Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2019/2020

2020

Indice

11. Costo dei programmi, algoritmi di ricerca e di ordinamento

11.1. Costo dei programmi

Il costo di un programma dipende dalle risorse utilizzate: ci concentriamo su spazio di memoria e tempo di elaborazione. Altre risorse di interesse sono: la quantità di traffico generata su una rete di calcolatori; la quantità di dati che devono essere trasferiti da e su disco, ecc.

Stabilire il costo computazionale di una funzione/programma/algoritmo serve a confrontare diversi funzioni/programmi/algoritmi che risolvono lo stesso problema, in modo da scegliere il più efficiente.

11.1.1. Misura del tempo

La misura del tempo può essere effettuata valutando il tempo della CPU necessario all'esecuzione del programma su un insieme di dati di ingresso particolari che fungono da banco di prova (benchmark), oppure tramite l'analisi di costo del programma.

Difficoltà della misura:

- Uno stesso programma potrebbe comportarsi diversamente se eseguito su due diversi computer.
- Se abbiamo due diversi programmi che ordinano una collezione di dati, non è detto che si comportino allo stesso modo sugli stessi dati di ingresso.
- Uno stesso algoritmo scritto in due diversi linguaggi di programmazione, potrebbe esibire comportamenti diversi a causa dei differenti compilatori dei due linguaggi.

- Uno stesso programma che riceve in input 100 nomi (come stringhe) potrebbe impiegare k secondi se ciascuna stringa è formata da un singolo carattere e 200 k secondi se ciascuna stringa è formata da 200 caratteri.

11.1.2. Modello di costo

Una misura del costo computazionale soddisfacente deve:

- basarsi su un modello di calcolo astratto, indipendente dalla particolare macchina
- svincolarsi dalla configurazione dei dati in ingresso, ad esempio basandosi sulle configurazioni più sfavorevoli (caso peggiore), così da garantire che le prestazioni nei casi reali saranno al più costose quanto il caso analizzato
- essere una funzione (non un numero!!!) della dimensione dell'input (la configurazione l'abbiamo fattorizzata via considerando il caso peggiore)
- essere asintotica, cioè fornire una idea dell'andamento del costo all'aumentare della dimensione dell'input(si noti che essere troppo precisi non avrebbe senso visto che non consideriamo la macchina effettiva che esegue il calcolo)

Cioè siamo interessati al *costo asintotico nel caso peggiore*.

11.1.3. Modello di costo

Vogliamo definire un modello di costo (cioè del tempo di esecuzione di un programma) che sia quindi indipendente dall'implementazione.

Definiamo una funzione di costo come segue.

- istruzione atomica (operazioni e confronti su tipi di dati primitivi) ha costo costante
- istruzione di ingresso o di uscita (tipo primitivo) ha costo costante
- istruzione di assegnazione ha costo costante
- blocco di istruzioni

```
{
    <istruzione_1>
    ....
    <instruzione_k>
}
```

ha costo pari a $c_1 + \ldots + c_k$, dove c_i è il costo dell'istruzione i-esima del blocco.

- istruzione condizionale

```
if (<condizione>)
    <parte if>
else
    <parte else>
```

ha costo pari a $c + k$ se la condizione è vera oppure $c + h$ se la condizione è falsa, dove c è il costo del calcolo della condizione, h è il costo della parte if e h il costo della parte else.

- istruzione di ciclo (consideriamo una istruzione for, le altre istruzioni di ciclo, come il while, si possono ridurre a queste)

```
for (int i =0; i<=k; i++)
    <corpo-del-ciclo>
```

ha costo pari a $c_1 + k(c_2 + h) + c_3$, dove c_1 e il costo di inizializzazione del ciclo, c_2 è il costo del confronto e dell'incremento che vengono eseguiti ogni volta che il programma entra nel ciclo, h è il costo totale di tutte le istruzioni nel corpo del ciclo, k è il numero di volte che viene eseguito il ciclo, c_3 è il costo dell'ultimo confronto prima dell'uscita dal ciclo. Il costo di un cilco può essere approssimato asintoticamente a $c + kd$, in cui c riassume i costi delle operazioni svolte una sola volta, e d riassume i costi delle operazioni svolte in ogni esecuzione del ciclo.

- invocazione di una funzione ha costo pari al costo di esecuzione di tutte le istruzioni del corpo della funzione.

Denotiamo con $T(n)$ il costo di un programma misurato come funzione della dimensione n dei dati di ingresso.

11.1.4. Analisi per casi

Nella valutazione del costo computazionale dei programmi si distinguono tre casi.

- Caso peggiore: il caso peggiore corrisponde al tempo massimo che richiede un problema di dimensione n.
- Caso migliore: il caso migliore corrisponde al tempo minimo che richiede un problema di dimensione n.
- Caso medio: il caso medio corrisponde al tempo di soluzione medio per un problema di dimensione n e richiede di valutare il programma con diverse configurazioni di input.

Se non si specifica diversamente, si considera il caso peggiore per determinare il costo.

11.1.5. Studio del comportamento asintotico

Il comportamento asintotico del costo di un programma si basa sull'idea di trascurare costanti moltiplicative e termini di ordine inferiore e fornisce quindi un nozione del costo all'aumentare della dimensione dell'input.

Il comportamento asintotico inoltre rende l'analisi del costo insensibile rispetto alle approssimazioni introdotte nel modello di costo adottato e consente di semplificare i calcoli.

Esempi: il costo $an + b$ ha comportamento asintotico lineare, il costo $an^2 + bn + c$ ha comportamento asintotico quadratico, il costo $alog_b n + c$ ha comportamento asintotico logaritmico, ecc.

11.1.6. Notazione O-grande

Definizione. Una funzione $f(n)$ è $O(g(n))$ (che si legge O-grande di g) se e solo se esistono due costanti positive c e n_0, tali che

$$|f(n)| \leq cg(n)$$

per tutti i valori $n \geq n_0$.

La definizione di notazione O-grande ci dice che da un certo valore n_0 in poi, la dimensione di $f(n)$ non supera quella di $g(n)$ dato un certo fattore di scala c.

Considerando O-grande siamo interessati all'andamento asintotico della funzione di costo del programma. Ad esempio, se consideriamo un programma che ha tempo di esecuzione $T(n) = (n+1)^2$, allora $T(n)$ è $O(n^2)$. Infatti se poniamo $c = 2$ e $n_0 = 3$ abbiamo che $(n+1)^2 \geq 2n^2$, per ogni $n \geq 3$.

11.1.7. Costo di un programma/algoritmo

Definizione. Un algoritmo/programma A ha costo $O(g(n))$ se la quantità di tempo (spazio) sufficiente per eseguire A su un input di dimensione n nel caso peggiore è $O(g(n))$.

11.1.8. Funzioni di costo

La notazione O-grande permette di definire le seguenti funzioni di costo.

$O(1)$ funzione di costo costante (che non dipende dai dati in ingresso).

$O(logn)$ funzione di costo logaritmica

$O(n)$ funzione di costo lineare

$O(nlogn)$ funzione di costo $nlogn$ (quasi-lineare)

$O(n^2)$ funzione di costo quadratica

$O(n^k)$ funzione di costo polinomiale quando è limitata da un polinomio del tipo $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, dove k è una costante

$O(k^n)$ funzione di costo esponenziale quando è limitata da una funzione esponenziale k^n, con k costante maggiore di 1

11.1.9. Valutazione semplificata: operazioni dominanti

Sia A un programma con costo $O(f_A(n))$, una operazione si dice *dominante* se, nel caso peggiore, viene ripetuta un numero di volte $g(n) = O(f_A(n))$.

Se un programma A ha una operazione dominante che viene eseguita un numero di volte $O(g(n))$, allora il costo del programma è $O(g(n))$.

Tipicamente per individuare le operazioni dominanti basta esaminare le istruzioni e i confronti eseguiti nei cicli più interni dei programmi, oppure eseguiti nell'ambito delle attivazioni ricorsive.

