Appungi Algoritmi e Strutture Dati

Luca Seggiani

24 Aprile 2024

1 Programmazione dinamica

Finora abbiamo visto algoritmi divide et impera. Vediamo adesso un'altro tipo di approccio: la cosiddetta programmazione dinamica. La programmazione dinamica consiste nel risolvere i sottoproblemi dal basso e conservare i risultati ottenuti per usarli successivamente (strategia bottom-up). Torna molto utile nel caso in cui non si sappia con esattezza in quali sottoproblemi dividere il nostro problema. La programmazione dinamica si può applicare quando il problema presenta:

- Sottostruttura ottima, ovvero una soluzione ottima del problema contiene una soluzione ottima dei sottoproblemi;
- Sottoproblemi comuni, ovvero gli stessi sottoproblemi ricorrono più volte nella soluzione secondo l'approccio ricorsivo.

Più lunga sottosequenze comune

Un'esempio di problema risolvibile da algoritmi di questo tipo è la ricerca della più lunga sottosequenza comune (PLSC). Poniamo di avere 2 stringhe: ci chiediamo di trovare il più grande sottoinsieme di caratteri (non necessariamente contigui, ma strettamente successivi) comune ad entrambe le stringhe. Siano due stringhe:

$$\alpha = \alpha_1, ..., \alpha_i, ..., \alpha_m, \quad \beta = \beta_1, ..., \beta_j, ..., \beta_n$$

E $L(i, j)$ la funzione che associa alle sottostringhe:

$$\alpha = \alpha_1, ..., \alpha_i \quad \beta = \beta_1, ..., \beta_j$$

La loro lunghezza di sottosequenza di lunghezza maggiore. Potremo allora stabilire la seguente relazione di ricorrenza:

- $L(0,0) = L(i,0) = L(0,j) = 0$
- $L(i,j) = L(i-1,j-1) + 1$ se $\alpha_i = \beta_j$
- $L(i,j) = \max(L(i,j-1), L(i-1,j))$ se $\alpha_i \neq \beta_j$

In pseudocodice, possiamo convertire la relazione di ricorrenza in un'algoritmo ricorsivo:

```
int length(char* a, char* b, int i, int j) {
  if(i == 0 || j == 0) return 0;
  if(a[i] == b[j])
    return length(a, b, i - 1, j - 1) + 1;
  else
    return max(length(a, b, i, j - 1),
               length(a, b, i - 1, j));
}
```

E trovare la complessità dalla (altra) relazione di ricorrenza:

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

che corrisponde a $O(2^n)$: l'algoritmo è particolarmente inefficiente. Vediamo se è possibile usare la programmazione dinamica. Abbiamo verificato l'esistenza di sottoproblemi comuni attraverso l'implementazione di un'algoritmo ricorsivo, vediamo allora se la sottostruttura del problema è ottima. Riprendiamo le stringhe:

$$\alpha = \alpha_1, ..., \alpha_i, ..., \alpha_m, \quad \beta = \beta_1, ..., \beta_j, ..., \beta_n$$

Intuitivamente, un certo sottoinsieme χ di α e β conterrà in sé una sottosequenza ottimale (di χ) che sarà a sua volta sottoinsieme della sottosequenza ottimale di α e β. Un'approccio in programmazione dinamica potrà allora essere implementato, e sarà quello di usare una matrice $(m+1) \times (n+1)$ (ci servono una riga e una colonna in più per le sottostringhe date da sottoinsiemi vuoti), dove precalcoleremo tutte le possibili combinazioni di i e j date dalla formula di ricorrenza. La matrice si evolverà quindi in complessità dal punto in alto a sinistra a quello in basso a destra, che rappresenterà il nostro risultato finale.

```
const int m = 7; const int n = 6;
int L [m+1][n + 1];

int quickLength(char* a, char* b) {
  for(int j = 0; j <= n; j++) {
    L[0][j] = 0; //azzero la prima riga
  }
```

```
  for(int i = 0; i <= m; i++) {
    L[i][0] = 0; //azzero la prima colonna
    for(j = 1; j <= n; j++) {
      if(a[i] != b[j])
        L[i][j] = max(L[i][j - 1], L[i - 1][j]);
      else
        L[i][j] = L[i - 1][j - 1] + 1;
    }
  }
  return L[m][n];
}
```

