



Programmazione 1 (Canale O-Z)

Lezione 10: Algoritmi Iterativi di Ordinamento e Ricerca

Docente: Prof. Antonino Furnari

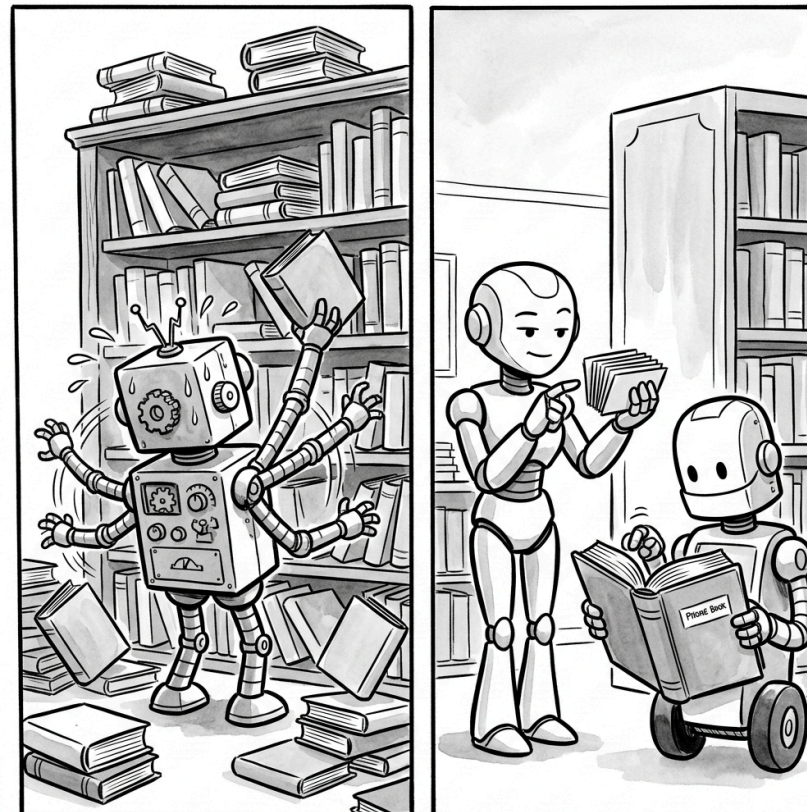
Email: antonino.furnari@unict.it

Sito Web: antoninofurnari.github.io/prog1-2627

🔍 Il Problema dell'Ordinamento

Perché Ordiniamo i Dati?

- Cerchiamo "*Mario Rossi*" in un elenco di **un milione di persone**.
- Elenco **disordinato**: nomi esaminati uno ad uno. Caso peggiore: **1.000.000 di confronti**.
- Elenco **ordinato**: apriamo a metà, scartiamo la parte inutile, ripetiamo. Bastano al massimo **20 confronti**.
- L'ordinamento è la **precondizione** per la ricerca efficiente.



? Con l'elenco disordinato, quante pagine dovrete girare nel caso peggiore?

Lo Strumento dello Scambio (swap)

Tutti gli algoritmi di oggi **scambiano** elementi in memoria. Per non perdere un valore serve una variabile temporanea:

```
void swap(int *a, int *b) {  
    int temp = *a; // Salva il valore puntato da a in una cassaforte temporanea  
    *a = *b;        // Copia il valore puntato da b nell'indirizzo di a  
    *b = temp;      // Sposta il valore temporaneo nell'indirizzo di b  
}
```

- Chiamata: `swap(&arr[i], &arr[j]);`
- Passiamo gli **indirizzi** (Lezione 8): il C passa i parametri **per valore**, senza puntatori scambieremmo solo copie locali.

? Prima di ordinare, il problema opposto: come **mescoliamo** un array in modo equo?

Il Mescolamento Casuale: Shuffling

L'Approccio "Ingenuo" (Sbagliato!)

