Appunti Algoritmi e Strutture Dati

Luca Seggiani

3 Marzo 2024

1 Introduzione ai programmi ricorsivi

Iniziamo con l'esempio classico del calcolo del fattoriale. Fissato $0! = 1$, la definizione iterativa è:

$$n! = 1 \times 2 \times 3 \times ... \times n \quad n > 0$$

un'altra possibilite definizione, forse più naturale, è quella ricorsiva:

$$n! = n * (n-1)!, \quad n > 0$$

Vediamo i due approcci in pseudocodice:

```
  int fact(int n) {
    if(n == 0)
      return 1;
  int a = 1;
  for(int i = 1; i <= n; i++)
    a = a*i;
  return a;
}
```

```
int factRic(int n) {
  if(n == 0)
    return 1;
  return n * factRic(n-1);
}
```

possiamo dare definizioni ricorsive anche di algoritmi più semplici, come ad esempio la moltiplicazione:

```
int mult(int x, int y) {
int(x == 0)
  return 0;
  return y + mult(x - 1, y);
}
```

proviamo a fare l'unrolling ("srotolamento") di questo algoritmo: poniamo una condizione di partenza mult(3,4).

```
mult(3,4)
  4 + mult(2, 4)
    4 + mult(1, 4)
      4 + mult(0, 4) -- fine ricorsione
-> 3 * 4 = 12
```

prendiamo l'algoritmo per il calcolo di un numero pari:

```
int pari(int x) {
  if(x == 0) return 1;
  if(x == 1) return 2;
  return pari(x-2);
}
```

oppure l'algoritmo di Euclide nella sua formulazione ricorsiva:

```
int MCD(int x, int y) {
  if(x == y) return x;
  if(x < y) return MCD(x, y - x);
  return MCD(x - y, y);
}
```

chiararamente, tutti gli esempi precedentemente riportati rispettano una serie di regole:

- Regola 1: individuare il caso base in cui la funzione è immediatamente definita
- Regola 2: effettuare le chiamate ricorsive su un sottinsieme strettamente minore dei dati
- Regola 3: assicurarsi che la serie di chiamate ricorsive decade sempre su un caso base (in modo da terminare la ricorsione, che in caso contrario proseguirebbe all'infinito).

vediamo un'esempio di funzioni errate, cioè che non rispettano queste regole:

```
int pari_errata(int x) {
  if(x == 0) return 1;
  //fa che non sia dispari! regole 1 e 2 violate!
  return pari_errata(x - 2);
}
```

```
int MCD_errata(int x, int y) {
  if(x == y) return x;
  if(x < y) return MCD_errata(x, y-x);
```

```
  return MCD_errata(x, y);
  //nessuna restrizione del dominio: regola 3 violata!
}
```

2 Programmi ricorsivi su liste

Diamo una definizione di lista:

- Una sequenza vuota è una lista
- un elemento seguito da una lista è una lista

diamo una definizione della struttura Elem*, nodo della mia lista, in pseudocodice C++:

```
struct Elem {
  int inf;
  Elem* next;
}
```

vediamo un algoritmo per calcolare la lunghezza di una lista:

```
int length(Elem* p) {
  if(p == NULL) return 0;
  return 1 + length(p->next);
}
```

oppure un'algoritmo che restituisce il numero di volte che un nodo x compare in una lista:

```
int howMany(Elem* p, int x) {
  if(p == NULL) return 0;
  return (p->inf == x) + howMany(p->next, x);
}
```

altri algoritmi definiti sulle liste possono essere:

```
int belongs(Elem* l, int x) {
  if(l == NULL) return 0;
  if(l->inf == x) return 1;
  return belongs(l->next, x);
}
```

```
void tailDelete(Elem*& l) { //passaggio per riferimento
  if(l == NULL) return;
  if(l-> next == NULL) {
    delete l;
    l = NULL;
  }
  else tailDelete(l->next);
}
```

```
void tailInsert(Elem*& l, int x) {
  if(l == NULL) {
    l = new Elem;
    l->inf = x;
    l->next = NULL;
  }
  else tailInsert(l->next, x);
}
```

```
void append_1(Elem*& l1, Elem* l2) {
  if(l1 == NULL) l1 = l2;
  else append(l1->next, l2);
}
```

```
Elem* append_2(Elem* l1, Elem* l2) {
  if(l1 == NULL)
    return l2;
  l1->next = append(l1->next, l2);
  return l1;
}
```

3 Induzione

Riportiamo brevemente la definizione di induzione sui numeri naturali:

> Sia una certa proprietà P vera per un naturale n_0. Se è vero che $P(n) \Rightarrow P(n+1)$, allora P è vera per qualsiasi naturale $n > n_0$.

