui confronti, e che hanno complessità proporzionale al numero di confronti effettuatu duranti l'esecuzione dell'algoritmo. L'unità fondamentale di un albero di decisione è un confronto:



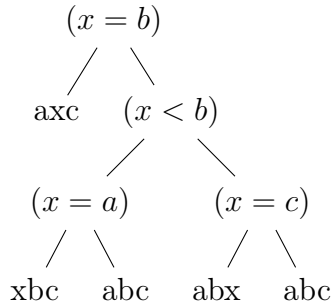
che può produrre una di due possibilità, vero o falso. A questo punto, possiamo dire che un'albero di decisione rappresenta un albero binario che corrisponde all'algoritmo che vogliamo analizzare. Per la precisione:

- Ogni **foglia** corrisponde ad un possibile esito dell'algoritmo;
- Ogni **cammino** lungo l'albero corrisponde ad una possibile esecuzione dell'algoritmo che restituisce un certo risultato, ovvero una certa foglia.

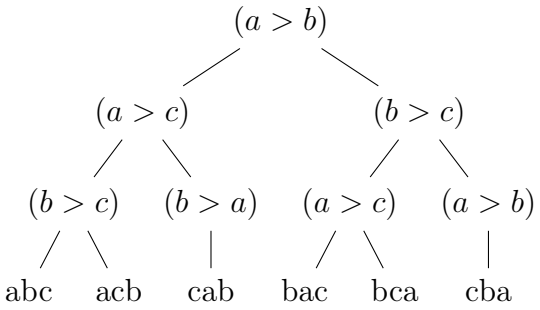
Ad esempio, vediamo l'albero di decisione della ricerca lineare, immaginando di cercare valori su stringhe in forma "abc":



Come vediamo, per dimensioni di stringa uguali a 3 avremo un albero di profondità 3, dove ogni elemento viene scansionato almeno una volta. Vediamo allora la ricerca binaria:



Possiamo infine vedere il selection sort su tre elementi:



Dove notiamo inoltre il confronto $a > b$ potrà essere eseguito due volte lungo un solito cammino. Questo sarà conseguenza naturale dell'esecuzione del selection sort.
Esiste quindi un'equivalenza fra alberi di decisione ed algoritmi. Chiedersi qual'è l'algoritmo più efficiente per risolvere un dato problema significa chiedersi quale sia la lunghezza massima dei percorsi sull'albero binario corrispondente all'algoritmo. Sarà questa lunghezza a minimizzare la complessità dell'algoritmo (fornire un limite inferiore) nel suo caso peggiore. Si possono fare considerazioni simili per quanto riguarda il caso medio: basterà prendere

la lunghezza media dei cammini su uno o più alberi di decisione. In questo caso però andranno fatte anche considerazioni statistiche sulla natura degli input forniti al nostro algoritmo, come va comunque fatto sempre quando si studia la complessità media.

Applicazioni degli alberi di decisione

Un'albero binario di k livelli ha al massimo 2^k foglie nel caso sia bilanciato. Possiamo vedere questa relazione dal senso opposto: un albero binario con s foglie ha almeno $\log_2(s)$ livelli. Gli alberi binari bilanciati minimizzano quindi sia il caso peggiore che quello medio: hanno $\log(s(n))$ livelli.

Vediamo il caso specifico dell'algoritmo di ordinamento: il numero di soluzioni è effettivamente il numero di permutazioni di un insieme di dimensione n, che si calcola come $n!$. Se il numero di soluzioni e $n!$, il cammino medio e massimo saranno $\log(n!) = n \log n$ (dalla formula di Sterling). Per la precisione, avremo che:

$$n! \sim \sqrt{2\pi x} \left(\frac{x}{e}\right)^x \in O(n \log n)$$

Questo ci suggerisce che il mergesort è ottimo, e il quicksort è ottimo nel caso medio. Per quanto riguarda le ricerche, $\log(n)$ è il caso migliore, ma non è sempre raggiungibile: si ha con la ricerca binaria. Esistono poi algoritmi che, per quanto possa sembrare assurdo, adottano alcune soluzioni per avere complessità minore del limite inferiore.

Counting sort

Diciamo di avere una restrizione sui nostri input: abbiamo n elementi, tutti compresi fra 0 e k. Il counting sort sfrutta queste prerogative per ottenere una complessità minore al limite inferiore. Chiaramente, il counting sort conviene solamente quando conosciamo i valori minimo e massimo degli elementi da ordinare. Lo svolgimento è questo:

- Per ogni valore dell'array, si contano i duplicati utilizzando un array ausiliario di dimensione k.
- Successivamente si ordinano i vaori tenendo conto dell'array ausiliario.

In sostanza si tiene conto solamente del numero di occorrenze di ogni valore in k, che vengono poi restituiti in ordine, contati una volta per ogni loro presenza nell'array di partenza. In codice, questo si traduce come:

```
1 void counting_sort(int A[], int k, int n) {
2   int i, j; int C[k + 1];
3   for(i = 0; i <= k; i++) C[i] = 0;
4   for(j = 0; j < n; j++) C[A[j]]++;
5   j = 0;
```

```
  for(i = 0; i <= k; i++) {
    while(C[i] > 0) {
      A[j] = i;
      C[i]--;
      j++;
    }
  }
}
```

Notiamo che il counting sort è estraneo al meccanismo degli alberi di decisione, in quanto non è nemmeno basato sui confronti. Questo gli permette di "eludere" il limite inferiore. Per la precisione, ha complessità $O(n + k)$, e conviene quando k è $O(n)$. Inoltre, il counting sort necessita, a differenza degli algoritmi di ordinamento *in-place*, di memoria ausiliara.

