Programmazione avanzata ed elementi di Ingegneria del Software

Merge Sort

Merge Sort

InsertionSort usa un approccio **incrementale**: dopo aver ordinato **v**[0… i-1], inserisce **v**[i] nel punto giusto, ottenendo **v**[0… i] ordinato.

Altro approccio è quello **divide et impera**:

- suddividiamo il problema in sottoproblemi simili al problema originale ma di dimensioni più piccole
- risolviamo i sottoproblemi in modo ricorsivo
- combiniamo le soluzioni

Un tipico esempio è il **Merge Sort**

Merge Sort

Divide: divide la sequenza degli n elementi da ordinare in due sottosequenze di n/2 elementi ciascuna.

Impera: ordina le due sottosequenze in modo ricorsivo utilizzando l'algoritmo Merge Sort.

Combina: fonde le due sottosequenze ordinate per generare la sequenza ordinata.

Quando la sequenza da ordinare ha lunghezza 1, la ricorsione si interrompe (ogni sequenza di lunghezza 1 è già ordinata)

Merge Sort

sequenza iniziale

Merge Sort

sequenza ordinata
1 2 2 3 4 5 6 7
fusione
2 4 5 7
1 2 3 6
fusione
fusione
2 5
4 7
1 3
2 6
fusione
fusione
fusione
fusione
5
2
4
7
1
3
2
6
sequenza iniziale

Merge Sort

Il passo chiave dell'algoritmo è nella procedura MERGE

INPUT: array **v**, interi p,q,r:
v[p…q] e **v**[q+1… r] sono ordinati

OUTPUT: array **v**, con **v**[p… r] ordinato

MERGE

Dati due vettori ordinati, fonderli in un unico vettore (ordinato)

MERGE

Dati due vettori ordinati, fonderli in un unico vettore (ordinato)

Step 1
Scorrere i 2 vettori v1, v2
Inserire in v3 l'elemento più piccolo fra v1 e v2

il ciclo termina quando ho esplorato completamente uno dei due vettori

Step 2
ricopio in v3 il vettore non completamente esaminato.

MERGE

```
int main() {
  int nEl1=6, nEl2=9, nEl3=15;
  int vett1[6]={1,3,5,7,9,11};
  int vett2[9]={2,4,6,8,10,12,13,14,15};
  int vett3[15];
  merge(vett1,vett2, vett3, nEl1, nEl2);
  printArray(vett3, nEl3);
```

MERGE

```
void merge(int* v1, int *v2, int *v3, int n1, int n2){
  int i1=0,i2=0,i3=0 ;
  // STEP 1
  do {
    if (v1[i1]<=v2[i2]){
        v3[i3]=v1[i1];
        i3++; i1++;
      }
      else{
        v3[i3]=v2[i2];
        i3++; i2++;
      }
    }
    while (i1<n1 && i2<n2);
```

MERGE

```
// STEP 2
while(i1<n1){
  v3[i3]=v1[i1];
  i3++; i1++;
}
while(i2<n2){
  v3[i3]=v2[i2];
  i3++; i2++;
}
```

MERGE

Posso implementare in modo che operi avendo come ingresso un solo array v

```
void merge(int* v, int p, int q, int r){
    int i;
    int i_left, i_right;
    int len_left=q-p+1;
    int len_right=r-q;
    int v_left[len_left];
    int v_right[len_right];

    for (i=0;i<len_left;i++ ){
        v_left[i]=v[p+i];
    }
    for (i=0;i<len_right;i++ ){
        v_right[i]=v[q+1+i];
    }
…
```

Creo i due vettori **left** e **right** che contengono i sottovettori ordinati

MERGE

Unisco **left** e **right** , mantenendo l'ordinamento, nel sottovettore **v[q … r]**

```
i_left=0;
i_right=0;
i=p;

do {
    if (v_left[i_left]<=v_right[i_right]){
        v[i]=v_left[i_left];
        i++; i_left++;
    }
    else{
        v[i]=v_right[i_right];
        i++; i_right++;
    }
}
while (i_left<len_left && i_right<len_right);
…
```

MERGE

```
 while(i_left<len_left){
   v[i]=v_left[i_left];
   i++; i_left++;
 }
 while(i_right<len_right){
   v[i]=v_right[i_right];
   i++; i_right++;
 }
}
```

Unisco **left** e **right** , mantenendo l'ordinamento, nel sottovettore **v[q … r]**

Complessità computazionale (asintotica)
$O(n)$

[Codice **merge.c**]

Merge Sort

Ora abbiamo merge(v, p, q, r) che unisce i sottoinsiemi individuati dagli indici p, q e q+1, r.