L'induzione può essere usata ad esempio per dimostrare che la somma dei primi n naturali è:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

introduciamo anche l'induzione completa:

> Se P vale per ogni naturale $m <= n$, e per ogni n vale per $n+1$, allora P vale per ogni naturale

e l'induzione ben fondata:

> Sia S un certo insieme ben ordinato. Se la proprietà P vale per i minimali di S, e per ogni elemento $E \in S$ P è vera su E se è vera su ogni elemento minore di E, allora P vale su tutto l'insieme S.
> (si noti che il totale abuso di notazione $P(x)$ indica P vera su x).

4 Complessità dei programmi ricorsivi

Facciamo il quadro della situazione, applicando quello che abbiamo visto finora e quello che avevamo già definito sugli algoritmi iterativi. Consideriamo il seguente algoritmo per il calcolo di $n!$:

```
int fact(int x) {
  if(x == 0) return 1;
  else return x * fact(x - 1);
}
```

l'algoritmo non fa altro che definire la relazione di ricorrenza:

$$T(0) = a, \quad T(n) = b + T(n-1)$$

andando a svolgere la serie di operazioni definite dall'algoritmo avremo:

$$T(0) = a$$

$$T(1) = b + a$$

$$T(2) = b + b + a = 2b + a$$

$$T(3) = b + 2b + a = 3b + a$$

$$\dots$$

$$T(n) = nb + a$$

l'ultima funzione, $T(n)$, non dipende da alcun passo ricorsivo e fornisce un'interpretazione della complessità dell'algoritmo (che è O(n)).

Selection sort

Applichiamo quanto visto finora ad un programma più complicato: il selection sort, formulato ricorsivamente.

```
void r_selectionSort(int* A, int m, int i = 0) {
  if(i == m - 1) return;
  int min = i;
  for(int j = i + 1; j < m; j++)
    if(A[j] < A[min]) min = j;
  exchange(A[i], A[min]);
  r_selectionSort(A, m, i + 1);
}
```

la relazione di ricorrenza in questo caso sarà:

$$T(1) = a, \quad T(n) = bn + T(n-1)$$

che potremo svolgere come:

$$T(1) = a$$

$$T(2) = 2b + a$$

$$T(3) = 3b + 2b + a$$

$$...$$

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

5 Introduzione al quicksort

L'algoritmo quicksort è un'algoritmo di ordinamento particolarmente efficiente, schematicamente così definito:

- Scelta di un perno
- Divisione dell'array in 2 sottoinsiemi: elementi minori del perno ed elementi maggiori del perno
- Esegui la stessa operazione, ricorsivamente, sui 2 sottoinsiemi finchè non ne incontri uno di 2 elementi.

nel dettaglio, la divisione in due sottoinsiemi dell'array avviene mediante scambi fra elementi utilizzando due cursori (siano s e d), inizialmente posti, rispettivamente, ad inf e sup. Da qui in poi:

- Finché s $<$ d,
  porta avanti s finche A[s] è minore del perno;
  porta indietro d fino a che A[d] è maggiore del perno;
  scambia A[s] e A[d]; incrementa S e decrementa D.
- Se inf $<$ d ripeti il quicksort sulla parte da inf a d.
- Se s $<$ sup ripeti il quickosrt sulla parte da s a sup.

Vediamo infine lo pseudocodice:

```
1 void quickSort(int A[], int inf = 0, int sup = n -1) {
2   int perno = A[(inf + sup) / 2]; //prendo l'elemento
     centrale come perno
3   int s = inf, d = sup;
4   while(s < d) {
5     while(A[s] < perno) s++;
6     while{A[d] > perno} d--;
```

```
7       if(s > d) break;
8       exchange(A[s], A[d]);
9       s++;
10      d--;
11    }
12
13    if(inf < d)
14      quickSort(A, inf, d);
15    if(s < sup)
16      quickSort(A, s, sup);
17  }
```