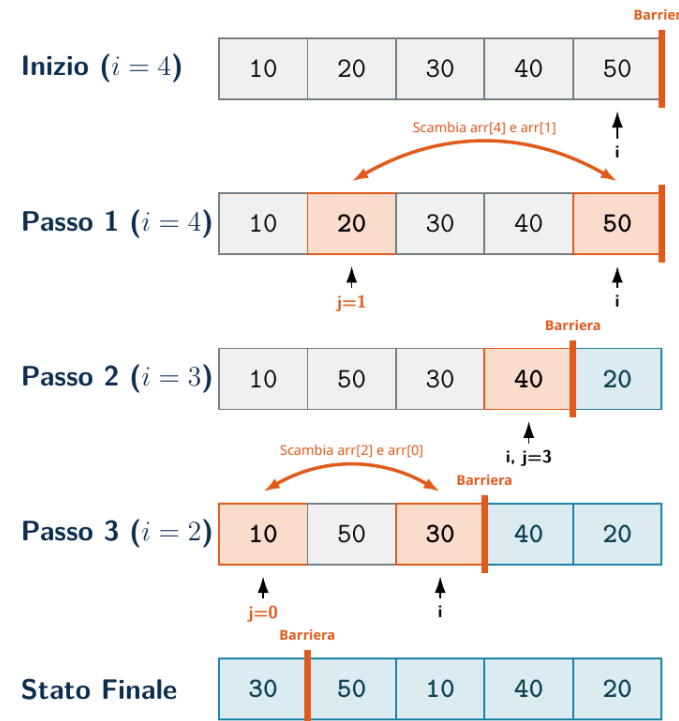
Mescolare in modo **uniforme** serve in simulazioni, videogiochi (mazzo di carte), crittografia.

```
// Algoritmo errato / distorto!
for (int i = 0; i < n; i++) {
    int j = rand() % n; // Scelta tra 0 e n-1
    swap(&arr[i], &arr[j]);
}
```

- N scelte, ciascuna con N possibilità: N^N percorsi possibili.
- Le permutazioni sono $N!$, e N^N non è quasi mai divisibile per $N!$.
- ➡ Alcune permutazioni sono **più frequenti** di altre (bias statistico).

Fisher-Yates Shuffle (Durstenfeld)

- **Idea:** restringere progressivamente lo spazio di scelta.
- Si parte dall'ultimo elemento ($i = n - 1$).
- Si sceglie $j \in [0..i]$ (incluso) e si scambiano `arr[i]` e `arr[j]`.
- Si decrementa i e si ripete fino all'indice 1.
- L'elemento scambiato a destra della barriera è **bloccato**.



L'intero array [10, 20, 30, 40, 50] è da mescolare.
Il cursore i punta all'ultimo elemento (indice 4).

Generiamo un indice casuale $j \in [0..4]$.
Esce $j = 1$.
Scambiamo l'elemento all'indice 4 col valore all'indice 1.

Il 20 è bloccato. Spostiamo i a 3. Generiamo $j \in [0..3]$.
Esce $j = 3$. L'elemento si scambia con se stesso (nessun movimento).

Gli elementi [40, 20] sono bloccati. Spostiamo i a 2.
Scegliamo $j \in [0..2]$. Esce $j = 0$. Scambiamo l'indice 2 con l'indice 0.

Mescolamento completato!
Anche l'elemento all'indice 0 è ora implicitamente bloccato.
La sequenza finale è [30, 50, 10, 40, 20].



Fisher-Yates: Codice C

```
void fisher_yates(int arr[], int n) {  
    for (int i = n - 1; i > 0; i--) {  
        // Sceglie un indice casuale tra 0 e i (incluso)  
        int j = rand() % (i + 1);  
  
        // Scambia solo se gli indici differiscono  
        if (i != j) {  
            swap(&arr[i], &arr[j]);  
        }  
    }  
}
```

- Percorsi possibili: $N \cdot (N - 1) \cdot \dots \cdot 1 = N!$ ➡ ogni permutazione ha probabilità $1/N!$.
- Tempo **lineare** $O(N)$ e *in-place* (nessuna memoria aggiuntiva).

 **Live:** gcc -Wall shuffle.c -o shuffle && ./shuffle mescolando {1, 2, 3, 4, 5}; con `srand(time(NULL))` (Lezione 5) ogni esecuzione dà una permutazione diversa.



Selection Sort: "Trova il Minimo"

- Una **barriera** logica divide l'array: a sinistra la parte **ordinata**, a destra quella **disordinata**. All'inizio è tutto disordinato.
- Ad ogni passata:
 - i. Cerca il **minimo** nella parte disordinata.
 - ii. Scambialo con il **primo** elemento della parte disordinata.
 - iii. Sposta la barriera di una cella a destra.
- **Vantaggio**: al massimo $N - 1$ `swap` fisici (ottimo se scrivere in memoria è costoso).



Selection Sort in Azione

Stato Iniziale



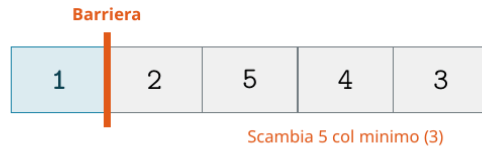
L'intero array [4, 2, 5, 1, 3] è disordinato. La barriera è posizionata all'estrema sinistra. Scansiamo tutto l'array alla ricerca del minimo assoluto.

Passata 1 (Minimo = 1)



Il valore minimo identificato è 1 (all'indice 3).
Lo scambiamo con l'elemento in testa `arr[0]` (4).

Post Passata 1



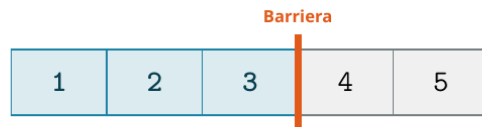
L'elemento 1 è ora nella parte ordinata. La barriera si sposta a destra.
Scansiamo la sotto-area disordinata [2, 5, 4, 3]. Il minimo è 2: si trova già in testa, nessun cambio reale.

Passata 3 (Minimo = 3)



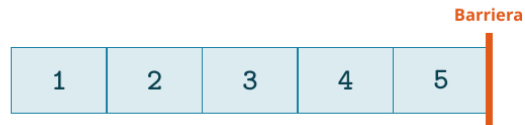
La barriera si trova a destra dell'indice 1. Nell'area disordinata [5, 4, 3], il minimo è 3 (indice 4).
Lo scambiamo con l'elemento all'inizio dell'area disordinata, ovvero il 5 (indice 2).

Post Passata 3



Il 3 è ora inserito nella parte ordinata. La barriera avanza.
Anche se l'array appare visivamente ordinato, per il C gli indici 3 e 4 ([4, 5]) sono ancora nell'area disordinata.

Stato Finale



Fatto! L'algoritmo valuta [4, 5]. Il minimo è 4 (già a posto, lo scambio non serve).
Sposta la barriera.
L'ultimo elemento (5) si blocca automaticamente. Tutto l'array è consolidato ed ordinato!



Selection Sort: Codice C

```
void selection_sort(int arr[], int n) {  
    // La barriera i avanza fino al penultimo elemento  
    for (int i = 0; i < n - 1; i++) {  
        int min_idx = i; // Assumiamo che il minimo sia all'indice i  
  
        // Cerca il minimo nella sotto-area disordinata  
        for (int j = i + 1; j < n; j++) {  
            if (arr[j] < arr[min_idx]) {  
                min_idx = j; // Trovato un nuovo minimo  
            }  
        }  
        // Scambia solo se il minimo non coincide con la testa corrente  
        if (min_idx != i) {  
            swap(&arr[i], &arr[min_idx]);  
        }  
    }  
}
```



Selection Sort: Proviamolo

```
int main(void) {  
    int dati[] = {4, 2, 5, 1, 3};  
    int n = 5;  
  
    selection_sort(dati, n);  
  
    printf("Selection Sort: ");  
    for(int i = 0; i < n; i++) {  
        printf("%d ", dati[i]);  
    }  
    printf("\n");  
    return 0;  
}
```

Selection Sort: 1 2 3 4 5

 **Live:** gcc -Wall selection.c -o selection && ./selection, poi stampare l'array a ogni passata e confrontarlo col diagramma.

Bubble Sort: "Le Bollicine in RAM"

- Confronta ogni elemento con l'**adiacente** destro (`arr[j]` con `arr[j+1]`).
- Se il sinistro è più grande, i due vengono **scambiati**.
- Dopo la prima passata, il **massimo** è nell'ultima cella (congelato).
- Si ripete sui rimanenti $n - 1$ elementi.

```
// Gli ultimi i elementi sono già congelati: il ciclo interno si accorcia
for (int j = 0; j < n - i - 1; j++) {
    if (arr[j] > arr[j + 1]) {
        swap(&arr[j], &arr[j + 1]);
    }
}
```



Bubble Sort in Azione



? Se l'array diventa ordinato a metà dell'opera, il Bubble Sort se ne accorge?



Bubble Sort Ottimizzato: il Flag scambiato

Se in una passata completa **non avviene alcuno scambio**, l'array è già ordinato: usciamo in anticipo.

```
#include <stdbool.h>

void bubble_sort(int arr[], int n) {
    bool scambiato;
    for (int i = 0; i < n - 1; i++) {
        scambiato = false;
        // Gli ultimi 'i' elementi sono già consolidati (congelati)
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
                scambiato = true; // Uno scambio è avvenuto
            }
        }
        if (!scambiato) {
            break; // Nessuno scambio: array ordinato, uscita anticipata!
        }
    }
}
```



Bubble Sort: Proviamolo

```
int main(void) {  
    int dati[] = {5, 1, 4, 2, 3};  
    int n = 5;  
  
    bubble_sort(dati, n);  
  
    printf("Bubble Sort: ");  
    for(int i = 0; i < n; i++) {  
        printf("%d ", dati[i]);  
    }  
    printf("\n");  
    return 0;  
}
```

Bubble Sort: 1 2 3 4 5

 **Live:** gcc -Wall bubble.c -o bubble && ./bubble. Provare su un array già ordinato stampando i prima del break: l'uscita avviene alla prima passata.



Insertion Sort: "L'Ordinamento delle Carte"

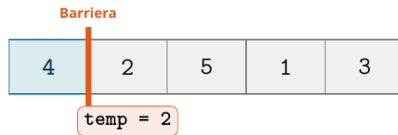
Come ordiniamo le carte in mano:

- `arr[0]` è un sotto-array già ordinato di lunghezza 1.
- Per ciascun elemento successivo:
 - i. Lo **estriamo** in una variabile `temp` (la carta in mano).
 - ii. **Slittamento (Shifting)**: a ritroso, finché gli elementi ordinati sono **maggiori** di `temp`, li facciamo slittare a destra di un posto.
 - iii. **Inserimento**: collochiamo `temp` nella cella rimasta libera.
- Niente `swap`: slittamenti in **un'unica direzione** ➡ meno scritture in RAM.

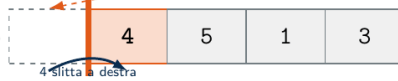


Insertion Sort in Azione

Stato Iniziale



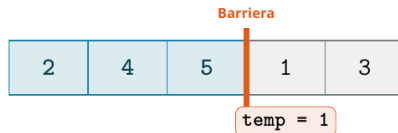
Estrazione & Slittamento



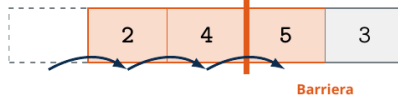
Inserimento



Inserimento 5



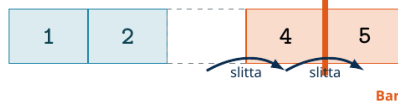
Slittamento Multiplo



Inserimento 1



Slittamento per 3



Stato Finale



L'elemento `arr[0]` (4) è considerato ordinato. La barriera è posizionata a destra dell'indice 0.
Iniziamo a valutare il secondo elemento: 2 (all'indice 1).

Fase 1: Salviamo la "carta" 2 in una variabile temporanea `temp`.
Confrontiamo col 4: poiché $4 > 2$, facciamo slittare il 4 a destra sull'indice 1, liberando l'indice 0.

Fase 2: Depositiamo il valore 2 nell'indice 0 rimasto vacante.
La barriera avanza di una posizione. La sotto-area ordinata ora è `[2, 4]`.

Estrauiamo la "carta" 5. La confrontiamo col 4: poiché $4 < 5$, l'element è già al suo posto.
Nessuno slittamento. Spostiamo la barriera. L'area ordinata è ora `[2, 4, 5]`.

Fase 1: Estrauiamo l' 1. Lo confrontiamo a ritroso con l'area ordinata:
Poiché $5 > 1$, $4 > 1$ e $2 > 1$, facciamo slittare tutti e tre gli elementi a destra di un posto.
L'indice 0 è libero.

Fase 2: Depositiamo l' 1 nella cella iniziale svuotata (indice 0).
L'area ordinata è ora consolidata su `[1, 2, 4, 5]`.

Estrauiamo il 3. Confrontiamo a ritroso: $5 > 3$ (slitta), $4 > 3$ (slitta).
Incontrando il 2 ($2 < 3$) lo slittamento si ferma. L'indice 2 è libero.

Inseriamo il 3 nell'indice 2. La barriera si sposta alla fine dell'array.
Completato! L'intero vettore `[1, 2, 3, 4, 5]` è ordinato.



Insertion Sort: Codice C

```
void insertion_sort(int arr[], int n) {  
    // arr[0] è assunto già ordinato. Partiamo dall'indice 1  
    for (int i = 1; i < n; i++) {  
        int temp = arr[i]; // Estraiamo l'elemento corrente  
        int j = i - 1;  
  
        // FASE 1: Slittamento a destra degli elementi maggiori di temp  
        while (j ≥ 0 && arr[j] > temp) {  
            arr[j + 1] = arr[j];  
            j--;  
        }  
  
        // FASE 2: Inserimento nella posizione liberata  
        arr[j + 1] = temp;  
    }  
}
```

Ricerca Lineare vs Ricerca Dicotomica

Ricerca Lineare

- Funziona su array **ordinati e disordinati**.
- Confronta le celle una ad una da sinistra.
- In media $N/2$ confronti, N nel caso peggiore.

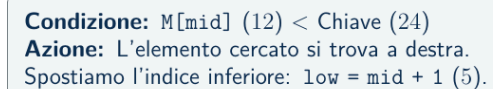
Ricerca Dicotomica

- Richiede un array **rigorosamente ordinato**.
- *Divide et Impera*: confronta la chiave con l'elemento **centrale** `mid`.
- Scarta la metà in cui la chiave non può stare e ripete.

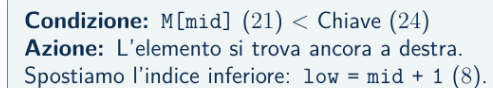
? Pensate un numero da 1 a 100: quante domande "è maggiore di...?" mi servono al massimo? (Solo 7: $2^7 = 128$)



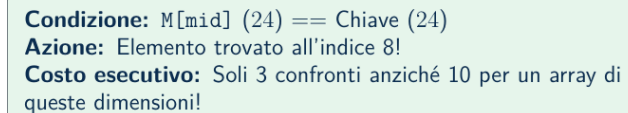
mid



mid



mid (Trovato!)



Prevenire l'Overflow nel Calcolo di `mid`

La formula intuitiva `(low + high) / 2` nasconde un bug classico:

- Se `low` e `high` sono vicini al limite di un `int` ($2^{31} - 1$), la loro somma va in **integer overflow**.
- Il risultato diventa negativo ➡ accesso fuori dall'array e crash di segmentazione.

La soluzione sicura (aritmeticamente equivalente):

$$\text{mid} = \text{low} + \frac{\text{high} - \text{low}}{2}$$

- La differenza `high - low` non supera mai la dimensione dell'area: niente overflow.



