Progetto Corda

Alberto Ferrari

Alberto Ferrari
Ingegneria dell'Informazione, UniPR

Array (ordinamento)

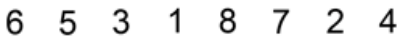
Sort

- L’ordinamento degli elementi di un array avviene considerando il valore della *chiave primaria*
  - Nei nostri esempi le *chiavi* saranno *numeri interi* e la relazione d'ordine totale sarà <=
- Oltre alla valutazione dell’*efficienza* saranno anche considerate le proprietà di
  - *stabilità*: un algoritmo di ordinamento è stabile se non altera l'ordine relativo di elementi dell'array aventi la stessa chiave primaria
  - *sul posto*: un algoritmo di ordinamento opera sul posto se la dimensione delle strutture ausiliarie di cui necessita è indipendente dal numero di elementi dell'array da ordinare

insertion sort

insertion sort

- Al *generico passo* **i** l'array è considerato *diviso* in
  - una *sequenza di destinazione* a[0] ... a[i - 1] *già ordinata*
  - una *sequenza di origine* a[i] ... a[n - 1] *ancora da ordinare*
  - L'obiettivo è di *inserire il valore contenuto in a[i] al posto giusto* nella sequenza di destinazione facendolo scivolare a ritroso, in modo da ridurre la sequenza di origine di un elemento



Esercizio 1

- Scrivere un programma di ordinamento di un array utilizzando l'algoritmo insertion sort
  - *void insertsort(int a[], int n)*
- Calcolare la *complessità computazionale*
  - nel caso ottimo, pessimo e medio
  - Definire la classe di complessità asintotica nel caso medio
- L'algoritmo prodotto è *stabile*? Opera *sul posto*?

insertion sort

6 5 3 1 8 7 2 4

swap function

C

```c
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

insertion sort in C

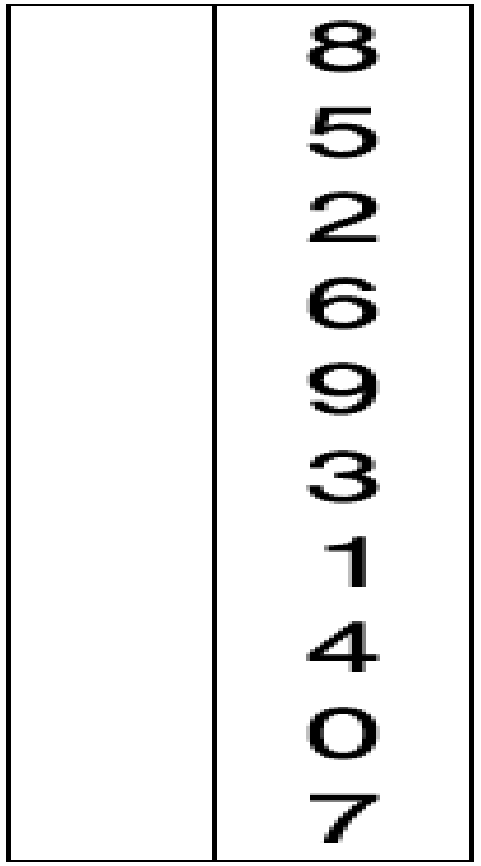
C

```c
void insertsort(int array[], int size) {
  int i, j, app;
  for (i=1; i<size; i++) {
    app = array[i];
    j = i-1;
    while (j>=0 && array[j]>app) {
      array[j+1] = array[j];
      j--;
    }
    array[j+1] = app;
  }
  return;
}
```

selection sort

selection sort

- *Selection sort* è un algoritmo di ordinamento iterativo che, come insertion sort, al generico passo *i* vede l'array diviso in
- una sequenza di *destinazione* a[0] ... a[i - 1] già *ordinata*
- una sequenza di *origine* a[i] ... a[n - 1] ancora *da ordinare*
- L'obiettivo è *scambiare* il valore minimo della seconda sequenza con il valore contenuto in *a[i]* in modo da ridurre la sequenza di origine di un elemento



Esercizio 2

- Scrivere un programma di ordinamento di un array utilizzando l'algoritmo selection sort
  - *void selectsort(int a[], int n)*
- Calcolare la *complessità computazionale*
  - nel caso ottimo, pessimo e medio
  - Definire la classe di complessità asintotica nel caso medio
- L'algoritmo prodotto è *stabile*? Opera *sul posto*?

selection sort in C

C

