Appunti Algoritmi e Strutture Dati

Luca Seggiani

27-28 Febbraio 2024

1 Introduzione agli algoritmi

Un'algoritmo non è altro che un procedimento che descrive una sequenza di passi ben definiti atti a risolvere un problema, o più schematicamente:

- Insieme finito di istruzioni
- Accetta un input e restituisce un output
- Ogni istruzione deve essere:
  - ben definita
  - eseguibile in tempo finito da un'agente di calcolo
- Può eventualmente usufruire di una memoria per stadi intermedi.

noi tratteremo di algoritmi aritmetici. Alcuni esempi di tali algoritmi possono essere;

- L'algoritmo di Euclide per il calcolo del MCD di due numeri
- Il setaccio di Eratostene per il calcolo dei primi numeri primi

Si può dire che l'algoritmo è l'essenza computazionale di un dato programma, notando però che:

$$\text{Algoritmo} \neq \text{Programma}$$

in quanto l'algoritmo è slegato dal linguaggio di implementazione, la macchina e l'ambiente su cui gira, mentre il programma equivale al suo codice ormai completamente definito.

2 Esempi basilari di algoritmi

Si riportano due semplici algoritmi ed alcune loro caratteristiche e possibili ottimizzazioni.

Algoritmo di Euclide

Nella sua definizione più semplice, l'algoritmo di Euclide si basa sul teorema: *"Il MCD fra 2 numeri è il MCD del più piccolo fra i due e la loro differenza"* e si implementa, in pseudocodice C++, nella forma:

```
int gcd(int a, int b) {
  while(a != b) {
    if(a > b) a -= b;
    else b -= a;
  }
  return a;
}
```

una chiara ottimizzazione possibile è il passaggio dall'operazione di sottrazione a quella di sottrazione ripetuta (divisione), nel seguente modo:

```
int betterGcd(int a, int b) {
  while(b > 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

si nota che la complessità dell'algoritmo è di $O(n)$.

Setaccio di Eratostene

Il setaccio di Eratostene è un algoritmo che ci permette di trovare tutti i numeri primi fra un insieme dei primi n naturali. Un approccio naive potrebbe essere quello di dividere ogni numero per ogni suo predecessore, ottenendo una complessità nel caso peggiore di $O(n^2)$. Un'alternativa più efficiente di questo algoritmo potrebbe considerare una tabella con i primi n naturali, e successivamente cancellare ogni multiplo di ogni numero primo trovato finora. raggiunto un numero il cui quadrato è maggiore di n, l'algoritmo si arresta e i numeri rimasti sono i primi cercati. In pseudocodice:

```
void sieve(int n) {
  bool num[n];

  //preparazione array
  num[0] = false; num[1] = true;
  for(int i = 2; i < n; i++) {
    num[i] = true;
```

```
  }

  int i = 2;
  while(i*i < n) {
    while(num[i] == false) i++;

    int m = 2;
    while(i * m < n) {
      num[i * m] = false;
      m++;
    }
    i++;
  }

  for(int i = 0; i < n; i++) {
    if(num[i] == true) {
      cout << i << " ";
    }
  }
}
```

che ottiene una complessità temporale di $O(n \log \log n)$.

3 Nozioni sugli algoritmi

Definito un determinato problema, si può utilizzare un'algoritmo su una sua certa istanza, analizzando quindi le seguenti caratteristiche:

- Problema: il problema da risolvere (e.g. trovare i primi n primi)
- Dimensione dell'istanza: la dimensione della mole di dati su cui dovrò lavorare nell'istanza attuale (e.g. il valore di n)
- Modello di calcolo: un modello che associa ad ogni operazione effettuata dall'algoritmo un certo costo per il mio ambiente
- Correttezza: se il mio algoritmo ottiene effettivamente la risposta giusta!

oltre alle caratteristiche riportate, è poi fondamentale la valutazione della

4 Complessità

La complessità di un'algoritmo è una funzione che associa alla dimensione del problema il costo della sua risoluzione, sia esso in termini di memoria

o tempo (in questo caso, *complessità temporale*). La dimensione dell'istanza dipende dai suoi stessi dati, ed è in generale opportuno determinare, su una istanza generica:

- Caso peggiore (best-case): scenario dove l'algoritmo ottiene il costo di esecuzione maggiore
- Caso migliore (worst-case): scenario dove l'algoritmo ottiene il costo di esecuzione minore
- Caso medio (average-case): scenario medio di esecuzione dell'algoritmo, necessita di un'analisi statistica dei dati delle istanze in entrata.

Si ricorda che l'efficienza di un'algoritmo dipende solo dal modello di calcolo adottato, ed è slegata dall'efficienza di un qualsiasi ambiente di esecuzione.

Posso ad esempio definire, per un determinato algoritmo P, la funzione complessità:

$$T_P(n)$$

che determina la complessità temporale di P al variare della dimensione d'istanza n. Prendiamo in esempio un semplice algoritmo che cerca il massimo di un vettore:

```
int max(int vett[], int n) {
  int m = a[0];
  for(int i = 0; i < n; i++) {
    if(m > a[i]) m = a[i];
  }
  return m;
}
```

possiamo adesso definire un modello di calcolo molto semplice, che associa il costo di 1 ad ogni operazione di assegnamento o confronto. Tenendo conto dell'assegnamento iniziale, quello dell'istruzione di ritorno, e gli assegnamenti e confronti eseguiti per l'inizializzazione e l'esecuzione del ciclo for, otteniamo:

$$1 + 1 + 4(n - 1) + 1 + 1 = 4n$$

ovvero la nostra funzione complessità dell'algoritmo. Per poter trattare di complessità senza però perdersi nelle specifiche, visto che l'unico fattore di interesse è la scalabilità dell'algoritmo, occorre definire la

5 Notazione O grande

Definiamo la notazione $O(f(n))$ nel seguente modo:

$$f(n) \in O(g(n)) \Leftrightarrow \exists n_0, \quad c > 0 \quad \text{t.c.} \quad \forall n > n_0 : f(n) \leq cg(n)$$

si definisce poi una serie di regole per l'utilizzo di tale notazione:

- fattori costanti: ammesso una certa costante k, si ha:

$$O(f(n)) = O(kf(n))$$

- somma:

$$f(n) \in O(g(n)) \Rightarrow f(n) + g(n) \in O(g(n))$$

- prodotto:

$$f(n) \in O(f_1(n)), \quad g(n) \in O(g_1(n)) \Rightarrow f(n)g(n) \in O(f_1(n)g_1(n))$$

- transitività:

$$f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$$

- costanti:

$$\forall k, \quad k \in O(1)$$

- potenze:

$$m \leq p, \quad n^m \in O(n^p)$$

- polinomi:

$$p(x) \in R[x], \quad deg(p) = m, \quad p(x) \in O(n^m)$$

si nota che esistono funzioni tra di loro incommensurabili. Inoltre, per qualsiasi k:

$$\forall k, \quad n^k \in O(a^n), \quad \forall n > 1$$

ovvero qualsiasi polinomiale è sempre migliore di ogni esponenziale.

Per continuare la discussione della complessità degli algoritmi, occorre poi definire anche la:

Notazione Ω grande
La notazione Ω grande definisce effettivamente l'opposto della O grande:

$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists n_0, \quad c > 0 \quad \text{t.c.} \forall n > n_0 : f(n) \geq cg(n)$$

Notazione Θ grande
La notazione Θ grande correla due funzioni che condividono lo stesso ordine di complessità:

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$$

oppure ancora:

$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists n_0 \quad \exists c_1, c_2 \quad \text{t.c.} \quad \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

per la notazione Ω e Θ grande esistono inoltre le regole:

- antisimmetria O grande / Ω grande:

$$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

- simmetria Θ grande:

$$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$$

- per Θ grande e Ω grande valgono le regole definite per O grande, ovvero dei fattori costanti, somma e prodotto, transtività, potenze e polinomi. Si nota soratutto che per ogni polinomio:

$$p(x) \in R[x], \quad deg(p) = m, \quad p(x) \in \Theta(n)$$

