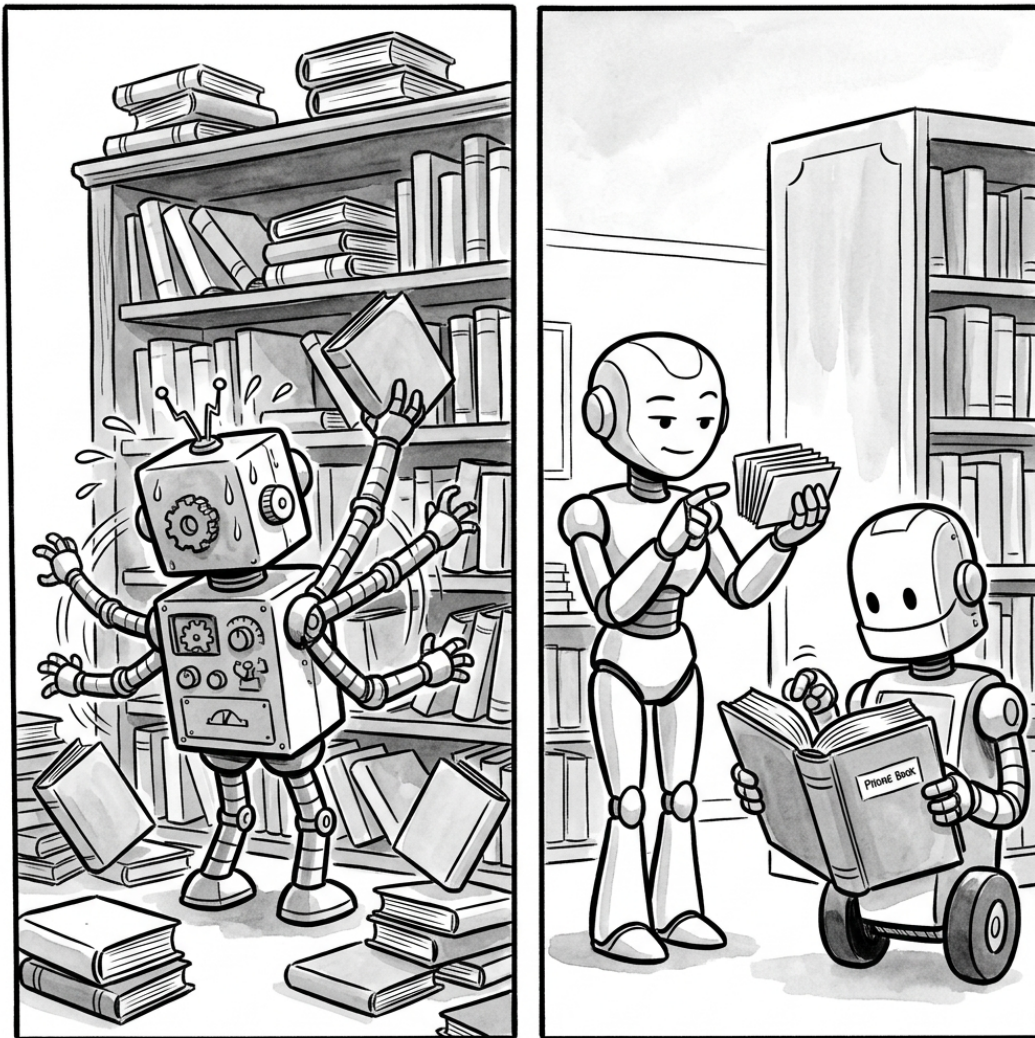


Algoritmi Iterativi di Ordinamento e Ricerca



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

Algoritmi Iterativi di Ordinamento e Ricerca

Nelle lezioni precedenti abbiamo visto come allocare, scansionare e manipolare dati in memoria RAM organizzandoli in strutture sequenziali (gli array monodimensionali e le matrici multidimensionali). Tuttavia, la vera utilità dei dati risiede nella nostra capacità di elaborarli in modo efficiente. In questa lezione affronteremo uno dei pilastri fondamentali dell'informatica: **l'ordinamento dei dati**. Impareremo tre algoritmi iterativi classici (**Selection Sort**, **Bubble Sort** e **Insertion Sort**) analizzando come lavorano sui puntatori e indici a livello fisico, e scopriremo come l'ordinamento sia la chiave per abilitare la **Ricerca Dicotomica (Binary Search)**, una delle ottimizzazioni algoritmiche più potenti ed eleganti della programmazione.

1.1 Il Ruolo dell'Ordinamento e lo Swap

1.1.1 Perché Ordiniamo i Dati?

Per comprendere l'importanza dell'ordinamento, immaginiamo uno scenario reale: cercare un nominativo specifico all'interno di un elenco telefonico di un milione di persone.

- Se l'elenco fosse **disordinato**, non avremmo altra scelta se non quella di esaminare i nomi uno ad uno, dal primo all'ultimo. Nel caso peggiore (nominativo assente o all'ultima pagina), dovremmo effettuare esattamente **1 milione di confronti**.
- Se l'elenco è **ordinato alfabeticamente**, possiamo individuare il nome in pochissimi secondi aprendo l'elenco a metà, scartando la porzione non utile e ripetendo il processo. Come vedremo, per un milione di persone basteranno al massimo **20 confronti**.

L'ordinamento è quindi la preconditione fondamentale che trasforma la ricerca di un'informazione da un processo estenuante a una frazione infinitesima di secondo.