Possiamo realizzare una procedura **merge_sort(a, p, r)** che ordina gli elementi nel sottoarray a[p, r], in questo modo:

- se p==r, il sottoarray ha un solo elemento, ed è ordinato.
- altrimenti, si calcola un **indice q** che separa

a[p… q] ->$\lceil m/2 \rceil$ elementi
a[q+1… r] -> $\lfloor m/2 \rfloor$elementi

e richiama se stessa.

Merge Sort

Ora abbiamo merge(v, p, q, r) che unisce i sottoinsiemi individuati dagli indici p, q e q+1, r.

Possiamo realizzare una procedura **merge_sort(a, p, r)** che ordina gli elementi nel sottoarray a[p, r], in questo modo:

- se p==r, il sottoarray ha un solo elemento, ed è ordinato.
- altrimenti, si calcola un indice q che separa

a[p… q] ->$\lceil m/2 \rceil$ elementi

a[q+1… r] -> $\lfloor m/2 \rfloor$elementi

m/2 arrotondato all'intero superiore

m/2 arrotondato all'intero inferiore

e richiama se stessa.

Merge Sort

```
void merge_sort(int* v, int p, int r) {
    int q;

    if (p<r) {

        q = (p + r) / 2; // DIVISIONE FRA INTERI
        merge_sort(v, p, q);
        merge_sort(v, q + 1, r);
        merge(v, p, q, r);
    }
}
```

Merge Sort

Infine chiamo **merge_sort** passando l'intero vettore.

```
int main() {
    int nEl=7;
    int vett[7]={38, 27, 43, 3, 9, 82, 10};
    printArray(vett, nEl);

    merge_sort(vett, 0, nEl-1);

    printArray(vett, nEl);

    getchar();
    return 0;
}
```

Merge Sort

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort

```
void merge_sort(int* v, int p, int r) {
    int q;

    if (p<r) {

        q = (p + r) / 2; // DIVISIONE FRA INTERI
        merge_sort(v, p, q);
        merge_sort(v, q + 1, r);
        merge(v, p, q, r);
    }
}
```

Suggerimento: con carta e penna seguite il flusso del programma

- la prima chiamata a *merge_sort* divide in due il vettore v
  - richiama se stessa passando la prima metà di v
  - richiama se stessa passando la seconda metà di v

Seguite il flusso delle varie chiamate. Solo alla fine viene chiamata la *merge*.

Merge Sort - Complessità

Analisi dell'algoritmo

Scriviamo una **equazione di ricorrenza** che esprime il tempo di esecuzione totale per un problema di dimensione *n* in funzione dei tempi di esecuzione per input più piccoli.

Merge Sort - Complessità

Dalle **Ricorrenze Fondamentali** sappiamo che

Formula generale divide et impera

A)

$$C_N = 2\,C_{N/2} + 1; N \geq 2; C_1 = 0$$

$$C_N \cong 2N$$

B)

$$C_N = 2\,C_{N/2} + N; N \geq 2; C_1 = 0$$

$$C_N = N\ \log_2 N$$

Merge Sort - Complessità

Dalle **Ricorrenze Fondamentali** sappiamo che

Formula generale divide et impera

A)

$$C_N = 2\,C_{N/2} + 1; N \geq 2; C_1 = 0$$

$$C_N \cong 2N$$

B)

$$C_N = 2\,C_{N/2} + N; N \geq 2; C_1 = 0$$

$$C_N = N\ \log_2 N$$

```
{ …
  merge_sort(v, p, q);
  merge_sort(v, q+1,r);
  merge(v, p, q, r);
}
```

Controllando il nucleo fondamentale dell'algoritmo vediamo che abbiamo

due operazioni di costo $C_{N/2}$ (merge_sort)

e una operazione O(N) (merge)

Dunque siamo nel caso B

Merge Sort - Complessità

Rivediamo i risultati sulla complessità computazionale con un livello di dettaglio leggermente maggiore (la sostanza non cambia)

$T(n)$ è il tempo di esecuzione per un input n

Se *n* è piccolo, la soluzione è a tempo costante.
Se *n* non è piccolo,
    dividiamo il problema in *a* sottoproblemi
    di dimensione *n*/*b*

Inoltre occorre considerare
tempo $Div(n)$ per dividere i sottoproblemi
tempo $Comb(n)$ per combinare le soluzioni