in una nota un'attimo più formale, si può affermare che le notazioni O grande, Ω grande e Θ grande stabiliscono sull'insieme delle funzioni naturali $f(n)$ delle classi di equivalenza, rispettando effettivamente le tre proprietà di riflessività, simmetria e transitività. Inoltre, si può dire che l'appartenenza ad una classe di complessità stabilisce fra funzioni una relazione d'ordinamento, nell'ordine:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^m) < O(m^n) < O(n^n)$$

Si ricorda inoltre che queste notazioni considerano solamente l'andamento *asintotico* di una funzione, in quanto tutto ciò che accade prima di n_0 è effettivamente irrilevante all'analisi che vogliamo fare (almeno dal punto di vista della scalabilità).

6 Complessità degli algoritmi iterativi

Si presenta ora un semplice modello di calcolo per programmi scritti in pseudolinguaggio C++, compatibile con i tre paradigmi della programmazione strutturata (esecuzione sequenziale, condizionale ed iterativa).

Quello che vogliamo definire è una funzione del tipo:

$$\text{C[costrutti del linguaggio]} \rightarrow \text{classi di complessità}$$

useremo inoltre le abbreviazioni:

$$\text{V: costanti, I: variabili, E: espressioni, C: comandi}$$

iniziamo con semplici operazioni di assegnamento e lettura variabili, valutazioni di espressioni, selezione da vettori nonché l'istruzione di salto *return*:

$$C[V] = C[I] = O(1)$$

$$C[E_1 \circ E_2] = C[E_1] + C[E_2]$$

$$C[I[\,]] = C[E]$$

$$C[I = E] = C[E]$$

$$C[I[E_1] = E_2] = C[E_1] + C[E_2]$$

$$C[\text{return } E] = C[E]$$

si definisce poi per l'esecuzione sequenziale:

$$C[C_1, ..., C_n] = C[C_1] + ... + C[C_n]$$

per l'esecuzione condizionale:

$$C[\text{if}(E)C] = C[E] + C[C]$$

$$C[\text{if}(E)\, C_1 \text{ else } C_2] = C[E] + \max(C[C_1], C[C_2])$$

e per l'esecuzione condizionale ($O(g(n))$ è il numero di iterazioni in funzione della dimensione d'istanza):

$$C[\text{for}(E_1, E_2, E_3)C] = C[E_1] + C[E_2] + (C[C] + C[E_2] + C[E_3]) \cdot O(g(n))$$

$$C[\text{while}(E)C] = C[E] + (C[C] + C[E]) \cdot O(g(n))$$

si definisce inoltre per la chiamata di funzione:

$$C[F(E_1, ..., E_n)] = C[E_1] + C[E_n] + C[C...C]$$

7 Caratterizzazione di caso peggiore, migliore e medio

Definiamo matematicamente le nozioni di caso peggiore, migliore e medio sull'insieme di istanze In:

- Caso peggiore:
$$T_{worst} = max_{In}(\text{tempo}(I))$$
- Caso migliore:
$$T_{best} = min_{In}(\text{tempo}(I))$$
- Caso medio (la funzione $P(I)$ restituisce la probabilita di verificarsi di una certa istanza):
$$T_{average} = \sum_{In}(\text{tempo}(I) \cdot P(I)), \quad P(I) \in [0, 1]$$
da cui possiamo ovviamente dire:
$$T_{worst} \leq T_{average} \leq T_{best}$$

8 Complessità di algoritmi di uso comune

Iniziamo la discussione di alcuni algoritmi di ordinamento di vettori in loco (in-place), basati sul confronto, ammettendo l'accesso diretto in memoria in tempo $O(1)$. Definiamo inanzitutto una funzione di scambio con ausilio di variabile temporanea:

```
void swap(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}
```

Selection sort

Il selection sort lavora in maniera incrementale su n elementi, selezionando iterativamente l'i-esimo elemento e confrontandolo con glin-i elementi successivi in modo da scambiarlo con il minimo trovato. Dopo n-1 iterazioni il vettore risultaordinato. In pseudocodice:

```
void selectionSort(int vett[], int n) {
  for(int i = 0; i < n - 1; i++) {
    int m = i;
    for(int j = i + 1; j < n; j++) {
```

```
      if(vett[j] < vett[m]) m = j;
    }
    scambia[vett[m], vett[i]];
  }
}
```

il selection sort presenta caratteristiche identiche nel suo caso migliore e peggiore, effettuando sempre le stesse operazioni su qualsiasi tipo di istanza. In particolare, la complessità dipende dal numero di confronti:

$$(n-1)+(n-2)+(n-3)+...+2+1=\sum_{i=1}^{n} i=\frac{n(n-1)}{2}\in O(n^2)$$

e il numero di scambi, che è $O(n)$. Alternativa al selection sort può essere il

Bubble sort

Il bubble sort (algoritmo di ordinamento a bolle) non adotta un'approccio incrementale, ma si limita a scambiare da destra verso sinistra le coppie adiacenti non ordinate n-1 volte, finchè il vettore non risulta ordinato. In pseudocodice:

```
void bubbleSort(int vett[], int n) {
  for(int i = 0; i < n - 1; i++) {
    for(int j = n-1; j > i; j--) {
      if(vett[j] < vett[j - 1]) scambia(vett[j], vett[j - 1])
   ;
    }
  }
}
```

a differenza del selection sort, il bubble sort un numero sia di confronti che di scambi pari a $O(n^2)$. Questo apparente difetto è però compensato da una possibile ottimizzazione disponibile in fase di scorrimento del vettore: basterà infatti controllare eventuali scorrimenti senza scambi per poter ottenere una complessità temporale in caso migliore di $O(n)$ (caso di scorrimento singolo al primo ciclo). In pseudocodice:

```
void betterBubbleSort(int vett[], int n) {
  for(int i = 0; i < n - 1; i++) {
    bool q = false;
    for(int j = n-1; j > i; j--) {
      if(vett[j] < vett[j - 1]) {
        scambia(vett[j], vett[j - 1]);
        q = true;
      }
    }
    if(q == false) {
      break;
```

```
    }
  }
}
```

si può in generale applicare i limiti asintotici sia al costo di un singolo algoritmo che alla complessità di un intero programma. Possiamo ad esempio definire, su un qualsiasi problema:

- Limite asintotico superiore: il minimo ordine di complessità superiore a tutti gli ordini di complessità degli algoritmi atti a risovlere il problema (in sostanza caso peggiore)
- Limite asintotico inferiore il massimo ordine di complessita inferiore a tutti gli ordini di complessità degli algoritmi atti a risolvere il problema, quindi il miglior caso possibile per un qualsiasi algoritmo.

  Ad esempio, per quanto riguarda l'ordinamento di vettori monodimensionali, il limite asintotico inferiore è pari a $O(n \log n)$.

Ricerca lineare L'algoritmo di ricerca lineare di una certa chiave x su un certo vettore scansiona ogni singolo elemento confrontandolo alla suddetta, ottenendo complessità (appunto) lineare $O(n)$. In pseudocodice:

```
int linearSearch(int vett[], int n, int x) {
  int b = 0;
  for(int i = 1; i < n; i++) {
    if(vett[i] == x) b = 1;
    return b;
  }
}
```

si nota che il caso migliore (quello in cui il primo elemento è quello cercato) ha complessità $O(1)$.

Divisioni ripetute

Si riporta un algoritmo che divide un certo numero per 2 ciclicamente fino ad azzerarlo (ovviamente sul gruppo degli interi), magari per volerlo convertire in base 2:

```
int div_2(int n) {
   int i = 0;
   while(n > i) {
   n = \frac{n}{2}; i++;
   }
   return i;
}
```

possiamo quindi estrapolare una misura della complessità sul caso peggiore (quello in cui n è potenza di 2):

$$n = 2^m \Leftrightarrow m = \log_2 n \rightarrow O(log(n))$$

Ricera binaria

L'algoritmo di ricerca binaria (o ricerca dicotomica) effettua la stessa operazione della ricerca lineare, ma parte dall'assunto che il vettore sia già stato ordinato per approfittare di una notevole ottimizzazione, che porta la complessità temporale a $O(log(n))$. L'ottimizzazione usata si basa sul confronto della chiave con l'elemento centrale del vettore (e non di ogni suo elemento), che permette di dimezzare ciclicamente il vettore ottenendo quindi una riduzione a velocità logaritmica della dimensione d'istanza. In pseudocodice l'implementazione ricorsiva:

```
int binarySearch(int vett, int min, int max, int x) {
  if(max < min) return -1;
  int med = (min + max) / 2;
  if(vett[med] == key) {
    return med;
  }
  if(vett[med] > key) {
    return binarySearch(0, med - 1, key, dest);
  } else {
    return binarySearch(med + 1, max, key, dest);
  }
}
```