

Analisi di algoritmi

Metodo Scientifico

Un simile approccio che gli scienziati utilizzano per comprendere il mondo naturale è effettivo per studiare il Tempo di esecuzione dei programmi:

- Osservare alcuni particolari del mondo naturale.
- Ipotizzare un modello che sia consistente con le osservazioni.
- Prevedere eventi usando le ipotesi.
- Verificare le previsioni facendo osservazioni approfondite.
- Validare rilevando l'esperimento finché ipotesi e osservazioni coincidono.

Osservazioni

La nostra prima sfida è di determinare come fare misure quantitative del tempo di esecuzione del nostro programma.

Ogni volta che eseguiamo un programma stiamo eseguendo un esperimento scientifico che collega il programma al mondo reale, e che risponde ad una delle domande core: Quanto tempo necessita il mio programma?

Esempio: Pg 173, ThreeSum, avendo un programma che, tra una lista di numeri interi, trova tutte le Triplette che abbiano come somma 0.

StopWatch: abbiamo bisogno di misurazioni eadeguate per generare dati accurati che poniamo avere usati per formulare ipotesi.

Per questo motivo, usiamo la classe Stopwatch(). Il metodo elapsedTime() ritorna il tempo passato da quando è stato creato. La classe usa il metodo di sistema currentMillis(), che ritorna i millisecondi correnti.

Analisi di dati sperimentali

Il programma DoublingTest è uno stopwatch più sofisticato che produce data sperimentale per ThreeSum. Genera una sequenza di input random, raddoppiando la grandezza dell'array ad ogni passo, e stampa i Tempi di esecuzione di ThreeSum.count() per ogni grandezza di input.

Di conseguenza ci poniamo le domande:

Quanto tempo impiegherà il programma in funzione delle grandezze di input?

Possiamo stimare il running time con un grafico log-log

$$\lg(T(N)) = 3 \lg N + \lg a \quad (\text{a costante})$$
$$\Leftrightarrow T(N) = a N^3$$

Tilde approximation

In analisi di questo tipo sono generate espressioni matematiche complicate e lunghe. Basta considerare l'espressione di Threesum:

$$N(N-1)(N-2)/6 = N^3/6 - N^2/2 + N/3$$

Come è tipico in espressioni di questo tipo, i termini dopo il termine principale, sono relativamente insignificanti.

Ad esempio se $N=1000$ il valore di $-N^2/2 + N/3 \approx 499$ è insignificante rispetto a $N^3/6 \approx 166.666$.

Per permetterci di ignorare termini e semplificare usiamo un device conosciuto come Tilde Notation ($\tilde{}$). Questa notazione ci permette di eliminare i termini low-order:

Definizione: scriviamo $\tilde{f}(N)$ per rappresentare una qualsiasi funzione, che quando divisa per $f(N)$ si avvicina ad 1 per $N \rightarrow \infty$.

Scriviamo $g(N) \sim f(N)$ per indicare che $g(N)/f(N)$ si avvicina ad 1 per $N \rightarrow \infty$

ORDINI DI CRESCITA

costante $\longrightarrow 1$

quadratico $\longrightarrow N^2$

logaritmico $\longrightarrow \log N$

cubico $\longrightarrow N^3$

lineare $\longrightarrow N$

esponenziale $\longrightarrow 2^N$

linearitrico $\longrightarrow N \log N$

Analisi degli algoritmi

Lavorare con gli ordini di crescita ci permette di separare un programma dall'algoritmo che implementa.

Dire che l'ordine di crescita di ThreeSum è N^3 non dice solo del fatto che sia implementato in Java o eseguito sul mio computer o un supercomputer; ma di dire solo del fatto che esamina ogni singola Tripletta.

Cost model

Concentriamo la nostra attenzione su proprietà degli algoritmi articolando un cost model che definisce le operazioni base usate dagli algoritmi.

Ad esempio, nell'algoritmo ThreeSum utilizziamo il numero di volte che l'algoritmo accede ad un array. Con questo modello siamo in grado di fare delle dichiarazioni matematiche sulle proprietà dell'algoritmo.

Proposizione B. L'algoritmo brute-force 3-Sum usa $\sim N^3/2$ accesi alla memoria per elaborare il numero di Tripletti la cui somma sia 0 tra N numeri.

Dimostrazione: L'algoritmo accede ad ognuno dei 3 numeri per ogni $\sim N^3/6$ Tripletti.

classificazione ordini di crescita

- Costante: Un programma che ha come ordine di crescita un numero costante esegue un numero fisso di operazioni, e il suo running time non dipende da N .
- Logaritmico: Un algoritmo poco più lento di uno costante; un esempio è la ricerca binaria (pg 47).
- Lineare: Programmi che spendono una quantità di tempo costante per processare ogni pezzo di input, o che sono basati su un singolo for loop. Il tempo di esecuzione viene incrementato linearmente rispetto al numero di elementi di input.
- Linearitico: Sono così descritti i programmi che per un input di grandezza N ha un ordine di crescita $N \log N$.
Alcuni programmi di questo tipo sono il MergeSort() e quicksort().
- Quadratico: Un programma che ha come ordine di crescita di N^2 , o che sfrutta due cicli for innestati. Esempi di questo tipo sono il SelectionSort() e InsertionSort()
- Cubico: N^3 o 3 cicli for innestati. Un esempio può essere Threesum().

Ricerca Binaria

L'algoritmo è implementato nel metodo statico rank(), che riceve un intero int input ed un array ORDINATO. Ri torna l'indice dell'array se Trova l'intero, e -1 se non è presente nell'array.

Inizialmente si punta all'elemento centrale $lo + (hi - lo)/2$ e lo si confronta con l'elemento da cercare. Se $mid = key$ abbiamo Trovato l'intero e usciamo. Se $mid > key$ ci spostiamo a SINISTRA, se $mid < key$ ci spostiamo a DESTRA. Iteriamo. Se all'ultima iterazione, ovvero ci troviamo con un solo numero, non abbiamo Trovato la key, ritorniamo -1.

```
public static int rank(int key, int[] a) {  
    int lo = 0;  
    int hi = a.length - 1;  
    while (lo <= hi) {  
        int mid = lo + (hi - lo) / 2;  
        if (key < a[mid]) hi = mid - 1;  
        else if (key > a[mid]) lo = mid + 1;  
        else return mid;  
    }  
    return -1;  
}
```

Sum-2

Consideriamo il caso più semplice del problema 3-Sum, dove dobbiamo trovare tutte le coppie di numeri, la quale somma dia 0.

La soluzione è semplice, basta eliminare un loop da 3-Sum ed avremo che il problema sarà risolto in un tempo quadratico: N^2 .

Poniamo avere una versione migliorata dell'algoritmo in modo da ottenere un tempo di esecuzione linearitico, sfruttando il merge sort e la ricerca binaria.

L'algoritmo sfrutta l'idea che, per ottenere una somma pari a 0, dobbiamo sommare a m, -m, o escluso.

L'algoritmo: come primo passo ordiniamo in ordine crescente la collezione di interi. Dopo aver fatto ciò utilizziamo la ricerca binaria per trovare l'opposto per ogni numero.

Se la ricerca ritorna un indice j , con $j > i$ (Ricercando $-a[i]$), allora incrementiamo il contatore.

Se la ricerca ritorna j , ma con $0 < j < i$ non incrementiamo il contatore, perché avremmo un conteggio doppio.

Averemo una soluzione linearitica

lower bounds

Abbiamo parlato degli algoritmi 2-Sum, 3-Sum e 2-Sum_{FAST}, 3-Sum_{FAST}. Ma la domanda è:
Esiste un algoritmo lineare per 2-Sum e 3-Sum?
La risposta è NO per il 2-Sum, e per quanto riguarda il 3-Sum, non si sa.

d'idea di un lower bound sull'ordine di crescita sul tempo di esecuzione peggiore è un'idea molto potente.