1.1.2 La Funzione swap con Puntatori

Tutti gli algoritmi di ordinamento che analizzeremo si basano su un'operazione fondamentale: lo scambio (*swap*) di posizione tra due elementi in memoria. Per evitare che la copia sovrascriva un valore cancellandolo, dobbiamo utilizzare una variabile temporanea di appoggio. Sfruttando i puntatori (come introdotto nella Lezione 8), la funzione si presenta così:

```
1 void swap(int *a, int *b) {  
2     int temp = *a; // Salva il valore puntato da a in una cassaforte  
    ↪ temporanea  
3     *a = *b;      // Copia il valore puntato da b nell'indirizzo di a  
4     *b = temp;    // Sposta il valore temporaneo nell'indirizzo di b  
5 }
```

1.2 Il Mescolamento Casuale: Fisher-Yates Shuffle

Prima di affrontare il problema di mettere in ordine un array, consideriamo il problema opposto: **mescolarlo in modo completamente casuale ed equo**. Questa operazione, nota come *shuffling*, è fondamentale in ambiti quali simulazioni, videogiochi (es. mescolare un mazzo di carte) o generazione di chiavi di cifratura.

1.2.1 Perché lo Shuffling “Ingenuo” è Sbagliato

Un approccio istintivo potrebbe essere quello di scorrere l'array e scambiare ogni elemento con un altro scelto a caso tra tutti gli indici dell'array:

```
1 // Algoritmo errato / distorto!  
2 for (int i = 0; i < n; i++) {  
3     int j = rand() % n; // Scelta tra 0 e n-1  
4     swap(&arr[i], &arr[j]);  
5 }
```

Sebbene sembri corretto, questo metodo genera permutazioni **non uniformi**. In un array di N elementi, ci sono $N!$ permutazioni possibili. L'algoritmo sopra effettua N scelte, ciascuna con N possibilità, per un totale di N^N possibili percorsi. Poiché N^N non è quasi mai divisibile per $N!$, alcune permutazioni saranno statisticamente più frequenti di altre, introducendo una forte distorsione (bias).

1.2.2 L'Algoritmo di Fisher-Yates

L'algoritmo di **Fisher-Yates** (nella versione moderna di Durstenfeld) risolve questo problema riducendo progressivamente lo spazio di scelta.

1. Si posiziona una barriera virtuale che inizialmente esclude l'intero array.
2. Si parte dall'ultimo elemento dell'array (indice $i = n - 1$).
3. Si sceglie un indice casuale j compreso tra 0 e i (incluso).
4. Si scambiano gli elementi in posizione i e j .
5. Si decrementa i di uno e si ripete il processo fino all'indice 1.

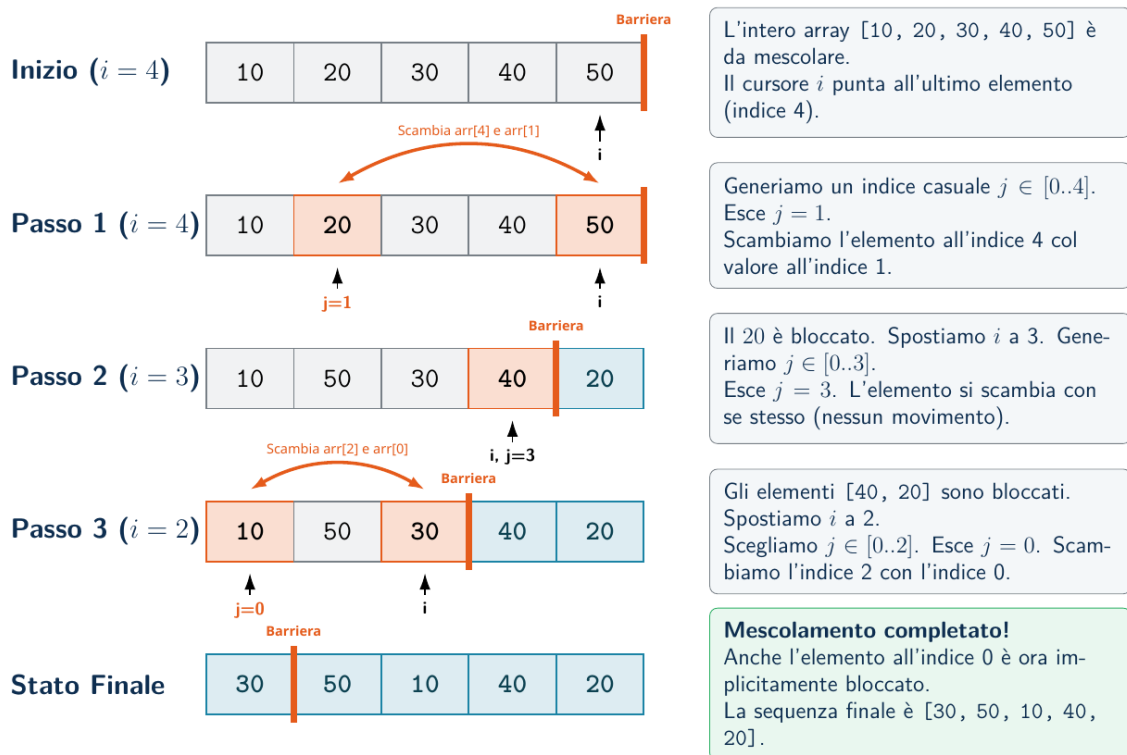


Figura 1.1: Algoritmo Fisher-Yates in azione

A ogni passo l'elemento scambiato finisce a destra della barriera rossa ed è bloccato; lo spazio di scelta si restringe esattamente a $i + 1$ elementi. Poiché il numero di percorsi possibili è esattamente $N \cdot (N - 1) \cdot \dots \cdot 2 \cdot 1 = N!$, ogni permutazione ha esattamente probabilità $1/N!$ di verificarsi. L'algoritmo opera in tempo lineare $O(N)$ e *in-place* (senza memoria aggiuntiva).

Codice C del Fisher-Yates Shuffle:

```

1 void fisher_yates(int arr[], int n) {
2     for (int i = n - 1; i > 0; i--) {
3         // Sceglie un indice casuale tra 0 e i (incluso)
4         int j = rand() % (i + 1);
5
6         // Scambia solo se gli indici differiscono
7         if (i != j) {
8             swap(&arr[i], &arr[j]);
9         }
10    }
11 }

```

1.3 I Tre Algoritmi di Ordinamento Iterativi



Figura 1.2: Selection Sort in Azione

1.3.1 Selection Sort: “Trova il Minimo”

La strategia del **Selection Sort** è estremamente intuitiva:

1. Immaginiamo una barriera logica che separa l'array in due parti: a sinistra gli elementi già ordinati, a destra quelli ancora disordinati. All'inizio l'intero array è disordinato.
2. Scansioniamo l'area disordinata alla ricerca del valore minimo.
3. Scambiamo questo valore minimo con l'elemento che si trova all'estrema sinistra dell'area disordinata.
4. Spostiamo la barriera di un passo verso destra e ripetiamo il processo per la parte rimanente dell'array.

Codice C Completo del Selection Sort:

```

1  #include <stdio.h>
2
3  void swap(int *a, int *b) {
4      int temp = *a;
5      *a = *b;
6      *b = temp;
7  }
8
9  void selection_sort(int arr[], int n) {
10     // La barriera 'i' avanza fino al penultimo elemento dell'array
11     for (int i = 0; i < n - 1; i++) {
12         int min_idx = i; // Assumiamo temporaneamente che il minimo sia
13         // all'indice 'i'
14
15         // Scansione della sotto-area disordinata (a destra di 'i')
16         for (int j = i + 1; j < n; j++) {

```

```

16         if (arr[j] < arr[min_idx]) {
17             min_idx = j; // Trovato un nuovo minimo, memorizziamo l'
           ↪ indice
18         }
19     }
20
21     // Se il minimo identificato non coincide con la testa corrente,
           ↪ scambiamo
22     if (min_idx != i) {
23         swap(&arr[i], &arr[min_idx]);
24     }
25 }
26 }
27
28 int main(void) {
29     int dati[] = {4, 2, 5, 1, 3};
30     int n = 5;
31
32     selection_sort(dati, n);
33
34     printf("Selection Sort: ");
35     for(int i = 0; i < n; i++) {
36         printf("%d ", dati[i]);
37     }
38     printf("\n");
39     return 0;
40 }

```

1.3.2 Bubble Sort: “Le Bollicine in RAM”

L'idea fondamentale del **Bubble Sort** è far risalire gli elementi più grandi verso l'estremità destra dell'array (come bollicine di anidride carbonica in un bicchiere d'acqua):

1. Si confronta ciascun elemento con il suo adiacente destro ($arr[j]$ con $arr[j+1]$).
2. Se l'elemento di sinistra è maggiore di quello di destra, vengono scambiati.
3. Al termine della prima passata completa, l'elemento massimo assoluto avrà necessariamente raggiunto l'ultima cella dell'array.
4. Si ripete la passata per i rimanenti $n - 1$ elementi, congelando progressivamente gli elementi massimi a destra.

L'Ottimizzazione del Flag scambiato

Nella sua versione base, il Bubble Sort esegue sempre tutti i cicli anche se l'array viene ordinato a metà dell'opera. Possiamo ottimizzarlo introducendo una variabile booleana `scambiato`: se durante una passata completa non avviene alcun scambio fisico, significa che l'array è già perfettamente ordinato ed è inutile continuare.



Figura 1.3: Bubble Sort in Azione

Codice C Completo del Bubble Sort Ottimizzato:

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  void swap(int *a, int *b) {
5      int temp = *a;
6      *a = *b;
7      *b = temp;
8  }
9
10 void bubble_sort(int arr[], int n) {
11     bool scambiato;
12     for (int i = 0; i < n - 1; i++) {
13         scambiato = false;
14
15         // Gli ultimi 'i' elementi sono già consolidati (congelati)
16         for (int j = 0; j < n - i - 1; j++) {
17             if (arr[j] > arr[j + 1]) {
18                 swap(&arr[j], &arr[j + 1]);
19                 scambiato = true; // Uno scambio ? avvenuto
20             }
21         }
22
23         // Se non ci sono stati scambi, l'array è ordinato. Uscita
24         ⇨ anticipata!
25         if (!scambiato) {
26             break;
27         }
28     }
29
30 int main(void) {
31     int dati[] = {5, 1, 4, 2, 3};
32     int n = 5;

```

```

33
34     bubble_sort(dati, n);
35
36     printf("Bubble Sort: ");
37     for(int i = 0; i < n; i++) {
38         printf("%d ", dati[i]);
39     }
40     printf("\n");
41     return 0;
42 }

```

1.3.3 Insertion Sort: “L’Ordinamento delle Carte”

L’**Insertion Sort** ricalca fedelmente il modo in cui ordiniamo le carte da gioco che teniamo in mano:

1. Consideriamo il primo elemento (`arr[0]`) come un sotto-array già ordinato di lunghezza 1.
2. Esaminiamo il secondo elemento. Lo “estrai” memorizzandolo in una variabile temporanea `temp` (la nostra carta in mano).
3. **Fase di Slittamento (Shifting):** Confrontiamo `temp` a ritroso con gli elementi della porzione ordinata a sinistra. Finché questi elementi sono maggiori di `temp`, li facciamo slittare a destra di una posizione (sovrascrivendo la cella successiva).
4. **Fase di Inserimento:** Non appena troviamo un elemento minore o uguale a `temp` (o raggiungiamo l’inizio dell’array), inseriamo la nostra “carta” nella posizione vuota rimasta libera.

Nota

Nota A differenza del Selection/Bubble Sort che lavorano effettuando scambi diretti (swap), l’Insertion Sort lavora spostando i dati tramite slittamenti progressivi in un’unica direzione, riducendo il numero di accessi in scrittura sulla memoria RAM.

Codice C Completo dell’Insertion Sort:

```

1  #include <stdio.h>
2
3  void insertion_sort(int arr[], int n) {
4      // arr[0] ? assunto gi? ordinato. Partiamo dall'indice 1
5      for (int i = 1; i < n; i++) {
6          int temp = arr[i]; // Estraiamo l'elemento corrente
7          int j = i - 1;
8
9          // Spostiamo verso destra tutti gli elementi maggiori di temp
10         while (j >= 0 && arr[j] > temp) {
11             arr[j + 1] = arr[j]; // Slittamento fisico a destra

```



Figura 1.4: Insertion Sort in Azione

```

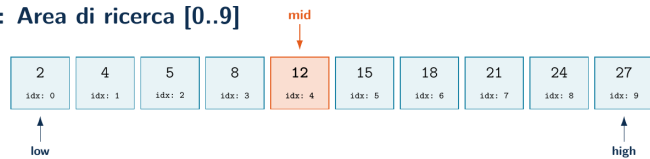
12         j--; // Continuiamo la scansione a ritroso
13     }
14
15     // Inseriamo temp nella posizione corretta liberata dallo
16     ↪ slittamento
17     arr[j + 1] = temp;
18 }
19
20 int main(void) {
21     int dati[] = {4, 2, 5, 1, 3};
22     int n = 5;
23
24     insertion_sort(dati, n);
25
26     printf("Insertion Sort: ");
27     for(int i = 0; i < n; i++) {
28         printf("%d ", dati[i]);
29     }
30     printf("\n");
31     return 0;
32 }

```

1.4 La Ricerca Dicotomica (Binary Search)

1.4.1 Ricerca Lineare vs Dicotomica

La ricerca di un elemento in un array può essere condotta in due modi:

Passo 1: Area di ricerca [0..9]

Condizione: $M[mid] (12) < Chiave (24)$
Azione: L'elemento cercato si trova a destra.
 Spostiamo l'indice inferiore: $low = mid + 1 (5)$.

Passo 2: Area di ricerca [5..9]

Condizione: $M[mid] (21) < Chiave (24)$
Azione: L'elemento si trova ancora a destra.
 Spostiamo l'indice inferiore: $low = mid + 1 (8)$.

Passo 3: Area di ricerca [8..9]

Condizione: $M[mid] (24) == Chiave (24)$
Azione: Elemento trovato all'indice 8!
Costo esecutivo: Soli 3 confronti anziché 10 per un array di queste dimensioni!

Figura 1.5: Ricerca Dicotomica in Azione

- **Ricerca Lineare:** Scorre l'array dall'inizio e confronta ogni cella con la chiave cercata. Funziona su array sia ordinati che disordinati, ma richiede in media $N/2$ confronti e N confronti nel caso peggiore.
- **Ricerca Dicotomica (Binary Search):** Richiede che l'array sia **rigorosamente ordinato**. Adotta una strategia di tipo *Divide et Impera*: confronta la chiave con l'elemento centrale. Se sono diversi, scarta la metà dell'array in cui la chiave non può risiedere e ripete la ricerca nella metà rimanente.

1.4.2 Prevenire l'Overflow nel Calcolo del Punto Medio

Nel calcolare l'indice centrale, la formula intuitiva sarebbe $(low + high) / 2$. Tuttavia, se low e $high$ assumono valori molto grandi (vicini al limite superiore del tipo `int`, pari a $2^{31} - 1$), la loro somma diretta provocherà un **integer overflow**, producendo un valore negativo e causando un crash di segmentazione durante l'accesso all'array.

Per evitare questo bug critico, si adotta la formula aritmeticamente equivalente ma sicura:

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

In questo modo, la sottrazione $high - low$ non potrà mai superare la dimensione dell'area, scongiurando il rischio di overflow.

Codice C della Ricerca Dicotomica:

```
1 #include <stdio.h>
2
3 // Restituisce l'indice dell'elemento se trovato, altrimenti -1
4 int ricerca_dicotomica(int arr[], int n, int chiave) {
5     int low = 0;
6     int high = n - 1;
7 }
```

```
8  while (low <= high) {
9      // Calcolo sicuro del punto medio per evitare overflow numerico
10     int mid = low + (high - low) / 2;
11
12     if (arr[mid] == chiave) {
13         return mid; // Trovato! Restituiamo l'indice
14     }
15
16     if (arr[mid] < chiave) {
17         low = mid + 1; // La chiave si trova nella met? destra
18     } else {
19         high = mid - 1; // La chiave si trova nella met? sinistra
20     }
21 }
22
23 return -1; // Chiave non presente nell'array
24 }
25
26 int main(void) {
27     int elenco[] = {2, 4, 5, 8, 12, 15, 18, 21, 24, 27};
28     int n = 10;
29     int chiave = 24;
30
31     int pos = ricerca_dicotomica(elenco, n, chiave);
32
33     if (pos != -1) {
34         printf("Trovato! Il valore %d si trova all'indice: %d\n", chiave,
35             ↪ pos);
36     } else {
37         printf("Valore %d non trovato.\n", chiave);
38     }
39     return 0;
40 }
```

1.4.3 Analisi quantitativa dell'efficienza

Vediamo numericamente perché la ricerca dicotomica è una svolta ingegneristica per la CPU. Supponiamo di cercare una chiave in un array ordinato contenente esattamente $N = 1.048.576$ elementi (circa 1 milione).

- Con la **Ricerca Lineare**, nel caso peggiore la CPU eseguirà **1.048.576 confronti**.
- Con la **Ricerca Dicotomica**, la dimensione dell'area si dimezza ad ogni passo:

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

Poiché $2^{20} = 1.048.576$, la ricerca si concluderà in al massimo **20 confronti**.

Passare da 1 milione di operazioni a solo 20 riduce drasticamente l'utilizzo di energia, tempo e risorse di calcolo della CPU, rendendo la ricerca dicotomica incredibilmente efficiente su grandi moli di dati.

1.5 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Qual è la condizione essenziale per poter applicare la ricerca dicotomica (Binary Search) su un array?

- A) L'array deve essere allocato dinamicamente nello Heap.
- B) L'array deve essere preventivamente ordinato.
- C) L'array deve contenere solo numeri positivi.
- D) L'array non deve contenere duplicati.

Spiegazione: La ricerca dicotomica si basa sulla certezza matematica che, confrontando la chiave con l'elemento centrale $arr[mid]$, se $arr[mid] < chiave$, allora la chiave non può risiedere nella metà sinistra. Questo principio crolla se gli elementi non sono ordinati.

Risposta corretta: B

Soluzione

Quiz 2

Domanda

2. Quale delle seguenti formule previene il bug di overflow aritmetico nel calcolo del punto medio?

- A) $(low + high) / 2$
- B) $low + (high - low) / 2$
- C) $high - (high - low) / 2$
- D) $(low / 2) + (high / 2)$

Spiegazione: Se low e $high$ sono molto grandi, la loro somma $low + high$ può superare il massimo numero intero rappresentabile ($2^{31} - 1$), andando in overflow (diventando negativa). La sottrazione $high - low$ è invece sempre minore del massimo indice ed è quindi sicura.

Risposta corretta: B

Soluzione

Quiz 3

Domanda

3. Nel Selection Sort, quante operazioni fisiche di swap vengono eseguite al massimo in un array di N elementi?

- A) N^2
- B) $N \log N$
- C) $N - 1$
- D) Nessuna, non usa swap.

Soluzione
Risposta corretta: C
Spiegazione: Il Selection Sort effettua al massimo uno swap per ogni iterazione del ciclo esterno. Poiché il ciclo esterno viene eseguito $N - 1$ volte, il numero massimo di swap eseguiti è esattamente $N - 1$. Questa proprietà lo rende ideale quando scrivere in memoria è molto costoso.

Quiz 4

Domanda

4. Che cosa succede nel Bubble Sort ottimizzato se durante una passata completa dell'array non viene eseguito nessuno swap?

- A) L'algoritmo va in loop infinito.
- B) Il flag scambiato rimane `false` e l'algoritmo termina immediatamente.
- C) L'algoritmo ricomincia dall'inizio azzerando gli indici.
- D) Viene generato un errore a runtime.

Soluzione
Risposta corretta: B
Spiegazione: Se non avviene alcuno scambio, significa che ogni elemento dell'array è già minore o uguale al suo successore. Il flag scambiato (inizializzato a `false`) non viene alterato, e la condizione `if ( !scambiato )` forza l'uscita anticipata dal ciclo (break).

Quiz 5

Domanda

5. Qual è la caratteristica distintiva del meccanismo di Insertion Sort rispetto a Bubble e Selection Sort?

- A) Non esegue confronti tra elementi.
- B) Sposta gli elementi tramite slittamento progressivo (shifting) anziché swap continui.
- C) Funziona solo su stringhe e non su interi.
- D) Richiede un array ausiliario per copiare i dati.

Spiegazione: L'insertion Sort estrae temporaneamente l'elemento corrente in tempo e fa slittare a destra di una posizione tutti gli elementi ordinati maggiori di esso. Non esegue scambi bidirezionali continui (swap), riducendo le scritture in RAM.

Risposta corretta: B

Soluzione

Quiz 6

Domanda

6. Se cerchiamo un elemento inesistente in un array ordinato di 4096 elementi con la ricerca dicotomica, quanti confronti faremo al massimo?

- A) 4096
- B) 2048
- C) 12
- D) 7

Spiegazione: Poiché ad ogni passo l'area di ricerca viene divisa per due, cerchiamo il più piccolo esponente k tale che $2^k \geq 4096$. Dato che $2^{12} = 4096$, l'algoritmo eseguirà al massimo 12 confronti.

Risposta corretta: C

Soluzione

Quiz 7

Domanda

7. Qual è lo scenario peggiore (worst-case) per il Bubble Sort non ottimizzato?

- A) Array già ordinato in senso crescente.
- B) Array ordinato in senso decrescente (inverso).
- C) Array contenente solo elementi identici.
- D) Array vuoto.

Spiegazione: Se l'array è ordinato al contrario, ogni singolo confronto richiederà uno swap per far risalire l'elemento adiacente. L'algoritmo dovrà effettuare il numero massimo di scambi fisici possibili.

Risposta corretta: B

Soluzione

Quiz 8

Domanda

8. Nella funzione `swap(int *a, int *b)`, perché è necessario passare gli indirizzi delle variabili (puntatori)?

- A) Perché il C passa i parametri per valore (copia), quindi senza puntatori modificherebbero solo copie locali delle variabili.
- B) Perché i puntatori rendono l'esecuzione più lenta ma sicura.
- C) Perché lo Stack Frame non permette di scambiare variabili locali.
- D) Perché è imposto dalla sintassi delle funzioni di ordinamento.

Soluzione
Risposta corretta: A
Spiegazione: Il passaggio dei parametri in C avviene per valore. Se passassimo direttamente `swap(x, y)`, la funzione scambierebbe i valori all'interno dei suoi parametri locali `a` e `b`, lasciando del tutto inalterate le variabili originali `x` e `y` del chiamante.

Quiz 9

Domanda

9. Quale algoritmo tra Selection, Bubble e Insertion Sort si adatta meglio ad ordinare elementi che vengono inseriti in tempo reale uno alla volta?

- A) Selection Sort, perché cerca il minimo.
- B) Bubble Sort, perché fa risalire le bollicine.
- C) Insertion Sort, perché inserisce l'elemento al posto giusto slittando solo il necessario.
- D) Nessuno di essi.

Soluzione
Risposta corretta: C
Spiegazione: L'Insertion Sort assume che la porzione a sinistra sia già ordinata. Se arriva un nuovo elemento in coda, basta farlo slittare all'indietro fino alla sua corretta collocazione. È l'approccio ideale per l'ordinamento "in-place" online.

Quiz 10

Domanda

10. Che cosa restituisce la funzione `ricerca\{}_dicotomica` se la chiave cercata è presente più volte nell'array ordinato?

- A) L'indice della prima occorrenza in assoluto.

- B) L'indice dell'ultima occorrenza.
- C) Un indice arbitrario tra le occorrenze della chiave (dipendente da dove si ferma `mid`).
- D) Ritorna `-1` per errore.

Soluzione
Risposta corretta: C
Spiegazione: La Binary Search standard si ferma non appena trova `arr[mid] == chiave`. Se ci sono duplicati, l'indice restituito dipenderà da quale occorrenza si trova ad essere colpita per prima dal punto medio `mid`. Per trovare la prima o l'ultima occorrenza servono varianti specifiche dell'algoritmo.

1.6 Esercizi Pratici

Esercizio 1: Trova l'Indice del Minimo

Domanda

Scrivete una funzione `C int trova_minimo(int arr[], int start, int end)` che restituisca l'indice dell'elemento minimo presente nell'array tra le posizioni `start` ed `end` (inclusi).

Spiegazione dell'algoritmo: Inizializziamo l'indice del minimo con il valore di partenza start. Scansioniamo l'array con un ciclo da start + 1 a end. Se troviamo un valore inferiore a quello puntato da idx\{\}_min, aggiorniamo idx\{\}_min. Al termine restituiamo l'indice.

```

1  #include <stdio.h>
2
3  int trova_minimo(int arr[], int start, int end) {
4      int idx_min = start;
5      for (int i = start + 1; i <= end; i++) {
6          if (arr[i] < arr[idx_min]) {
7              idx_min = i; // Aggiorna l'indice del minimo
8          }
9      }
10     return idx_min;
11 }
12
13 int main(void) {
14     int vet[] = {9, 4, 2, 7, 5, 1, 8};
15     // Cerca il minimo tra l'indice 1 e l'indice 5 (valori da 4 a
16     int pos = trova_minimo(vet, 1, 5);
17     printf("Il minimo tra gli indici 1 e 5 si trova all'indice %d (
18     valore: %d)\n", pos, vet[pos]);
19     return 0;

```

Soluzione

Esercizio 2: Bubble Sort Decrescente

Domanda

Modificate l'algoritmo del Bubble Sort per ordinare l'array in ordine decrescente (dal più grande al più piccolo).

Spiegazione dell'algoritmo: Per invertire l'ordine dell'ordinamento è sufficiente invertire la condizione del confronto all'interno del ciclo interno: modificando `arr[j] < arr[j + 1]` in `arr[j] > arr[j + 1]`, facciamo risalire a destra i valori più piccoli, spingendo a sinistra i valori più grandi.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  void swap(int *a, int *b) {
5      int temp = *a;
6      *a = *b;
7      *b = temp;
8  }
9
10 void bubble_sort_descendente(int arr[], int n) {
11     bool scambiato;
12     for (int i = 0; i < n - 1; i++) {
13         scambiato = false;
14         for (int j = 0; j < n - i - 1; j++) {
15             // Per ordinare in modo decrescente, scambiamo se l'
16             // elemento di sinistra è MINORE di quello a destra
17             if (arr[j] < arr[j + 1]) {
18                 swap(&arr[j], &arr[j + 1]);
19                 scambiato = true;
20             }
21             if (!scambiato) break;
22         }
23     }
24
25     int vett[] = {3, 1, 4, 5, 2};
26     bubble_sort_descendente(vett, 5);
27     printf("Decrescente: ");
28     for (int i = 0; i < 5; i++) {
29         printf("%d ", vett[i]);
30     }
31     printf("\n");
32     return 0;
33 }
34

```

Soluzione

Esercizio 3: Conta Confronti e Scambi nel Selection Sort

Domanda

Implementate una versione del Selection Sort che tenga traccia del numero totale di confronti effettuati tra elementi e del numero di swap eseguiti, stampando i risultati alla fine.

Spiegazione dell'algoritmo: Incrementiamo la variabile confronti ad ogni esecuzione della condizione `if` del ciclo interno e incrementiamo scambi solo all'interno del blocco condizionale `if (min_idx != i)` che esegue lo swap. Su 5 elementi disordinati vedremo 10 confronti e al massimo 4 swap.

```

1  #include <stdio.h>
2
3  void selection_sort_tracciato(int arr[], int n) {
4      int confronti = 0;
5      int scambi = 0;
6
7      for (int i = 0; i < n - 1; i++) {
8          int min_idx = i;
9          for (int j = i + 1; j < n; j++) {
10             confronti++;
11             if (arr[j] < arr[min_idx]) {
12                 min_idx = j;
13             }
14         }
15         if (min_idx != i) {
16             int temp = arr[i];
17             arr[i] = arr[min_idx];
18             arr[min_idx] = temp;
19             scambi++;
20         }
21         printf("Confronti eseguiti: %d\n", confronti);
22         printf("Swap eseguiti: %d\n", scambi);
23     }
24 }
25
26 int main(void) {
27     int vet[] = {4, 2, 5, 1, 3};
28     selection_sort_tracciato(vet, 5);
29     return 0;
30 }

```

Soluzione

Esercizio 4: Bubble Sort Bidirezionale (Cocktail Shaker)

Domanda

Sort)

Scrivete una variante del Bubble Sort che esegua passate alternate: una da sinistra verso destra (portando il massimo a destra) e una da destra verso sinistra (portando il minimo a sinistra).

--

Spiegazione dell'algoritmo: Chiamato anche *Cocktail Shaker Sort*, questo approccio ottimizza il Bubble Sort riducendo il problema degli elementi piccoli situati in fondo a destra (chiamati "tartarughe"), che impiegherebbero molto tempo a risalire a sinistra in passate unidirezionali.

```

7 #include <csdlib.h>
8
9
10 void cocktail_sort(int arr[], int n) {
11     bool scambiato = true;
12     int start = 0;
13     int end = n - 1;
14     while (scambiato) {
15         scambiato = false;
16         // Passata da sinistra a destra (Bubble Sort classico)
17         for (int i = start; i < end; i++) {
18             if (arr[i] > arr[i + 1]) {
19                 swap(&arr[i], &arr[i + 1]);
20                 scambiato = true;
21             }
22         }
23         // Se non ci sono stati scambi, l'array è ordinato
24         if (!scambiato) break;
25         // Riduciamo end perché l'elemento massimo è congelato a
26         // destra
27         end--;
28         scambiato = false;
29         // Passata da destra a sinistra
30         for (int i = end - 1; i >= start; i--) {
31             if (arr[i] > arr[i + 1]) {
32                 swap(&arr[i], &arr[i + 1]);
33                 scambiato = true;
34             }
35         }
36         // Incrementiamo start perché il minimo è congelato a
37         // sinistra
38         start++;
39     }
40 }
41
42 int main(void) {
43     int vet[] = {5, 1, 4, 2, 3};
44     cocktail_sort(vet, 5);
45     printf("Ordinato: ");
46     for (int i = 0; i < 5; i++) {
47         printf("%d ", vet[i]);
48     }
49     printf("\n");
50     return 0;
51 }

```

Esercizio 5: Ricerca Lineare di Tutte le Occorrenze

Domanda

Scrivete una funzione `int cerca_tutte(int arr[], int n, int chiave, int indici[])` che cerchi una chiave in un array disordinato e salvi tutti gli indici in cui compare all'interno dell'array `indici`. La funzione deve restituire il numero totale di occorrenze trovate.

Spiegazione dell'algoritmo: Scorriamo l'array cella per cella. Quando `arr[i]` è uguale alla chiave, salviamo il valore dell'indice `i` nella posizione `contatore` dell'array di output `indici` e incrementiamo `contatore`.

```

1  #include <stdio.h>
2
3  int cerca_tutte(int arr[], int n, int chiave, int indici[]) {
4      int contatore = 0;
5      for (int i = 0; i < n; i++) {
6          if (arr[i] == chiave) {
7              indici[contatore] = i; // Salva l'indice corrente
8              contatore++;
9          }
10         return contatore; // Restituisce il numero totale
11     }
12 }
13
14 int main(void) {
15     int vett[] = {4, 2, 5, 2, 3, 2, 8};
16     int ris[7]; // Vettore per salvare gli indici trovati
17     int occorrenze = cerca_tutte(vett, 7, 2, ris);
18     printf("Trovate %d occorrenze del valore 2 agli indici: ",
19         occorrenze);
20     for (int i = 0; i < occorrenze; i++) {
21         printf("%d ", ris[i]);
22     }
23     printf("\n");
24     return 0;
25 }

```

Soluzione

Esercizio 6: Inserimento Ordinato

Domanda

Scrivete una funzione `void inserimento_ordinato(int arr[], int *n, int max_size, int valore)` che inserisca un nuovo valore all'interno di un array già ordinato di dimensione logica `*n`, spostando gli elementi necessari per preservare l'ordinamento. La dimensione logica dell'array deve essere incrementata tramite il puntatore.

Spiegazione dell'algoritmo: Questo algoritmo è una singola passata del ciclo interno dell'insertion Sort. Partiamo dalla fine dell'array ($*n - 1$) e facciamo slittare a destra gli elementi maggiori del nuovo valore, inserendolo poi nello spazio vuoto.

```

1  #include <stdio.h>
2
3  void inserimento_ordinato(int arr[], int *n, int max_size, int
4      ↪ valore) {
5      // Controlla se c'è spazio fisico nell'array
6      if (*n >= max_size) {
7          printf("Errore: array pieno!\n");
8          return;
9      }
10     int j = *n - 1;
11     // Slitta gli elementi a destra finché sono maggiori del valore
12     ↪ da inserire
13     while (j >= 0 && arr[j] > valore) {
14         arr[j + 1] = arr[j];
15         j--;
16     }
17     // Inserisce il valore nella posizione liberata
18     arr[j + 1] = valore;
19     // Incrementa la dimensione logica puntata
20     (*n)++;
21 }
22
23 int main(void) {
24     int vet[10] = {2, 4, 7, 8, 12};
25     int n = 5; // Dimensione logica iniziale
26
27     inserimento_ordinato(vet, &n, 10, 5);
28
29     printf("Array dopo inserimento ordinato di 5: ");
30     for (int i = 0; i < n; i++) {
31         printf("%d ", vet[i]);
32     }
33     printf("\n");
34     return 0;
35 }

```

Soluzione

Esercizio 7: Ricerca Dicotomica Ricorsiva

Domanda

Riscrivete l'algoritmo della ricerca dicotomica in modalità ricorsiva sfruttando i parametri di frontiera low e high.

Spiegazione dell'algoritmo: Se $low > high$, l'elemento non è presente (caso base di fallimento). Altrimenti calcoliamo mid e confrontiamo. Se $arr[mid]$ è maggiore della chiave, richiamiamo ricorsivamente la funzione restringendo l'intervallo a $[low, mid - 1]$, altrimenti lo restringiamo a $[mid + 1, high]$.

```

1  #include <stdio.h>
2
3  int ricerca_dicotomica_ricorsiva(int arr[], int low, int high, int
4      // Caso base: area di ricerca esaurita
5      if (low > high) {
6          return -1;
7      }
8
9      int mid = low + (high - low) / 2;
10
11     if (arr[mid] == chiave) {
12         return mid; // Trovato!
13     }
14
15     if (arr[mid] > chiave) {
16         // Cerca nella metà sinistra
17         return ricerca_dicotomica_ricorsiva(arr, low, mid - 1,
18             chiave);
19     } else {
20         // Cerca nella metà destra
21         return ricerca_dicotomica_ricorsiva(arr, mid + 1, high,
22             chiave);
23     }
24 }
25
26 int elenco[] = {2, 4, 5, 8, 12, 15, 18, 21, 24, 27};
27 int pos = ricerca_dicotomica_ricorsiva(elenco, 0, 9, 24);
28 printf("Indice trovato ricorsivamente: %d\n", pos);
29 return 0;

```

Soluzione

Esercizio 8: Selection Sort su Array di Float

Domanda

Adattate la funzione del Selection Sort per ordinare un array di numeri reali in virgola mobile (float).

Spiegazione dell'algoritmo: L'algoritmo rimane strutturalmente identico. È stato sufficiente modificare la dichiarazione del tipo del parametro formale dell'array (float arr[]) e della variabile temporanea all'interno dello swap (float temp).

```

1  #include <stdio.h>
2
3  void selection_sort_float(float arr[], int n) {
4      for (int i = 0; i < n - 1; i++) {
5          int min_idx = i;
6          for (int j = i + 1; j < n; j++) {
7              if (arr[j] < arr[min_idx]) {
8                  min_idx = j;
9              }
10             if (min_idx != i) {
11                 float temp = arr[i];
12                 arr[i] = arr[min_idx];
13                 arr[min_idx] = temp;
14             }
15         }
16     }
17 }
18
19 int main(void) {
20     float vet[] = {3.14, 1.41, 2.71, 0.707, 1.618};
21     selection_sort_float(vet, 5);
22
23     printf("Float ordinati: ");
24     for (int i = 0; i < 5; i++) {
25         printf("%.3f ", vet[i]);
26     }
27     printf("\n");
28     return 0;
29 }

```

Soluzione

Esercizio 9: Rimozione dei Duplicati in un Array Ordinato

Domanda

Scrivete una funzione `int rimuovi\_duplicati(int arr[], int n)` che prenda un array già ordinato e rimuova gli elementi duplicati in situ (senza vettori ausiliari). La funzione deve restituire la nuova dimensione logica dell'array.

Spiegazione dell'algoritmo: Sfruttiamo l'ordinamento: gli elementi uguali sono adiacenti. Scorriamo il vettore confrontando `arr[i]` con `arr[i+1]`. Se sono diversi, spostiamo `arr[i]` all'indice `j` (che tiene traccia della porzione senza duplicati) e avanziamo `j`.

```

1  #include <stdio.h>
2
3  int rimuovi_duplicati(int arr[], int n) {
4      if (n == 0 || n == 1) return n;
5
6      int j = 0; // Indice per memorizzare gli elementi unici
7      for (int i = 0; i < n - 1; i++) {
8          // Se l'elemento corrente è diverso dal successivo, lo
9          // salviamo
10         if (arr[i] != arr[i + 1]) {
11             arr[j] = arr[i];
12             j++;
13         }
14         // Salva l'ultimo elemento che non è stato confrontato nel
15         // ciclo
16         arr[j] = arr[n - 1];
17         j++;
18     }
19     return j; // Nuova dimensione logica
20 }
21
22 int main(void) {
23     int dati[] = {1, 1, 2, 2, 3, 4, 4, 5};
24     int n = 8;
25     int nuova_dim = rimuovi_duplicati(dati, n);
26     printf("Array senza duplicati (%d elementi): ", nuova_dim);
27     for (int i = 0; i < nuova_dim; i++) {
28         printf("%d ", dati[i]);
29     }
30     printf("\n");
31     return 0;
32 }

```

Soluzione

Esercizio 10: Ricerca Dicotomica con Verifica Indici

Domanda

Limite

Scrivete una variante di `ricerca_dicotomica` che, se la chiave è presente più volte, garantisca di restituire l'indice della **prima occorrenza** (quella più a sinistra) all'interno dell'array.

Soluzione

```

1  #include <stdio.h>
2
3  int prima-occorrenza-dicotomica(int arr[], int n, int chiave) {
4      int low = 0;
5      int high = n - 1;
6      int ris = -1; // Memorizza la posizione temporanea
7
8      while (low <= high) {
9          int mid = low + (high - low) / 2;
10
11         if (arr[mid] == chiave) {
12             ris = mid; // Trovata! Memorizziamo l'indice
13             high = mid - 1; // Continuiamo a cercare a SINISTRA per
14             // trovare occorrenze precedenti
15         } else if (arr[mid] < chiave) {
16             low = mid + 1;
17         } else {
18             high = mid - 1;
19         }
20     }
21     return ris;
22 }
23
24 int main(void) {
25     // Array ordinato con duplicati del valore 12
26     int elenco[] = {2, 4, 12, 12, 12, 15, 18, 21};
27     int pos = prima-occorrenza-dicotomica(elenco, 8, 12);
28     printf("La prima occorrenza di 12 si trova all'indice: %d\n",
29         pos); // Deve stampare 2
30     return 0;
31 }

```

Spiegazione dell'algoritmo: Quando troviamo `arr[mid] == chiave`, anziché fermarci restituendo subito `mid`, salviamo questo indice in una variabile di appoggio `ris` e forziamo la ricerca a procedere nella metà sinistra restringendo il range di ricerca con `high = mid - 1`. Se non ci sono altre occorrenze, la ricerca terminerà restituendo l'ultima salvata in `ris`.