11.2. Algoritmi di ricerca

Il problema della ricerca di un elemento all'interno di una collezione si definisce nel seguente modo: data una collezione di valori C e un certo valore k, restituire vero se k fa parte della collezione C, falso altrimenti.

11.2.1. Ricerca sequenziale

La ricerca sequenziale è un algoritmo che confronta tutti gli elementi della collezione con l'elemento da cercare.

```
// verifica la presenza del valore k nell'array a
bool ricercaSequenziale(int *a, int n, int k) {
   bool r = false;
   for (int i = 0; i < n && !r; i++)
      r = r || (a[i]==k);
   return r;
}
```

Nota: La condizione !r nel controllo del ciclo for non cambia il caso peggiore e quindi non ha effetto sull'analisi del costo nel caso peggiore.

11.2.1.1. Costo della ricerca sequenziale

Il caso peggiore per la funzione ricercaSequenziale è il caso in cui l'elemento cercato k non occorre nell'array a.

Costo per un array di dimensione n (dimensione dell'input):

inizializzazione i = 0: 1
confronti i < a: n
confronti a[i] == k: n
istruzioni i++: n
istruzione return true: 1
totale: $3n + 3$, cioè $O(n)$

La ricerca sequenziale ha un costo lineare.

Esempio: Sia l'array a = [3, 5, 7 ,11, 17 ,19, 23, 31] e l'elemento k cercato 6, il costo della ricerca sequenziale è proporzionale a 8.

11.2.2. Ricerca binaria

Se la collezione in cui cercare l'elemento è ordinata, si può implementare un algoritmo più efficiente, cioè con costo minore rispetto a quello della ricerca sequenziale.

Sia a un vettore di n elementi $a_0, \ldots, a_{n-1}$ ordinati in modo crescente e k l'intero da cercare in a.

Sia left l'indice del primo elemento di a, right l'indice dell'ultimo elemento di a, med l'indice dell'elemento centrale pari a (left + right)/2.

```
Se right supera left, allora
    la ricerca di k non ha avuto successo: ritorna false
Altrimenti
    Se a[med] = k allora
        abbiamo trovato l'elemento e ritorna true
    Se a[med] < k allora
        cerchiamo k nella metà destra del vettore
        ponendo left = med
    se a[med] > k allora
        cerchiamo k nella metà sinistra del vettore
        ponendo right = med
```

Il codice C della ricerca binaria è il seguente.

```
bool ricercaBinaria(int *a, int n, int k) {
   int left = 0, right = n-1;
   while (left<=right) {
      int med = (left+right)/2;
      if (a[med]==k)
         return true;
      else if (a[med]<k)
         left = med+1;
      else // a[med]>k
         right = med-1;
   }
   return false;
}
```

11.2.2.1. Costo della ricerca binaria

Per valutare il costo della ricerca binaria, consideriamo l'istruzione dominante, che è qualsiasi operazione all'interno del ciclo while: ad esempio, la condizione left <= right oppure istruzione med = (left+right)/2, ecc.

Il caso peggiore si presenta di nuovo quando l'elemento k non è presente nell'array, quindi la funzione termina dopo aver eseguito tutte le iterazioni del ciclo fino al caso in cui la codizione left <= right diventa

falsa. Ad ogni iterazione vengono scartati metà degli elementi dell'array. Infatti nella prima iterazione, gli estremi left e right racchiudono n elementi, nella seconda iterazione sono $n/2$ elementi, nell'iterazione i-esima avremo $n/2^i$ elementi.

Il procedimento continua fino a quando $n/2^i \geq 1$, cioè $2^i \leq n$, ovvero $i \leq log_2(n)$. Il numero di iterazioni è al massimo pari ad $log(n)$ e poiché il costo di ciascuna iterazione è costante, abbiamo che il costo di ricercaBinaria è $O(log(n))$.

Esempio: Sia l'array a = [3, 5, 7 ,11, 17 ,19, 23, 31] e l'elemento k cercato 6, il costo della ricerca binaria è proporzionale a $log_2(8) = 3$.

11.3. Algoritmi di ordinamento

Problema: Data una sequenza di elementi in ordine qualsiasi, fornire una sequenza ordinata degli stessi elementi.