```c
void selectsort(int array[],int size){
  int i,j,min;
  for (i=0; i<size; i++) {
    min = i;
    for (j=i+1; j<size; j++)
        if (array[min]> array[j])
            min = j;
    if (min != i)
        swap(&array[i],&array[min]);
    }
}
```

bubble sort

Bubblesort

- *bubblesort* è un algoritmo di ordinamento iterativo che, come insertsort, al generico passo *i* vede l'array diviso in
- una sequenza di *destinazione* a[0] ... a[i - 1] già *ordinata*
- una sequenza di *origine* a[i] ... a[n - 1] ancora *da ordinare*
- L'obiettivo è di far *emergere* (come se fosse una bollicina) il valore minimo della sequenza di origine *confrontando e scambiando* sistematicamente i valori di *elementi adiacenti* a partire dalla fine dell'array, in modo da ridurre la sequenza di origine di un elemento

6 5 3 1 8 7 2 4

Esercizio 3

- Scrivere un programma di ordinamento di un array utilizzando l'algoritmo bubble sort
  - *void bubblesort(int a[], int n)*
- Calcolare la *complessità computazionale*
  - nel caso ottimo, pessimo e medio
  - Definire la classe di complessità asintotica nel caso medio
- L'algoritmo prodotto è *stabile*? Opera *sul posto*?

Migliorabile?

- **Se in una iterazione non avvengono più scambi ???**
- Esercizio 4
- modificare l'algoritmo
- rivalutare la complessità computazionale

bubble sort in C

C

```c
void bubble_sort(int array[], int size) {
    int i,last;
    for (last = size - 1; last > 0; last-- ){
        for (i=0; i<last; i++){
            if (array[i]>array[i+1])
                swap(&array[i],&array[i+1]);
        }
    }
}
```

merge sort

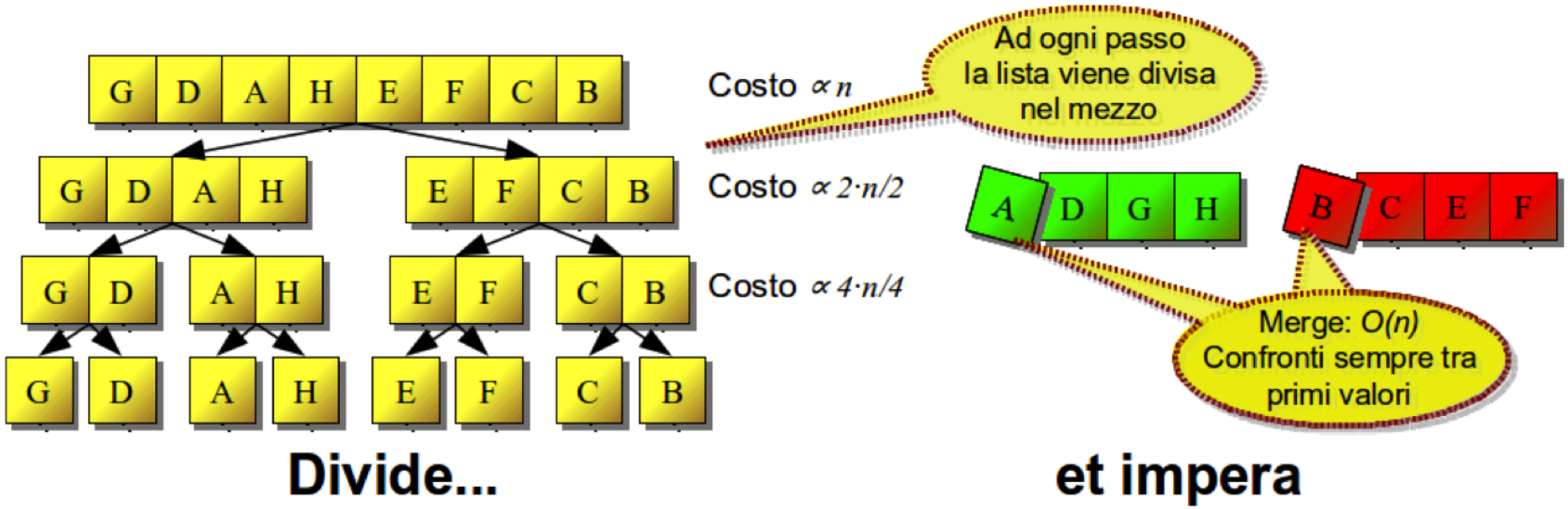
Merge sort

- Algoritmo di ordinamento basato su confronti che utilizza un processo di risoluzione *ricorsivo*, sfruttando la tecnica del *Divide et Impera*, che consiste nella suddivisione del problema in sottoproblemi della stessa natura di dimensione via via più piccola.

Merge sort

- Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata.
- Altrimenti:
- La sequenza viene divisa (*divide*) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda)
- Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo(*impera*)
- Le due sottosequenze ordinate vengono fuse (*combina*).
- Per fare questo, si estrae ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata

Divide et impera



Esempio

- Partenza: [10 3 15 2 1 4 9 0]
- l'algoritmo procede ricorsivamente dividendola in metà successive, fino ad arrivare alle coppie [10 3] [15 2] [1 4] [9 0]
- A questo punto si fondono (merge) in maniera ordinata gli elementi, riunendo le metà: [3 10] [2 15] [1 4] [0 9]
- Al passo successivo, si fondono le coppie di array di due elementi: [2 3 10 15] [0 1 4 9]
- Infine, fondendo le due sequenze di quattro elementi, si ottiene la sequenza ordinata: [0 1 2 3 4 9 10 15]

merge sort in C

C