Merge Sort - Complessità

$T(n)$ è il tempo di esecuzione per un input n

Se *n* è piccolo, la soluzione è a tempo costante.
Se *n* non è piccolo,
dividiamo il problema in *a* sottoproblemi
di dimensione *n*/*b*

Inoltre occorre considerare
tempo $Div(n)$ per dividere i sottoproblemi
tempo $Comb(n)$ per combinare le soluzioni

$$T(n) = \begin{cases} c & per\ n\ piccolo \\ a\,T\left(\frac{n}{b}\right) + Div(n) + Comb(n) & per\ n\ grande \end{cases}$$

Merge Sort - Complessità

Semplifico considerando n potenza di 2

$$T(n) = \begin{cases} c & per\ n\ piccolo \\ a\,T\left(\frac{n}{b}\right) + Div(n) + Comb(n) & per\ n\ grande \end{cases}$$

Ogni passo genera due sottoproblemi (a=2)
di dimensione ½ del problema originale (b=2)

Merge Sort - Complessità

$$T(n) = \begin{cases} c & per\ n\ piccolo \\ 2\,T\left(\frac{n}{2}\right) + Div(n) + Comb(n) & per\ n\ grande \end{cases}$$

Se n=1, mergeSort impiega un tempo costante

Se n>1 devo

dividere: calcolo il centro dell'array. $Div(n)$= tempo costante

risolvere i due sottoproblemi: $2\,T\left(\frac{n}{2}\right)$

combinare: (merge) $Comb(n) = O(n)$

Merge Sort - Complessità

$$T(n) = \begin{cases} c & n = 1 \\ 2\,T\left(\frac{n}{2}\right) + Div(n) + Comb(n) & n > 1 \end{cases}$$

Se n=1, mergeSort impiega un tempo costante
Se n>1 devo
dividere: calcolo il centro dell'array. $Div(n)$= tempo costante
risolvere i due sottoproblemi: $2\,T\left(\frac{n}{2}\right)$
combinare: (merge) $Comb(n) = O(n)$

$$T(n) = \begin{cases} c & n = 1 \\ 2\,T\left(\frac{n}{2}\right) + O(n) & n > 1 \end{cases}$$

Merge Sort - Complessità

$$T(n) = \begin{cases} c & n = 1 \\ 2\,T\left(\frac{n}{2}\right) + c \cdot n & n > 1 \end{cases}$$

Merge Sort - Complessità

Un albero binario con k livelli ha n=2^(k-1) foglie

(1 livello: solo root)
(2 livelli: 2^1 foglie)

n foglie ->
$\lfloor \lg_2 n + 1 \rfloor$ livelli,
ciascuno di costo cn

costo totale
$cn \lg_2 n + cn$

$O(n \lg_2 n)$

Merge Sort - Complessità

Relazione tra numero di nodi n e numero di livelli

$n \rightarrow \lfloor \lg_2 n + 1 \rfloor$ livelli,

Esempio:
un albero binario con 4 livelli può avere da 8 a 15 nodi
se $\mathrm{n} = 8 = 2^3 \quad \rightarrow \lfloor \lg_2 n + 1 \rfloor = 4$
se $\mathrm{n} = 15 = 2^4 - 1 \quad \rightarrow \lfloor \lg_2 n + 1 \rfloor = 4$

Esercizi

- Illustrate l’operazione di Merge Sort su un array dato
- Calcolare la complessità f(n) utilizzando come input un array dato. Verificare la correttezza di f(n) contando le istruzioni durante l'esecuzione.
- Modificare Merge Sort in modo che disponga gli elementi in ordine non crescente
- Altri esercizi proposti per gli altri algoritmi di ordinamento

Esercizi

Sebbene merge sort venga eseguito nel tempo Θ(n lg n) nel caso peggiore e insertion sort venga eseguito nel tempo Θ(n^2) nel caso peggiore, **i fattori costanti di insertion sort lo rendono piu` veloce per piccoli valori di n.**

Quindi, ha senso usare insertion sort all'interno di merge sort quando i sottoproblemi diventano sufficientemente piccoli.

Considerate una versione modificata di merge sort in cui n/k sottoliste di lunghezza k siano ordinate utilizzando insertion sort e poi fuse mediante il meccanismo standard di merge sort.

Teorema

La complessità computazionate (tempo) minima di un **qualsiasi** algoritmo di ordinamento per confronto è pari a

O (n log n)