Questo è un problema fondamentale, che si presenta in moltissimi contesti e in diverse forme:

- ordinamento degli elementi di un vettore in memoria centrale
- ordinamento di una collezione in memoria centrale
- ordinamento di informazioni memorizzate in un file su memoria di massa (file ad accesso casuale o sequenziale)

Consideriamo inizialmente il problema di ordinare un array a di n elementi di tipo T (ad esempio, int), secondo un criterio di ordinamento definito su T (ad esempio, la relazione $<$ su dati di tipo int).

11.3.1. Ordinamento per selezione (Selection Sort)

Esempio: Ordinamento di una pila di carte:

```
seleziona la carta più piccola e mettila in prima
posizione seleziona la più piccola tra le rimanenti
e mettila in posizione 2 ... quando rimane una sola
carta, mettila in ultima posizione
```

Posso operare in maniera simile su un array di interi:

```
individua il valore minimo e
      scambia tale valore con quello in posizione 1
```

```
individua il valore minimo a partire da posizione 2 e
      scambialo con quello in posizione 2
...
individua il valore minimo a partire da posizione k e
      scambialo con quello in posizione k
fermati quando si arriva all'ultimo elemento
      che sarà il più grande in ultima posizione
```

L'algoritmo di ordinamento per selezione si può quindi descrivere nel seguente modo:

```
for (i=0; i<n-1; i++)
  trova il valore più piccolo da i a n-1,
        sia esso in posizione jmin
  scambia gli elementi in posizione i e jmin
```

Implementazione

```
void selectionSort(int* a, int n) {
  for (int i=0; i<n-1; i++) {
    // trova il piu' piccolo elemento da i a n-1
    int jmin = i;
    for (int j=i+1; j<n; j++) {
      if (a[j]<a[jmin])
        jmin = j;
    }
    // scambia gli elementi i e jmin
    int aux = a[jmin];
    a[jmin] = a[i];
    a[i] = aux;
  }
}
```

11.3.1.1. Ordinamento per selezione: costo

La funzione selectionSort è implementata usando due cicli annidati. Il ciclo esterno è ripetuto $n-1$ volte, mentre il ciclo interno ha un costo che dipende dalla dimensione della porzione di array considerata. Infine le operazioni di confronto e di scambio hanno costo costante.

Le operazioni di confronto vengono effettuate sempre indipendentemente dallo stato dell'array in input. Anche se l'array è inizialmente già

ordinato, ogni valore viene comunque confrontato con tutti gli altri. Il numeri di scambi invece dipende dall'array in input e può essere pari a 0, se l'array è già ordinato, o pari a n se l'array è ordinato in ordine inverso.

Non esiste un caso peggiore per quanto riguarda il numero di confronti, mentre per il numero di scambi il caso peggiore si presenta quando l'array è ordinato in ordine inverso.

Il numero di totale di operazioni di confronto è dato da:

$$(n-1)+(n-2)+.....+2+1=\frac{n(n-1)}{2}$$

Quindi il costo di selectionSort è $T(c_1(n^2-n)/2+c_2n)$, dove c_1 indica il costo delle operazioni del ciclo più interno (confronti e assegnazioni), mentre c_2 è il costo delle operazioni di scambio. Il costo asintotico è $O(n^2)$.

11.3.2. Ordinamento a bolle (Bubble Sort)

L'ordinamento a bolle si basa sulla seguente proprietà: se tutte le coppie di elementi adiacenti sono ordinate, allora tutta la sequenza è ordinata.

Il procedimento di ordinamento a bolle è il seguente:

```
Operazione base:
    se due elementi adiacenti non sono ordinati
        scambiali
Metodo:
    verifica se la collezione è ordinata e
        scambia tutte le coppie non ordinate
```

Algoritmo:

```
for (i=0; i<n-1; i++)
   scambia le coppie di elementi adiacenti non ordinate
   termina se non e' stato effettuato nessuno scambio
```

L'operazione di scambio di tutte le coppie non ordinate in una unica scansione della sequenza può essere eseguita sia nel verso della sequenza che nel verso opposto. Quando eseguita nel verso della sequenza,

l'elemento più grande si troverà in ultima posizione. Quando eseguita nel verso opposto, l'elemento più piccolo si troverà in prima posizione.

