Appunti senza pretese di P2+Lab:
Pensare ricorsivamente, programmare iterativamente, parte II: Implementazione iterativa di Mergesort

Alessandro Panconesi
DSI, La Sapienza
via Salaria 113, piano terzo
00198 Roma

In queste dispense vediamo come implementare iterativamente l'algoritmo *mergesort* tramite una pila. Il motivo per cui vogliamo fare questo é duplice:

1. É un ottimo esercizio di programmazione che coinvolge una struttura dati fondamentale quale é la pila;

2. L'algoritmo che vedremo é sostanzialmente il meccanismo usato dai compilatori, ma anche dall'interprete assembler, per l'implementazione della ricorsione.

1 L'albero della ricorsione

Definizione 1 *Dato un vettore di n interi $x[]$, denotiamo con $x[i \dots j]$ il sottovettore che va dalla posizione i-esima alla j-esima, estremi inclusi. La dimensione del vettore da ordinare sará denotata con n.*

Una implementazione C dello schema ricorsivo dell'algoritmo mergesort é riportato in Figura 1. La routine `merge(x, i, m, j)` crea il sottovettore ordinato $x[i \dots j]$ assumendo che i due sottovettori $x[i \dots m]$ e $x[m+1 \dots j]$ siano ordinati.

Esercizio 1 *Si dia una implementazione C della funzione* `merge`*.*

Eseguendo l'algoritmo ricorsivo viene generato un *albero delle chiamate ricorsive*. In Figura 2 é rappresentato l'intero albero delle chiamate ricorsive nel caso in cui il vettore da ordinare abbia dimensione $n = 8$.

Per la nostra simulazione l'osservazione cruciale é la seguente: *la sequenza delle chiamate dell'algoritmo ricorsivo corrisponde all'attraversamento in post-ordine dell'albero delle chiamate ricorsive.* In Figura 3 viene riportato l'ordine di visita dato dall'attraversamento in post-ordine nel caso di un albero binario completo di 3 livelli. Nel nostro caso la ricorsione termina alle foglie, anziché con un albero vuoto, e l'algoritmo di esplorazione dell'albero é pertanto il seguente:

- **(Base)** Se il nodo corrente é una foglia non fare nulla (skip), altrimenti

```
void mergesort(int *x, int i, int j) {
   int middle = (i + j) / 2;

   if (i<j) {
      mergesort(x, i, middle);
      mergesort(x, middle+1, j);
      merge(x, i, middle, j);
   }
}
```

Figure 1: Mergesort ricorsivo. `x[]` é un array di interi e si vuole ordinare il sottovettore $x[i \dots j]$. La base della ricorsione si ha quando $i \geq j$ nel qual caso il sottovettore ha 0 oppure 1 elemento ed é pertanto giá ordinato.

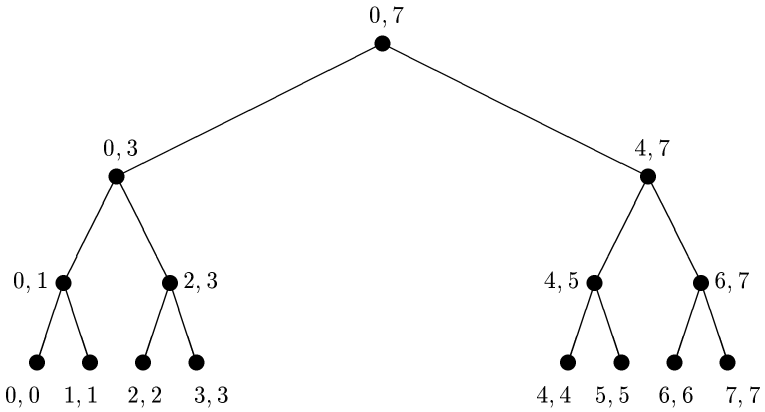

Figure 2: L'albero completo delle chiamate ricorsive nel caso $n = 8$, relativo a $x[0, 7]$. Ogni nodo corrisponde ad una chiamata ricorsiva e la sua etichetta denota gli estremi del sottovettore da ordinare.

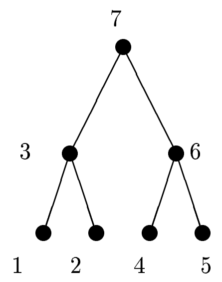

Figure 3: L'ordine in cui vengono visitati i nodi dell'albero delle chiamate. Quando il nodo viene visitato viene effettuata la `merge` relativa al nodo in questione.

- **(Ricorsione)** (a) esplora prima il sottoalbero sinistro, poi (b) esplora il sottoalbero destro ed infine (c) esplora la radice.

Nel nostro caso le visite dei due sottoalberi corrispondono alle chiamate ricorsive, mentre l'esplorazione della radice corrisponde all'esecuzione della routine `merge`.

Esercizio 2 *Etichettare i nodi dell'albero di Figura 2 in post-ordine.*

Notare che lo schema ricorsivo ricalca da vicino quello di una dimostrazione per induzione. Infatti, assumendo di disporre di una funzione `merge` che funzioni correttamente, é praticamente automatico dimostrare per induzione che l'algoritmo mergesort é corretto. L'ipotesi riguardante `merge` é la seguente: se $x[i \ldots m]$ e $x[m+1 \ldots j]$ sono ordinate allora `merge` ordina $x[i \ldots j]$.

Esercizio 3 *Dimostrare rigorosamente per induzione che* mergesort *é corretto.*

2 Eliminazione della ricorsione

Il problema della eliminazione della ricorsione puó quindi essere riformulato in questo modo: *trovare un algoritmo iterativo per l'attraversamento in post-ordine di un albero binario.*

Per fare ció, come detto, useremo una pila. É utile pensare all'esecuzione del programma ricorsivo come ad un processo di visita dei nodi dell'albero, in questo senso. Ogni nodo interno viene visitato 3 volte: (a) la prima volta per spostarsi sul sottoalbero sinistro; completata la visita del sottoalbero sinistro si visita il nodo (b) una seconda volta per spostarsi sul sottoalbero ed infine, dopo che entrambe le visite sono state completate, (c) si effettua una terza ed ultima visita per effettuare la `merge`. Pertanto la pila ed i suoi elementi saranno di questo tipo:

```
void mergesort(int *x, int dim) {
   Stack s = NULL;
   int middle;

   push(&s, newStackEl(0,dim-1,0));
   while (s != NULL) {
      middle = (s->left + s->right) / 2;
      if (!(s->left < s->right))
         free(pop(&s));
      else if (s->nrVisits == 0) {
         s->nrVisits++;
         push(&s, newStackEl(s->left,middle,0));
      }
      else if (s->nrVisits == 1) {
         s->nrVisits++;
         push(&s, newStackEl(middle+1,s->right,0));
      }       else { // s->nrVisits == 2
         merge(x, s->left, middle, s->right);
         free(pop(&s));
      }
   }
}
```

Figure 4: Implementazione di `mergesort`

```
typedef struct Element {
   int left, right, nrVisits;
   struct Element *next;
   } Element
}
typedef Element *Stack;
```

I parametri `left` e `right` denotano gli estremi del sottovettore da ordinare, mentre `nrVisists` assumerá i valori $0, 1, 2$ e servirá per denotare il numero di volte che il nodo é stato visitato. Denoteremo con $(<\ell, r>, v)$ un generico elemento della nostra pila (o stack). L'algoritmo di visita dell'albero per un vettore $x[]$ di n elementi é il seguente.

- Inizializza il processo eseguendo una push dell'elemento $(<0, n-1>, 0)$;
- **(loop)** Sia $(<\ell, r>, v)$ l'elemento in cima allo stack e sia $m := (\ell + r)/2$ (divisione intera).
  1. Se $\ell \geq r$ allora esegui una pop (il nodo in questione é una foglia);
  2. Altrimenti, se $v = 0$ allora incrementa v ed esegui una push dell'elemento $(<\ell, m>, 0)$ (fine prima visita);
  3. Altrimenti, se $v = 1$ allora incrementa v ed esegui una push dell'elemento $(<m+1, r>, 0)$ (fine seconda visita);
  4. Altrimenti ($v = 2$), esegui una `merge` dei sottovettori $x[\ell \ldots m]$ ed $x[m+1 \ldots r]$ (terza ed ultima visita). Esegui una pop.
  5. Se la pila é vuota stop, altrimenti esegui di nuovo il loop.

La Figura 4 mostra una implementazione C di questo algoritmo. Il programma completo viene mostrato in Figura 5. Nel programma si noti la funzione `randomPermute`, che genera in modo (pseudo) casuale una permutazione di `dim` elementi. Grazie ad essa é possibile generare vettori "disordinati" di dimensioni qualsiasi.

```
#include <stdio.h>
#include <stdlib.h>

// type declarations
typedef struct Element {
   int left, right, nrVisits;
   struct Element *next;
} Element;
typedef Element *Stack;

void randomPermute(int x[], int dim) {
   int seed = 0, i, r, temp;

   printf("Random seed:"); scanf("%d", &seed); srand(seed);
   for (i=0; i<dim; i++)
      x[i] = i;
   // permute array randomly
   for (i=0; i<dim-1; i++) {
      r = rand() % (dim-i);
      temp = x[i];  x[i] = x[r+i];  x[r+i] = temp;
   }
}

Stack newStackEl(int i, int j, int c) {
   struct Element *p;
   if ((p = (struct Element *) malloc(sizeof(Element))) != NULL) {
      p->next = NULL; p->left = i; p->right = j; p->nrVisits = c;
      return (p);
      }
   else { printf("Fatal memory allocation error!\n"); exit(1); }
}

Stack pop(Stack *ps) {
Stack t;
t = *ps; *ps = (*ps)->next; return t;
}

void push(Stack *ps, struct Element *p) {
p->next = *ps; *ps = p;
}

void merge(int x[], int i, int m, int j) {
esercizio ...
}

void printVector(char *s, int *x, int i, int j) {
   int k;
   printf("%s...  x[%d, %d] = [", s, i, j);
   for (k=i; k<j; k++) printf("%d ", x[k]);
   printf("] \n");
}

void mergesort(int *x, int dim) {
   Vedere Figura 4
}

main() {
   int dim = 0, *x, i;
   printf("Array dimension:"); scanf("   x = (int *) malloc(sizeof(int) * dim);
   randomPermute(x, dim);
   printVector("before", x, 0, dim);
   recursiveMerge(x, 0, dim-1);
   printVector("after", x, 0, dim);
}
```

Figure 5: Implementazione del mergesort iterativo tramite pila.