

Corso “Programmazione 1”

Capitolo 06: Gli Array

Docente: **Marco Roveri** - `marco.roveri@unitn.it`
Esercitori: **Giovanni De Toni** - `giovanni.detoni@unitn.it`
Stefano Berlato - `stefano.berlato-1@unitn.it`
C.D.L.: Informatica (INF)
A.A.: 2021-2022
Luogo: DISI, Università di Trento
URL: <https://bit.ly/2VgfYwJ>



Ultimo aggiornamento: 18 ottobre 2021

Terms of Use and Copyright

USE

This material (including video recording) is intended solely for students of the University of Trento registered to the relevant course for the Academic Year 2021-2022.

SELF-STORAGE

Self-storage is permitted only for the students involved in the relevant courses of the University of Trento and only as long as they are registered students. Upon the completion of the studies or their abandonment, the material has to be deleted from all storage systems of the student.

COPYRIGHT

The copyright of all the material is held by the authors. Copying, editing, translation, storage, processing or forwarding of content in databases or other electronic media and systems without written consent of the copyright holders is forbidden. The selling of (parts) of this material is forbidden. Presentation of the material to students not involved in the course is forbidden. The unauthorised reproduction or distribution of individual content or the entire material is not permitted and is punishable by law.

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

Funzione con parametri di tipo array

- Una funzione può avere un parametro formale del tipo “array di T”
 - Es: `float sum(float v[], int n) { ... }`
 - tipicamente si omette la dimensione (“`float v[]`”, non “`float v[dim]`”)
 - tipicamente associato al numero di elementi effettivamente utilizzati (dimensione virtuale)
- Il corrispondente parametro attuale è un array di oggetti di tipo T
 - Es:
`float a[DIM]; int n;`
`(...)`
`x = sum(a, n);`
- Nel passaggio, gli elementi dell'array **non vengono copiati**
 - N.B. viene copiato **solo l'indirizzo del primo elemento**
 - Equivalente a passare gli elementi dell'array **per riferimento**
 - È possibile impedire di modificarli usando la parola chiave **const**

Nota

Con allocazione statica di array, tipicamente si definiscono array sufficientemente grandi, e poi se ne usa di volta in volta solo una parte.

Passaggio di parametri array costanti

- È possibile definire passaggi di array **in sola lettura** (**passaggio di array costante**)
 - Sintassi: (**const** tipo identificatore [], ...)
 - Es: `int print(const int v[], ...) (...) {...}`
- Passaggio di array: il contenuto dell'array non viene duplicato
⇒ evito possibile spreco di tempo CPU e memoria
- **Non permette di modificare gli elementi di v!**
 - Es: `v[3] = 5; //ERRORE!`
 - ⇒ passaggio di informazione **solo dalla chiamante alla chiamata**
 - ⇒ solo un input alla funzione
- Usato per passare array **in input** alla funzione
 - efficiente (no spreco di CPU e memoria)
 - evita errori
 - permette di individuare facilmente gli input della funzione

Esempi

- passaggio di vettori, i/o di vettori, norma 1, somma, concatenazione di vettori:
{ ARRAY/leggimanipolaarray.cc }

Nota

Se `arr` è un parametro di tipo array di `Type` di una funzione

⇒ `sizeof(arr) = sizeof(Type *)`.

```
void addElement(int arr[], int &d, const int x, const int p, const int N) {  
    cout << "Array_of_size:_ " << sizeof(arr)/sizeof(int) << endl;  
    // Su macchina a 64bit stampa 2 indipendentemente da size di array passato!  
    cout << "===== " << endl;  
    cout << "Adding_" << x << "_at_position_" << p << endl;  
    if (p >= 0 && p <= d && d < N) {  
        for(int i = d; i > p; i--)  
            arr[i] = arr[i-1];  
        arr[p] = x;  
        d++;  
    }  
}  
// ARRAY/sizeof.cc
```

Array e funzioni ricorsive

- E frequente effettuare operazioni **ricorsive** su array
 - Tipicamente il parametro di ricorsione definisce il range dei sotto-array correntemente analizzati
- **somma di array 1 :**
{ ARRAY/array_rec1_nocomment.cc }
- **..., chiamate tracciate:**
{ ARRAY/array_rec1.cc }
- **somma di array 2 :**
{ ARRAY/array_rec2_nocomment.cc }
- **..., chiamate tracciate:**
{ ARRAY/array_rec2.cc }
- **somma di array 3 :**
{ ARRAY/array_rec3_nocomment.cc }
- **..., chiamate tracciate:**
{ ARRAY/array_rec3.cc }

Vedere file `ESERCIZI_PROPOSTI.txt`

Problema: ricerca di un elemento in un array

Quanti passi richiede in media il cercare un elemento in un array di N elementi?