In generale, all'i-esimo passo del ciclo esterno gli elementi successivi alla posizione i o quelli fino alla posizione i (a seconda del verso con sui si effettuano le operazioni di scambio) sono stati ordinati.

Implementazione

```
void bubbleSort(int* a, int n) {
  bool ordinato = false;
  for (int i=0; i<n-1 && !ordinato; i++) {
    ordinato = true;
    for (int j=n-1; j>i; j--)
      if (a[j-1] > a[j]) {
        int aux = a[j-1];
        a[j-1] = a[j];
        a[j] = aux;
        ordinato = false;
      }
  }
}
```

Ordinamento a bolle: esempio

```
Collezione  ancora non ordinata:
3 1 4 1 5 9 2 6 5 4

Passo 0: 1 3 1 4 2 5 9 4 6 5
Passo 1: 1 1 3 2 4 4 5 9 5 6
Passo 2: 1 1 2 3 4 4 5 5 9 6
Passo 3: 1 1 2 3 4 4 5 5 6 9
Passo 4: 1 1 2 3 4 4 5 5 6 9

Collezione ordinata:
1 1 2 3 4 4 5 5 6 9
```

11.3.2.1. Ordinamento a bolle: costo

In generale, il ciclo esterno viene eseguito $n-1$ volte, mentre il ciclo interno si riduce di 1 elemento ad ogni passo. Il numero di confronti sarà quindi pari a

$$(n-1)+(n-2)+.....+2+1=\frac{n(n-1)}{2}$$

Il caso peggiore si presenta quando l'array è ordinato in ordine inverso. Se la collezione in input è già ordinata, il costo si riduce a $(n-1)$

in quanto vengono eseguiti solamente $n-1$ confronti e il ciclo esterno viene eseguito 1 sola volta.

L'algoritmo di ordinamento a bolle ha costo $O(n^2)$ nel caso peggiore, e costo $O(n)$ nel caso migliore, ovvero quando la collezione è già ordinata.

11.3.3. Ordinamento per fusione (Merge Sort)

L'algoritmo di ordinamento per fusione (Merge Sort), è un algoritmo ricorsivo che si basa sull'operazione di fusione di due sequenze ordinate in una nuova sequenza ordinata contenente tutti gli elementi delle due sequenze. L'operazione di fusione di due sequenze ordinate si può eseguire con costo lineare.

L'algoritmo per fusione è definito dai seguenti passi.

```
Se n<2 allora l'array è già ordinato. Termina
Altrimenti (n>=2)
   1. Ordina la metà sinistra dell'array
   2. Ordina la metà destra dell'array
   3. Fondi le due parti ordinate in un array ordinato
```

Implementazione

```
// funzione ausiliaria ricorsiva
void mergeSort_r(int* v, int inf, int sup) {
  if (inf < sup) {
    int med = (inf+sup)/2;
    mergeSort_r(v,inf,med);
    mergeSort_r(v,med+1,sup);
    merge(v,inf,med,sup);
  }
}

// funzione principale
void mergeSort(int* v, int n) {
  mergeSort_r(v,0,n-1);
}
```

```
// funzione di fusione ( v[inf, ..., med]  e v[med+1,
    ..., sup] )
void merge(int* v, int inf, int med, int sup) {
  int m = sup-inf+1;
  int a[m]; // array ausiliario per la copia
  int i1 = inf;
  int i2 = med+1;
  int i = 0;
  while ((i1 <= med) && (i2 <= sup)) { // entrambi i
    vettori contengono elementi
    if (v[i1] <= v[i2]) {
      a[i] = v[i1];
      i1++;
      i++;
    }
    else {
      a[i] = v[i2];
      i2++;
      i++;
    }
  }
  if (i2 > sup) // e' finitq prima la seconda parte
    del vettore
    for (int k = i1; k <= med; k++) {
       a[i] = v[k];
      i++;
    }
  else // e' finita prima la prima parte del vettore
    for (int k = i2; k <= sup; k++) {
      a[i] = v[k];
      i++;
    }
  // copiamo il vettore ausiliario nel vettore
    originario
  for(int k = 0; k < m; k++) {
    v[inf+k] = a[k];
  }
}
```