```c
void merge_sort(int array[], int left, int right) {
  int center; // middle index
  if(left<right) {
     center = (left+right)/2;
     merge_sort(array, left, center);        //sort first half
     merge_sort(array, center+1, right);     //sort first half
     merge(array, left, center, right);      //merge sorted arrays
  }
}
```

funzione merge in C

C

```c
void merge(int array[], int left, int center, int right)
{
  int i = left;                    //index first array
  int j = center+1;                //index second array
  int k = 0;                       //index new temporary array
  int temp[DIM_ARRAY];             //temporary array
  while ((i<=center) && (j<=right)) {
    if (array[i] <= array[j]) { temp[k] = array[i]; i++; }
    else                      { temp[k] = array[j]; j++; }
    k++;
  }
  while (i<=center) { temp[k] = array[i]; i++; k++; }
  while (j<=right)  { temp[k] = array[j]; j++; k++; }
  for (k=left; k<=right; k++){ array[k] = temp[k-left]; }
}
```

Esercizio 5

- Scrivere un programma di ordinamento di un array utilizzando l'algoritmo merge sort
  - *void mergesort(int a[], int left, int right)*
- Calcolare la *complessità computazionale*
- L'algoritmo prodotto è *stabile*? Opera *sul posto*?

Merge sort

- E' *stabile*: il confronto tra array[i] e array[j] effettuato nella fusione ordinata usa <=
- *Non opera sul posto*: nella fusione usa un *array di appoggio* il cui numero di elementi è proporzionale al numero di elementi dell'array da ordinare
- La complessità asintotica è T (n) = O(n · log n)

quick sort

Quicksort

- *Quicksort* è un algoritmo di ordinamento ricorsivo proposto da Hoare nel 1962
- Data una *porzione* di un array e scelto un valore v detto *pivot* contenuto in quella porzione, quicksort divide la porzione in *tre parti*
  - la *prima* composta da elementi contenenti valori $\leq v$
  - la *seconda* (eventualmente vuota) composta da elementi contenenti valori $= v$
  - la *terza* composta da elementi contenenti valori $\geq v$
- poi applica l'algoritmo stesso alla prima e alla terza parte
- Quicksort determina la tripartizione effettuando degli *scambi* di valori ≥ v incontrati a partire dall'inizio della porzione di array con valori ≤ v incontrati a partire dalla fine della porzione di array, in modo tale da spostare i primi verso la fine della porzione di array e gli ultimi verso l'inizio della porzione dell'array

quick sort in C

C

```c
void quick_sort(int array[], int left, int right) {
   int p_index;     // pivot index
   if( left < right ) {
    p_index = partition( array, left, right);
    quick_sort( array, left, p_index-1);
    quick_sort( array, p_index+1, right);
   }
}
```

funzione di partizionamento in C

C

```c
int partition( int array[], int left, int right) {
   int pivot;   //pivot
   int i;       //index first part
   int j;       //index second part
   int temp;
   pivot = array[left];
   i = left;
   j = right+1;
   while( 1 ){
    do ++i; while( array[i] <= pivot && i <= right );
    do --j; while( array[j] > pivot );
    if( i >= j ) break;
    temp = array[i]; array[i] = array[j]; array[j] = temp;
   }
   temp = array[left]; array[left] = array[j]; array[j] = temp;
   return j;
}
```

Quicksort

- *non è stabile* in quanto nel confronto tra array[i] e pivot viene usato < anziché <= e nel confronto tra array[j] e pivot viene usato > anziché >= Questo è inevitabile al fine di ottenere una corretta tripartizione
- Opera *sul posto* in quanto usa soltanto due variabili aggiuntive (pivot e tmp)

Alberto Ferrari
Ingegneria dell'Informazione, UniPR
www.ce.unipr.it/~aferrari/