Radix sort

Quando si conosce la lunghezza massima d dei numeri da ordinare, può essere utile implementare il radix sort. Il suo funzionamento è il seguente:

- Si eseguono d passate ripartendo, in base alla d-esima cifra, i numeri in k contenitori, dove k sarà il numero di possibili valori di una cifra.
- Si rilegge il risultato in un determinato ordine.

Il radix sort si basa sulla rappresentazione dell'elemento da ordinare per ottenere un'ordinamento in tempo inferiore a $n \log n$. Si nota che il numero k rappresenta tutti i possibili valori di una cifra, rappresentato il numero in base k. Le passate vengono solitamente svolte da destra verso sinistra, ovvero dalla cifra meno significativa alla più significativa (LSD (*"Least Significant Digit"*) vs MSD (*"Most Significant Digit"*)): questo è utile alla stragrande maggioranza delle applicazioni del radix sort (ordinamento di interi e ordinamento lessicografico).
Vediamo ad esempio come ordinare i numeri 190, 051, 052, 207, 088, 010 con un radix sort (d = 3, k = 10): innanzitutto si inseriscono i numeri nei contenitori in base al valore nell'ultima cifra:

```
010|   |   |   |   |   |   |   |   |
190|051|   |   |054|   |   |207|088|
---|---|---|---|---|---|---|---|---|---
0   |1  |2  |3  |4  |5  |6  |7  |8  |9
```

Si rileggono poi i numeri da sinistra verso destra, e dal basso verso l'alto: 190, 010, 051, 054, 207, 088. Si ripete quindi la stessa operazione sulla penultima cifra:

```
   |   |   |   |   |054|   |   |   |
207|010|   |   |   |051|   |   |088|190
```

```
---|---|---|---|---|---|---|---|---|---
0  |1  |2  |3  |4  |5  |6  |7  |8  |9
```

Si rileggono in modo analogo a prima: 207, 010, 051, 054, 088, 190. Si ripete per un'ultima volta l'operazione sulla prima cifra:

```
088|   |   |   |   |   |   |   |   |
054|   |   |   |   |   |   |   |   |
051|   |   |   |   |   |   |   |   |
010|190|207|   |   |   |   |   |   |
---|---|---|---|---|---|---|---|---|---
0  |1  |2  |3  |4  |5  |6  |7  |8  |9
```

Si leggono a questo punti i valori in modo analogo a prima, ottenendo il vettore ordinato: 010, 051, 054, 088, 190, 207.
Il radix sort, come il counting sort, non è basato sui confronti. La sua complessità è $O(d(n+k))$ dove d è la lunghezza delle sequenze e k è il numero possibile di valori di ogni cifra (quindi nell'esempio precedente $O(3(n+10))$). Necessita, come il counting sort, di memoria ausiliaria. E' particolarmente conveniente quando d è molto minore di n. Si può usare agevolmente per ordinare in ordine alfabetico sequenze di caratteri. Vediamo ad esempio uno spezzone di codice che ordina un vettore STL di stringhe di lunghezza len, con una chiave k che sarà uguale al numero di lettere nell'alfabeto anglosassone (26).

```
void radix(vector<string>& vec, int len) {
  vector<vector<string>> temp;
  temp.reserve(k);
  for(int t = 1; t <= len; t++) {
    for(int i = 0; i < vec.size(); i++) {
      temp[vec[i][len - t]-'@'].push_back(vec[i]);
    }
    int count = 0;
    for(int i = 0; i < k; i++) {
      for(int j = 0; j < temp[i].size(); j++) {
        vec[count] = temp[i][j];
        count++;
      }
      temp[i].clear();
    }
  }
}
```

Bucket sort

Tutti questi algoritmi sono in qualche modo varianti del bucket sort. L'algoritmo bucket sort si limita a scandire la lista, noto la sua variazione di elementi, ed a suddividere i suoi elementi in k "secchielli" in base al loro valore. I secchiellli vengono poi ordinati uno per uno, e infine si ricompone

la lista completa ormai ordinata. Il radix sort è in questo modo una sorta di bucket sort ripetuto, senza alcun passo di ordinamento ma eseguita su diverse cifre degli elementi da ordinare. In pseudocodice, il bucketsort ha il seguente aspetto:

```
//helper
int getMax(int A[], int n) {
  int max = A[0];
  for(int i = 1; i < n; i++) {
    if(A[i] > max) max = A[i];
  }
  return max;
}

//funzione
void bucketSort(int A[], int n, int k) {
  int[][] buckets = new int[][k];
  int max = 1 + getMax(A, n);
  for(int i = 0; i < n; i++) {
    buckets[floor(k * A[i] / max)].push_back(A[i]);
  }
  for(int i = 0; i < k; i++) {
    buckets[i].sort;
  }
  int count = 0;
  for(int i = 0; i < k; i++) {
    for(int j = 0; j < buckets[i].size(); j++) {
      A[count] = buckets[i][j];
      count++;
    }
  }
}
```