- Con un array **generico**:

- $\approx N/2$ se l'elemento è presente, $\approx N$ se non è presente

⇒ $O(N)$ (un numero proporzionale ad N)

- ES: $N = 1.000.000 \Rightarrow \leq 1.000.000$ passi

- Ricerca lineare:

{ SORT/linear_search.cc }

- Su un array **ordinato**:

- $\leq \lceil \log_2(N) \rceil$ se l'elemento è presente, $\lceil \log_2(N) \rceil$ se non è presente

⇒ $O(\log_2(N))$ (un numero proporzionale al logaritmo di N)

- ES: $N = 1.000.000 \Rightarrow \leq 20$ passi

- Ricerca binaria:

{ SORT/binary_search.cc }

- ... versione ricorsiva:

{ SORT/binary_search_rec.cc }

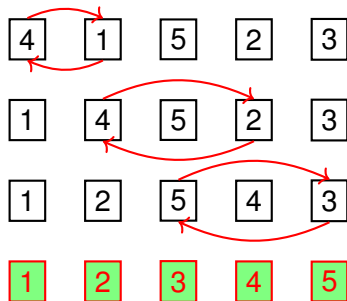
⇒ In molte applicazioni, è vitale mantenere un array ordinato

Moltissimi metodi di ordinamento

- Selection sort
- Insertion sort
- Bubble sort
- Quick sort
- ...

Ordinamento per selezione: Selection Sort

- Cerco elemento più piccolo dell'array e lo scambio con il primo elemento dell'array.
- Cerco il secondo elemento più piccolo dell'array e lo scambio con il secondo elemento dell'array.
- Proseguo in questo modo fintanto che l'array non è ordinato.



Selection sort:

```
{ SORT/selection_sort.cc }
```

Ordinamento per selezione: Selection Sort

```
void selectionsort(int A[], int N) {  
    for (int i = 0; i < N - 1; i++) {  
        int min = i;  
        for(int j = i + 1; j < N; j++) // cerco elemento piu'  
                                        // piccolo nella parte di  
                                        // array ancora da ordinare  
            if (A[j] < A[min]) min = j;  
        swap(A[i], A[min]); // scambio elemento trovato  
                            // con elemento dell'array  
                            // ancora da ordinare  
    }  
}
```

Ordinamento per selezione: Selection Sort

Selection sort

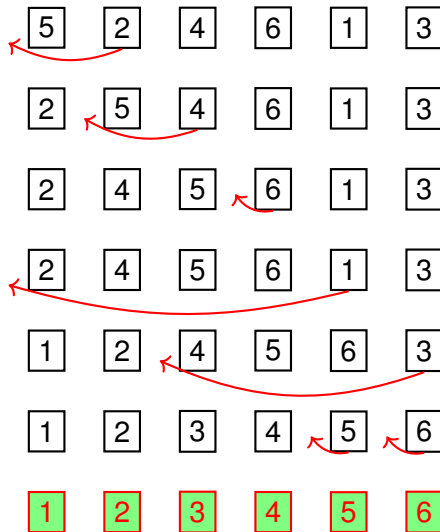
- Relativamente semplice:
- $O(N^2)$ passi in media
- ES: $N = 1.000.000 \implies \leq 1.000.000.000.000 = 10^{12}$ passi (!)

Ordinamento per inserzione: Insertion Sort

- È il metodo usato dai giocatori di carte per ordinare in mano le carte.
- Considero un elemento per volta e lo inserisco al proprio posto tra quelli già considerati (mantenendo questi ultimi ordinati).
 - L'elemento considerato viene inserito nel posto rimasto vacante in seguito allo spostamento di un posto a destra degli elementi più grandi.

Insertion sort:

```
{ SORT/insertion_sort.cc }
```



Ordinamento per inserzione : Insertion sort

```
void insertsort( int A[], int N) {  
    for(int i = N-1; i > 0; i--)    // porto elemento  
                                    // piu' piccolo in A[0]  
        if (A[i] < A[i-1]) swap(A[i], A[i-1]);  
    for(int i = 2; i <= N-1; i++) {  
        int j = i;  
        int v = A[i];  
        while( v < A[j-1] ) {  
            A[j] = A[j-1]; j--;  
        }  
        A[j] = v;  
    }  
}
```