Dalla stessa rapprentazione di prima possiamo poi ottenere anche un esempio della sottosequenza maggiore, partendo dal risultato in fondo a destra e risalendo verso l'alto a sinistra fino a tornare all'origine (quindi trovando man mano gli elementi che hanno formato la sottosequenza maggiore, certo in ordine inverso):

```
void print(int** L, char* a, char* b, int i = m, int j = n) {
  if((i == 0) || (j == 0)) return;
  if(a[i] = b[j]) {
    print(a, b, i - 1, j - 1);
    cout << a[i];
  }
  else if (L[i][j] == L[i - 1][j])
    print(a, b, i - 1, j);
  else print(a, b, i, j - 1);
}
```

Con complessità peggiore O(n + m), nel caso in cui un "salto" sia necessario ad ogni iterazione. Notiamo che in ogni caso la complessità è lineare in quanto in ogni caso la ricorsione è unica per chiamata di funzione (solo un'alternativa viene scelta) e svolta su dimensione d'istanza diminuita di uno su almeno un'asse.

2 Algoritmi greedy

Un'algoritmo greedy (o algoritmo avido, goloso, curioso, ecc...) è un algoritmo che non fa sempre la scelta ottima (ottima globalmente), ma utilizza una certa euristica per fare la scelta migliore (ottima localmente). Ovviamente si richiede di adottare una buona euristica, ovvero una che fa corrispondere il più possibile la scelta ottima localmente a quella ottima globalmente. Sono algoritmi di tipo top-down.

Codici di compressione

Vediamo un esempio: abbiamo che un'alfabeto è un'insieme di caratteri, co-

dificati in codice binario attraverso una certa stringa binaria (che può essere quella ASCII come quella Unicode, ecc...). La codifica di un testo significherà quindi la conversione di ogni carattere del testo nella sua corrispondente codifica in codice binario. La decodifica sarà allora la riconversione delle stringhe in codice binario in caratteri dell'alfabeto di partenza, che possano essere comprensibili ad un utente umano. Ci poniamo il problema di comprimere informazioni di questo tipo per ridurre lo spazio che occupano in memoria: questo può essere particolarmente utile nel caso tali informazioni debbano essere trasferite su qualche canale remoto, su rete, ecc... Questo può essere fatto, ad esempio, utilizzando codici a lunghezza variabile anziché fissa. Potremmo infatti decidere di assegnare codici di lunghezza maggiore a caratteri più rari, e di lunghezza minore a caratteri più frequenti: i caratteri più frequenti (e quindi la maggioranza del testo) richiederanno quindi un minore spazio di archiviazione. A questo punto è utile rappresentare i codici a lunghezza variabile attraverso un'albero binario. Sia 0 codifica di un passo a sinistra, 1 codifica di un passo a destra, la stringa in binario

rappresenterà la sequenza di caratteri: Possiamo dire che l'albero ha tante foglie quanti sono i caratteri dell'alfabeto che consideriamo. La decodifica di una stringa significherà trovare un certo cammino lungo quest'albero. La codifica sarà ottimale quando l'albero sarà completamente binario.

Codici di Huffman

Questo algoritmo è alla base dei codici di Huffman, il cui obiettivo è quello di costruire una codifica ottimale (che minimizza la lunghezza delle codifiche dei testi). L'algoritmo presentato è un'algoritmo greedy e bottom-up rispetto all'albero (cioè che parte dai caratteri (foglie) per arrivare all'albero). L'algoritmo di Huffman gestisce una foresta di alberi: all'inizio del ciclo abbiamo n alberi di un solo nodo con le frequenze dei caratteri. Ad ogni passo, da lì in poi, vengono fusi i due alberi con radice minore introducendo una nuova radice avente come etichetta la somma delle due radici. Vediamo un'implementazione: la struttura dati migliore per poter fare operazioni del genere è un min-heap, ovvero un heap che mantiene il valore minimo alla radice. Il ciclo descritto avrà su questa struttura complessità di $O(\log n)$ su n iterazioni, ovvero $O(n \log n)$. L'approccio greedy funziona perchè ottimo locale e globale coincidono: se sistemiamo prima i nodi con frequnza più bassa, questi apparterranno a livelli più alti dell'albero. Vediamo il codice: