Appunti di Algoritmi e Strutture Dati

4 ottobre 2022

Rosso Carlo

Contents

1 Introduzione

Un *algoritmo* è una procedura computazionale ben definita, che prende dei valori in input e produce dei valori in output in una quantità di tempo finito. Per cui, un algoritmo è una sequenza di step computazionali che trasforma l'input in output.
Un algoritmo serve per risolvere un problema computazionale ben definito.
Per esempio, riordinare un array è un problema computazionale. Un algoritmo si dice corretto se termina in una quantità finita di tempo e restituisce l'output atteso.
Un algoritmo può avere più soluzioni ed ha applicazioni pratiche.

Data Structure Una *data structure* è un modo per memorizzare e organizzare informazioni per facilitarne l'accesso, l'utilizzo e la modifica. Le strutture dati sono una componente importante del design degli algortmi, per cui approfondiamo anche questo argomento.

Tecniche Fondamentalmente impariamo le tecniche per analizzare e sviluppare un algoritmo, capirne e dimostrarne il funzionamento, oltre che l'efficienza.

2 Correttezza di un ciclo

Approfondiamo come dimostrare la correttezza di un ciclo.
A seconda dell'algoritmo non è detto che questo sia l'unico modo per mostrare la correttezza di un algoritmo; la tecnica che utilizziamo sfrutta le invarianti di un ciclo. Le *invarianti di un ciclo* hanno tre proprietà:

- Initialization: una condizione vera prima che cominci il ciclo;
- Maintenance: una condizione che se è vera prima di un'iterazione, è vera anche dopo;
- Termination: il ciclo termina e con l'invariante dimostriamo una proprietà che ci aiuta a dimostrare la correttezza dell'algoritmo.

Le prime due proprietà sono simili all'induzione matematica: la prima proprietà corrisponde al caso base, mentre la seconda corrisponde all'ipotesi induttiva. Quando sono vere le prime due proprietà, l'invariante di ciclo è vera prima di ogni iterazione. Combinando l'invariante di ciclo con l'ultima proprietà, che il ciclo termina, dimostriamo che la proprietà è vera anche alla fine del ciclo.

3 Analisi di un algoritimo

Analizzare un algoritmo vuol dire prevedere le risorse che richiederà per terminare. Le risorse sono la memoria, la larghezza della banda e il consumo di energia. Molto spesso vogliamo misurare il tempo di computazione. Di solito non ci preoccupiamo della precisione dei numeri a virgola mobile, ma in alcune applicazioni si tratta di un aspetto fondamentale.

Dal momento che il tempo di esecuzione di un processo dipende da molte variabili, che non hanno a che fare con l'algoritmo, per determinare la complessità di un algoritmo analizziamo l'algoritmo di per sé.
Il tempo di computazione, quasi sempre dipende dall'input (a noi non interessa studiare il tempo di computazione di algoritmi il cui risultato non dipenda dall'input). Per esempio, il numero di elementi che compone un array incide sul tempo di computazione di un algoritmo che ordina quell'array; oppure il numero di bit che compongono un numero incide sul tempo di computazione di un algoritmo che esegue la moltiplicazione.
Il *running time* di un algoritmo su un imput particolare è il numero di istruzioni e accesso ai data che sono eseguiti per completare l'algoritmo: in questo modo il costo computazionale di un algoritmo è indipendente dalla macchina sulla quale viene implementato. Il tempo di esecuzione è uguale alla somma dei tempi di esecuzione di ciascuno statement (istruzione) che viene eseguito.
In genere si analizza il caso peggiore del running time (se si tratta di ordinare un array mediante l'insertion sort, il caso peggiore è l'array ordinato in modo decrescente), di seguito sono riportati i motivi:

- Il caso peggiore peggiore è il limite superiore della complessità computazionale di un algoritmo, indipendentemente dall'input (per definizione);

- In alcuni algoritmi il caso peggiore accade di frequente, per esempio una ricerca in un database spesso da come esito la mancata presenza dell'oggetto nel database;
- Il "caso medio" di solito indica una complessità computazionale simile al caso peggiore, per esempio il MergeSort;

Order of Growth In informatica, di frequente si utilizzano input molto ampi (altrimenti l'implementazione di un computer non aiuterebbe), per questi input il costo computazione di un algoritmo ha poco a che fare con il costo di ciascuno statement; per questo motivo, quando analizziamo un algoritmo ci concentriamo sull'ordine di crescita dell'algoritmo (rispetto all'input). In particolare ci interessa conoscere l'O-grande ($\Theta(f(n))$, dove n indica le dimensioni dell'input, oppure una caratteristica significativa dell'input. Consideriamo un algoritmo più efficiente di un altro se nel suo caso peggiore ha un minor fattore di crescita ($\Theta(f(n))$).

Ci sono molte tecniche di progettazione di algoritmi; l'insertion sort utilizza la tecnica incrementale: scorre l'array ordinandolo da sinistra verso destra, mettendo in ordine l'elemento i-esimo, la parte dell'array a destra dell'i-esimo valore sarà ordinata, mentre a sinistra ci saranno i valori ancora da ordinare.

4 Divide and Conquer

Molti algoritmi utilizzano una struttura ricorsiva (una funzione che richiama se stessa; la natura della ricorsione è affine con l'iterazione). Un metodo per utilizzare la ricorsione consiste nell'adottare la tecnica *divide and conquer*:

1. Divide: il problema viene diviso in uno o più sottoproblemi che sono analoghi al problem ainiziale, ma di dimensioni ridotte;
2. Conquer: risolve i sottoproblemi, divisi ricorsivamente;
3. Combine: combina le soluzioni dei sottoproblemi per ottenere la soluzione del problema finale.

5 Studio della complessità di un algoritmo

Per studiare la complessità di un algoritmo sono utili alcuni strumenti matematici; in particolare è comodo utilizzare o-piccolo e o-grande di una funzione: perchè studiamo l'efficienza asintotica di un algoritmo (quando un input è infinitamente grande).

O-notation descrive il limite asintotico superiore: una funzione $f(n)$ si dice $f(n) = O(g(n))$ se $\exists c > 0 : f(n) \leq cg(n), \forall n > n_0$.

Ω-notation descrive il limite asintotico inferiore: una funzione $f(n)$ si dice $f(n) = \Omega(g(n))$ se $\exists c > 0 : f(n) \geq cg(n), \forall n > n_0$.

Θ-notation descrive il limite asintotico: una funzione $f(n)$ si dice $f(n) = \Theta(g(n))$ se $\exists c_1, c_2 > 0 : c_2 g(n) \geq f(n) \geq c_1 g(n), \forall n > n_0$.

Per cui $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$ implica $f(n) = \Theta(g(n))$.
o-notation: una funzione $f(n)$ si dice $f(n) = o(g(n))$ se $\exists c > 0 : f(n) < cg(n), \forall n > n_0$. Viceversa, w-notation: una funzione $f(n)$ si dice $f(n) = \omega(g(n))$ se $\exists c > 0 : f(n) > cg(n), \forall n > n_0$.

6 Complessità di un algoritmo Divide-and-Conquer

Il metodo divide-and-conquer è uno strumento utile per progettare algoritmi asintoticamente efficienti. Una ricorrenza T(n) si dice algoritmica se, per ogni $n_0 > 0$ sufficientemente grande, valgono le seguenti proprietà:

1. per ogni $n < n_0$, $T(n) = \Theta(1)$;
2. per ogni $n \geq n_0$, ogni passo ricorsivo arriva al caso base in un numero finito di chiamate ricorsive.

6.1 Tecniche

Ci sono più tecniche per calcolare la complessità computazionale del caso peggiore di un algoritmo:

- substitution method: si prende una funzione per ipotesi e si dimostra per induzione che la funzione è il limite asintotico ($\Theta(g(n))$);
- recursion-tree method: si disegna un albero per rappresentare la ricorsione, vicino ad ogni nodo si scrive il costo; permette di intuire la funzione da usare per ipotesi per il substitution method;
- master method: quando applicabile, è il metodo più semplice. Fornisce il limite asintotico per funzioni del tipo $T(n) = aT(n/b) + f(n)$, dove $a > 0, b > 1$ sono costanti. Calcola il limite asintotico di un algoritmo divide-and-conquer che crea a sottoproblemi, i quali sono $1/b$ volte le dimensioni del problema precedente;
- Akra-Bazzi method: è un modello di calcolo per risolvere ricorrenze del tipo divide-and-conquer. Quando i metodi precedenti non sono applicabili si utilizza questo.

Substitution Method

Il metodo della sotituzione è il metodo più generale; è diviso in due passaggi:

1. formulare un'ipotesi dell'asintoto;
2. dimostrare l'ipotesi mediante l'induzione matematica.

Si utilizza questo metodo per dimostrare gli asintoti superiori ed inferiori (è poco pratico dimostrare l'asintoto più stretto, $\Theta(g(n))$, in un colpo solo).
Questa tecnica funziona in questo modo:

1. si formula un'ipotesi dell'asintoto, $T(n) \leq g(n)$;
2. si sostituisce la funzione asintotica con la chiamata ricorsiva;
3. attraverso semplificazioni matematiche si mostra che l'*ipotesi induttiva* regge;
4. si mostra che esiste un n_0, tale che $T(n_0) \leq g(n_0)$.

Si tratta a tutti gli effetti dell'induzione matematica.
NB piuttosto che utilizzare la notazione degli O-grande oppure Θ-grande, conviene utilizzare una funzione con coefficienti generici, diminuisce gli errori nel calcolo.

Recursion-tree mathod

Nell'albero ricorsivo, ogni nodo rappresenta il costo di un singolo sottoproblema. Sommando il costo di ciascun nodo sullo stesso livello si ottiene il costo dell'albero per livelli. A questo punto in genere si ottiene una sommatoria.
Ci sono due modi per continuare, si risolve la sommatoria in modo esatto, oppure si fanno delle semplificazoni, per ottenere una funzione asintotica. Nel primo caso, dimostriamo di aver trovato l'asintoto. Nel secondo, otteniamo l'ipotesi di un asintoto, che utilizziamo nel metodo precedente.

Master method

Il *master method* permette di risolvere la maggior parte delle funzioni del tipo:

$$T(n) = \alpha T(\frac{n}{\beta}) + f(n) \tag{1}$$

dove $\alpha > 0$ e $\beta > 1$.

Master theorem Siano $\alpha > 0$ e $\beta > 1$ e $f(n) : \mathbb{R}_+ \to \mathbb{R}_+$. Definendo la ricorrenza $T(n), n \in \mathbb{N}$:

$$T(n) = \alpha T(\frac{n}{\beta}) + f(n)$$

allora il comportamento asintotico di $T(n)$ può essere caratterizzato come segue:

1. Se $\exists \epsilon$, costante, tale che $f(n) = O(n^{\log_b(\alpha-\epsilon)})$, allora $T(n) = \Theta(n^{\log_b \alpha})$;
2. Se $\exists k \leq 0$, costante, tale che $f(n) = \Theta(n^{\log_b \alpha} \log^k n)$, allora $T(n) = \Theta(n^{\log_b \alpha} \log^k n)$;
3. Se $\exists \epsilon$, costante, tale che $f(n) = \Omega(n^{\log_b(\alpha+\epsilon)})$, e $\alpha f(n/\beta) \leq cf(n)$, per qualche costante $c < 1$, per ogni $n > n_0$, allora $T(n) = \Theta(f(n))$;

7 Dynamic Programming

Proprio come il metodo divide and conquer (4), la programmazione dinamica sfrutta l'induzione, ovverosia si applica quando un problema per essere risolto ha bisogno di risolvere dei sotto-problemi, fino ad arrivare ad una serie di casi base.
La programmazione dinamica memorizza i risultati dei sotto-problemi, in modo da non doverli ricalcolare ogni volta.
Può essere implementata in due modi:

1. **Bottom-up**: si risolvono i sotto-problemi, partendo dai casi base, fino ad arrivare al problema iniziale;
2. **Top-down**: si risolvono i sotto-problemi, partendo dal problema iniziale, fino ad arrivare ai casi base.

Per sviluppare un algoritmo che sfrutti la programmazione dinamica, risulta comodo seguire i seguenti step:

1. caratterizzare la sottostruttura ottima, ovvero la struttura che risolve il problema in modo ottimale;
2. formulare un'ipotesi ricorsiva per calcolare la soluzione ottima;
3. dimostrare che l'ipotesi ricorsiva è corretta;
4. calcolare la soluzione ottima in modo efficiente.

Fondamentalmente per risolvere un problema con la programmazione dinamica, si deve trovare una relazione ricorsiva che permetta di calcolare la soluzione ottima, dipendente da soluzioni ottimali di sotto-problemi, proprio come il meccanismo del divide and conquer, in più salva ciascun risultato in una tabella, in modo tale da non doverlo ricalcolare.
Bisogna poi dimostrare che la relazione ricorsiva è corretta, e che i sotto- problemi siano indipendenti e che i sotto-problemi siano di dimensione decrescente.
La proprietà di sottostruttura ottima è proprio la relazione ricorsiva.
La programmazione dinamica offre un trade-off tra spazio e tempo, in quanto richiede più spazio per memorizzare i risultati, ma risolve i problemi in un tempo più breve, perché non ricalcola i risultati di sotto-problemi già risolti.

8 Greedy Algorithms

Un algoritmo **greedy** è un algoritmo che, per risolvere un problema, si basano su una scelta locale ottima, senza considerare il contesto globale.
Un algoritmo **greedy** è corretto se:

1. la soluzione ottima è costituita da una sequenza di scelte locali ottimali;
2. le scelte locali ottimali sono indipendenti dai risultati delle scelte future.

Per sviluppare un algoritmo **greedy** si deve seguire il seguente schema:

1. definire la sottostruttura ottima del problema;

2. sviluppare una soluzione ricorsiva;
3. dimostrare che utilizzando la scelta greedy rimane solo un sotto-problema da risolvere;
4. dimostrare che la scelta greedy è corretta;
5. sviluppare un algoritmo che sfrutta la scelta greedy;
6. convertire l'algoritmo da ricorsivo a iterativo.