Ordinamento per inserzione: Insertion Sort

Insertion sort

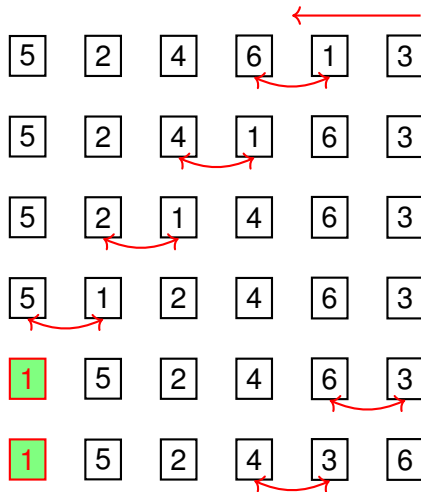
- Relativamente semplice:
- $O(N^2)$ passi in media
- ES: $N = 1.000.000 \implies \leq 1.000.000.000.000 = 10^{12}$ passi (!)

Ordinamento a bolle: Bubble Sort

- Si basa su scambi di elementi adiacenti se necessari, fino a quando non è più richiesto alcuno scambio e l'array risulta ordinato.

bubble sort:

```
{ SORT/bubblesort_nocomment.cc }
```

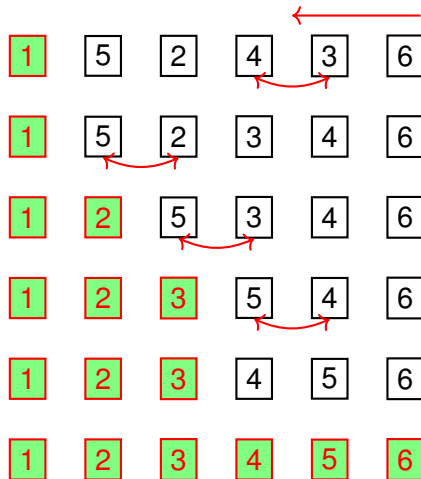


Ordinamento a bolle: Bubble Sort

- Si basa su scambi di elementi adiacenti se necessari, fino a quando non è più richiesto alcuno scambio e l'array risulta ordinato.

bubble sort:

```
{ SORT/bubblesort_nocomment.cc }
```



Metodi di ordinamento: Bubblesort

Bubblesort

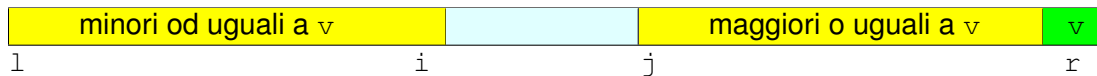
- Relativamente semplice:
 - $O(N^2)$ passi in media
 - ES: $N = 1.000.000 \implies \leq 1.000.000.000.000 = 10^{12}$ passi (!)
-
- bubblesort semplice:
{ SORT/bubblesort_nocomment.cc }
 - ... con passi tracciati:
{ SORT/bubblesort.cc }
 - bubblesort ottimizzato:
{ SORT/bubblesort_opt.cc }

Ordinamento QuickSort

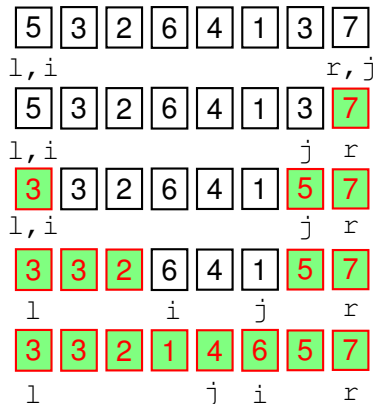
- È un algoritmo di ordinamento del tipo **divide et impera**.
- Si basa su un processo di **partizionamento** dell'array in modo che le seguenti tre condizioni siano verificate:
 - Per qualche valore di i , l'elemento $A[i]$ si trova al posto giusto.
 - Tutti gli elementi $A[0], \dots, A[i-1]$ sono minori od uguali ad $A[i]$.
 - Tutti gli elementi $A[i+1], \dots, A[N-1]$ sono maggiori od uguali ad $A[i]$.
- L'array è ordinato partizionando ed applicando ricorsivamente il metodo ai sotto array.

Ordinamento QuickSort (II)

- Scegliamo arbitrariamente un elemento (e.g. $A[r]$), che chiameremo pivot (o elemento di partizionamento).
- Scandiamo l'array dall'estremità sinistra fino a quando non troviamo un elemento $A[i] \leq A[r]$.
- Scandiamo l'array dall'estremità destra fino a che non troviamo un elemento $A[j] \geq A[r]$.
- Scambiamo $A[i]$ e $A[j]$, e iteriamo.
- Procedendo in questo modo si arriva ad una situazione in cui **tutti** gli elementi a sinistra di i sono minori di $A[r]$, mentre quelli a destra di j sono maggiori di $A[r]$.



Ordinamento QuickSort (III)



return i ;

- Partizioniamo a partire da $A[l] = 5$.
- Gli elementi dell'array precedenti ad $A[i]$ sono minori od uguali a 5.
- Gli elementi dopo $A[j]$ sono maggiori od uguali a 5.
- Ripartizioniamo i sotto array da l a $i-1$, e da $i+1$ a r .

Algoritmo QuickSort: (IV)

```
int partition(int A[],
              int l, int r) {
    int i = l-1, j = r, v = A[r];
    while (true) {
        while (A[++i] < v);
        while (v < A[--j])
            if (j == l) break;
        if (i >= j) break;
        swap(A[i], A[j]);
    }
    swap(A[i], A[r]);
    return(i);
}
```

```
void quicksort(int A[], int N) {
    quicksort_aux(A, 0, N-1);
}

void quicksort_aux(int A[],
                   int l,
                   int r) {
    if (r <= l) return;
    int i = partition(A, l, r);
    quicksort_aux(A, l, i-1);
    quicksort_aux(A, i+1, r);
}
```

Metodi di ordinamento: Quicksort

Quicksort

- Complesso
- $O(N \cdot \log_2(N))$ passi in media
- ES: $N = 1.000.000 \implies \leq 20.000.000 = 2 * 10^7$ passi
- quicksort semplice:
 { SORT/quicksort_nocomment.cc }
- ... con passi tracciati:
 { SORT/quicksort.cc }
- quicksort, con randomizzazione:
 { SORT/quicksort_rand.cc }

Nota

Quicksort algoritmo intrinsecamente ricorsivo!

Algoritmo di ordinamento: Shell Sort

- La lentezza dell'algoritmo Insertion Sort risiede nel fatto che le operazioni di scambio avvengono solo tra elementi contigui.
 - Esempio: se l'elemento più piccolo è in fondo all'array, occorrono N scambi per posizionarlo al posto giusto.
- Per migliorare questo algoritmo è stato pensato l'algoritmo Shell Sort.
- L'idea è quella di organizzare l'array in modo che esso soddisfi la proprietà per cui gli elementi aventi tra loro distanza h costituiscono una sequenza ordinata, indipendentemente dall'elemento di partenza.
 - Se si applica l'algoritmo con una sequenza di h che termina con 1, si ottiene un file ordinato.

```
void ShellSort(int A[],  
               int l, int r)  
{  
    int h;  
    for(h = 1; h <= (r-1)/9;  
        h = 3*h+1);  
    for( ; h > 0; h /= 3)  
        for(int i = l+h;  
            i <= r; i++) {  
            int j = i;  
            int v = A[i];  
            while((j >= l + h) &&  
                (v < A[j-h])) {  
                A[j] = A[j-h];  
                j = j - h;  
            }  
            A[j] = v;  
        }  
}
```


Shell Sort

- L'implementazione proposta $O(N^{3/2})$ passi in media
- ES: $N = 1.000.000 \implies \leq 1.000.000.000 = 10^9$ passi (!)
- Usando sequenze particolari di h si possono ottenere prestazioni diverse (e.g. $O(N \cdot \log_2(N)^2)$).

Metodi di ordinamento: Algoritmi a confronto

Algoritmo	$O(..)$
Selection sort	$O(N^2)$
Insertion sort	$O(N^2)$
Bubblesort sort	$O(N^2)$
Quick sort	$O(N \cdot \log_2(N))$
Shell sort	$O(N^{3/2})$
Merge sort*	$O(N \cdot \log_2(N))$

* Da cercare e implementare come esercizio

Algoritmi di sorting a confronto:

{ SORT/sorting_all.cpp }

Altre operazioni su array ordinati

Fusione ordinata di due array ordinati (merging)

- ES: $merge([1\ 3\ 4\ 8], [2\ 3\ 5\ 6]) \Rightarrow [1\ 2\ 3\ 3\ 4\ 5\ 6\ 8]$
- $O(N_1 + N_2)$
- merging:
 { SORT/merge.cc }

Inserimento di un elemento in un array ordinato

- ES: $insert, 5, [1\ 3\ 4\ 8]) \Rightarrow [1\ 3\ 4\ 5\ 8]$
- $O(N)$
- Equivalente a $merge([5], [1\ 3\ 4\ 8])$

Vedere file `ESERCIZI_PROPOSTI.txt`