Ricerca Dicotomica: Codice C

```
// Restituisce l'indice dell'elemento se trovato, altrimenti -1
int ricerca_dicotomica(int arr[], int n, int chiave) {
    int low = 0;
    int high = n - 1;

    while (low ≤ high) {
        int mid = low + (high - low) / 2; // Calcolo sicuro

        if (arr[mid] == chiave) {
            return mid; // Trovato!
        }
        if (arr[mid] < chiave) {
            low = mid + 1; // La chiave è nella metà destra
        } else {
            high = mid - 1; // La chiave è nella metà sinistra
        }
    }
    return -1; // Chiave non presente
}
```



Ricerca Dicotomica: Proviamola

```
int main(void) {  
    int elenco[] = {2, 4, 5, 8, 12, 15, 18, 21, 24, 27};  
    int n = 10;  
    int chiave = 24;  
  
    int pos = ricerca_dicotomica(elenco, n, chiave);  
  
    if (pos != -1) {  
        printf("Trovato! Il valore %d si trova all'indice: %d\n", chiave, pos);  
    } else {  
        printf("Valore %d non trovato.\n", chiave);  
    }  
    return 0;  
}
```

? Quanti giri del `while` servono per trovare il 24? Seguiamo gli indici passo per passo.

Traccia — Passo 1: Area [0 .. 9]

Indice	0	1	2	3	4	5	6	7	8	9
Valore	2	4	5	8	12	15	18	21	24	27

- `low = 0`, `high = 9` ➡ `mid = 0 + (9 - 0) / 2 = 4`
- `arr[4] = 12 < 24` ➡ la chiave è a destra: `low = mid + 1 = 5`

Traccia — Passo 2: Area [5 .. 9]

Indice	0	1	2	3	4	5	6	7	8	9
Valore	2	4	5	8	12	15	18	21	24	27

- `low = 5`, `high = 9` ➡ `mid = 5 + (9 - 5) / 2 = 7`
- `arr[7] = 21 < 24` ➡ ancora a destra: `low = mid + 1 = 8`

Traccia — Passo 3: Area [8 .. 9]

Indice	0	1	2	3	4	5	6	7	8	9
Valore	2	4	5	8	12	15	18	21	24	27

- `low = 8`, `high = 9` ➡ `mid = 8 + (9 - 8) / 2 = 8`
- `arr[8] = 24 == 24` ➡ **trovato**, `return 8`

Trovato! Il valore 24 si trova all'indice: 8

- Solo **3 confronti** anziché 10.

 **Live:** `gcc -Wall ricerca.c -o ricerca && ./ricerca`, poi cercare una chiave assente (es. 7) e stampare `low`, `mid`, `high` a ogni giro.

Analisi Quantitativa delle Prestazioni

Array ordinato con esattamente $N = 1.048.576$ elementi (circa 1 milione):

- **Ricerca Lineare:** nel caso peggiore **1.048.576 confronti**.
- **Ricerca Dicotomica:** l'area si dimezza ad ogni passo:

$$1.048.576 \rightarrow 524.288 \rightarrow 262.144 \rightarrow 131.072 \rightarrow \dots \rightarrow 1$$

Poiché $2^{20} = 1.048.576$, al massimo **20 confronti!**

? Preferite che lo smartphone esegua 1 milione di operazioni o solo 20 per trovare un contatto in rubrica?

✓ Cosa Abbiamo Imparato Oggi

- `swap` con **puntatori**: base di tutti gli ordinamenti in-place.
- **Fisher-Yates**: mescolamento uniforme ($N!$ percorsi), a differenza dell'approccio ingenuo (N^N).
- **Selection Sort**: trova il minimo, al massimo $N - 1$ scambi.
- **Bubble Sort**: scambi adiacenti, uscita anticipata con il flag `scambiato`.
- **Insertion Sort**: slittamento e inserimento, come le carte in mano.
- **Ricerca Dicotomica**: solo su array ordinati, `mid = low + (high - low) / 2`, $\approx \log_2 N$ confronti.

 **Per casa:** esercizi della Lezione 10 sul sito del corso.

Grazie per l'attenzione!

Sito Web del Corso: antoninofurnari.github.io/prog1-2627