11.3.3.1. Ordinamento per fusione: costo

Per valutare il costo computazionale dell'algoritmo Merge Sort si noti che: 1) ad ogni passo il vettore viene diviso in due parti uguali, sulle quali viene richiamata la funzione ricorsiva; 2) la procedura di fusione di due vettori ordinati di dimensioni $n/2$ è pari a $O(n)$,

Il numero di chiamate ricorsive del Merge Sort per un vettore di n elementi è pari a $logn$. Infatti alla k-esima chiamata ricorsiva la dimensione del vettore sarà $n/2^k$ e il procedimento continua finché $n/2^k \geq 2$, cioè finché $k \leq logn/2$. Ad ogni chiamata ricorsiva viene invocata la

funzione merge che ha costo $O(n)$.

In definitiva l'algoritmo Merge Sort ha costo pari a $O(nlogn)$.

11.3.4. Ordinamento veloce (Quick Sort)

L'ordinamento veloce si basa sul seguente metodo: si prende in esame un elemento x del vettore (detto anche pivot) e, mediante una scansione del vettore stesso, si determina la posizione in cui esso si dovrà trovare nel vettore ordinato (sia essa k); durante la stessa scansione si portano tutti gli elementi più piccoli di x nelle posizioni precedenti k, tutti gli elementi più grandi di x nelle posizioni seguenti k e infine x in posizione k. Quindi si ripete ricorsivamente la procedura di ordinamento sulla porzione di array precedente all'indice k e sulla porzione di array successiva all'indice k.

Esempio:

```
v = 6 12 7 8 5 4 9 3 1
x = 6  ->  k = 4
v' = 5 4 3 1 6 12 7 8 9
ordinamento del sotto-vettore [5 4 3 1]
ordinamento del sotto-vettore [12 7 8 9]
```

La scelta del pivot può essere effettuata secondo diversi criteri. Nell'implementazione che segue viene scelto semplicemente il primo elemento del vettore.

Implementazione

```
// funzione ausiliaria ricorsiva
void quickSort_r(int* v, int inf, int sup) {
  if (inf < sup) {
    int i = inf, j = sup, pivot = v[inf];
    while (i < j) {
      while (v[j] > pivot)
        j--;
      while (i < j && v[i] <= pivot)
        i++;
      if (i < j) {
        // ora v[i] > pivot e v[j] <= pivot e i < j
        // quindi scambio v[i] e v[j]
        int a = v[i];
        v[i] = v[j];
        v[j] = a;
      }
    }
    if (inf != j) {
      // mette il pivot al posto giusto j
      v[inf] = v[j];
      v[j] = pivot;
    }
    if (inf < j-1)
      quickSort_r(v,inf,j-1);
    if (sup > j+1)
      quickSort_r(v,j+1,sup);
  }
}

// funzione principale
void quickSort(int* v, int n) {
  quickSort_r(v,0,n-1);
}
```

11.3.4.1. Ordinamento veloce: costo

Nell'algoritmo di ordinamento Quick Sort il caso migliore si presenta quando ad ogni passo si riesce a partizionare il vettore in due vettori di dimensioni $n/2$. In questo caso il costo è $O(nlogn)$.

Nel caso peggiore, che si verifica ad esempio se il vettore è già ordinato, si ha un costo di $O(n^2)$, in quanto ad ogni passo si esamina un nuovo vettore di dimensione inferiore di uno rispetto a quella precedente.

Pur avendo un costo asintotico peggiore del Merge Sort, il Quick Sort si comporta particolarmente bene quando il vettore è mediamente disordinato, ed è per questo motivo che è noto come ordinamento veloce. Infatti rispetto al Merge Sort, il Quick Sort ha il vantaggio di non effettuare l'operazione di fusione e l'allocazione di memoria aggiuntiva

per memorizzare il risultato intermedio della fusione.