

Algoritmi di ordinamento e O grande

Algoritmi di ordinamento

- Abbiamo già visto come ordinare un array di dati in ordine crescente o decrescente con Bubble Sort
- Esistono algoritmi più efficienti?
- La notazione «O grande» permette di stimare il tempo di esecuzione di un algoritmo nel caso peggiore
- Vedremo e compareremo diversi algoritmi di ordinamento e li compareremo per
 - Tempo di esecuzione
 - memoria

O grande

- La notazione O grande descrive lo sforzo richiesto a un algoritmo
- Indica quanta mole di lavoro un algoritmo deve svolgere per risolvere un problema
- Per gli algoritmi di ricerca e ordinamento, dipende principalmente dal numero di elementi nei dati
- La notazione O grande è usata per descrivere i tempi di esecuzione nel caso peggiore degli algoritmi di ordinamento

Algoritmi O(1)

- Considera un algoritmo che controlla se il primo elemento di un array è uguale al secondo:
 - Se l'array ha 10 elementi, richiede un confronto
 - Se l'array ha 1000 elementi, richiede ancora un confronto
- Il numero di operazioni dell'algoritmo è indipendente dal numero di elementi dell'array
- Questo tipo di algoritmi ha un tempo di esecuzione **costante**: O(1)
- O(1) non implica solo un confronto, ma un numero costante di operazioni
- Un algoritmo che confronta il primo elemento con i tre successivi, richiede ancora tempo O(1) anche se richiede tre confronti

Algoritmi $O(n)$

- Un algoritmo che controlla se il primo elemento di un array è uguale a un qualsiasi altro elemento richiede al massimo $n-1$ confronti
 - Se l'array ha 10 elementi, richiede fino a 9 confronti
 - Se l'array ha 1.000 elementi, richiede fino a 999 confronti
- Al crescere di n , la parte dominante in $n-1$ sarà n
 - Sottrarre 1 diventa trascurabile
- La notazione O grande sottolinea i termini dominanti e ignora quelli trascurabili
- Un algoritmo che richiede $n-1$ confronti viene detto $O(n)$
 - Un algoritmo $O(n)$ ha un tempo di esecuzione **lineare**

Algoritmi $O(n^2)$

- Un algoritmo che controlla se tutti gli elementi di un array sono duplicati
 - Confronta il primo elemento con tutti gli altri
 - Poi il secondo con i restanti, e così via
- Il numero di confronti è $(n - 1) + (n - 2) + \dots + 2 + 1$, ovvero $\left(\frac{n^2}{2} - \frac{n}{2}\right)$
- La notazione O grande evidenzia il termine dominante n^2 e ignora i fattori costanti
 - Il tempo di esecuzione di questo algoritmo è $O(n^2)$
 - Il tempo di esecuzione $O(n^2)$ è **quadratico**

Algoritmi $O(n^2)$

- La relazione tra elementi da elaborare e tempo di esecuzione è
 - Algoritmi con n^2 confronti
 - 4 elementi, 16 confronti
 - 8 elementi, 64 confronti
 - Algoritmi con $n^2 / 2$ confronti
 - 4 elementi, 8 confronti
 - 8 elementi, 32 confronti
- In entrambi i casi raddoppiare i dati quadruplica i confronti

Algoritmi $O(n^2)$

- Stima delle prestazioni di un algoritmo $O(n^2)$
 - Su un milione di elementi: mille miliardi di operazioni (richiede ore)
 - Su un miliardo di elementi: un miliardo di miliardi di operazioni (richiede decenni)
- Gli algoritmi $O(n^2)$ sono facili da scrivere ma inefficienti per n grande
- Algoritmi più efficienti richiedono più lavoro e tecniche più raffinati, ma offrono prestazioni molto migliori

Ordinamento per selezione

- L'ordinamento per selezione (*selection sort*) è un algoritmo semplice, ma inefficiente
 - La prima iterazione seleziona l'elemento più piccolo nell'array e lo scambia con il primo elemento
 - La seconda iterazione seleziona il secondo elemento più piccolo (il più piccolo dei restanti) e lo scambia con il secondo elemento
 - Continua fino all'ultima iterazione che seleziona il secondo elemento più grande e lo scambia con il penultimo, lasciando ultimo l'elemento più grande

Ordinamento per selezione

- Consideriamo l'array

34 56 4 10 77 51 93 30 5 52

- L'ordinamento per selezione determina l'elemento più piccolo (4), poi lo scambia con il valore del primo elemento (34)

4 56 34 10 77 51 93 30 5 52

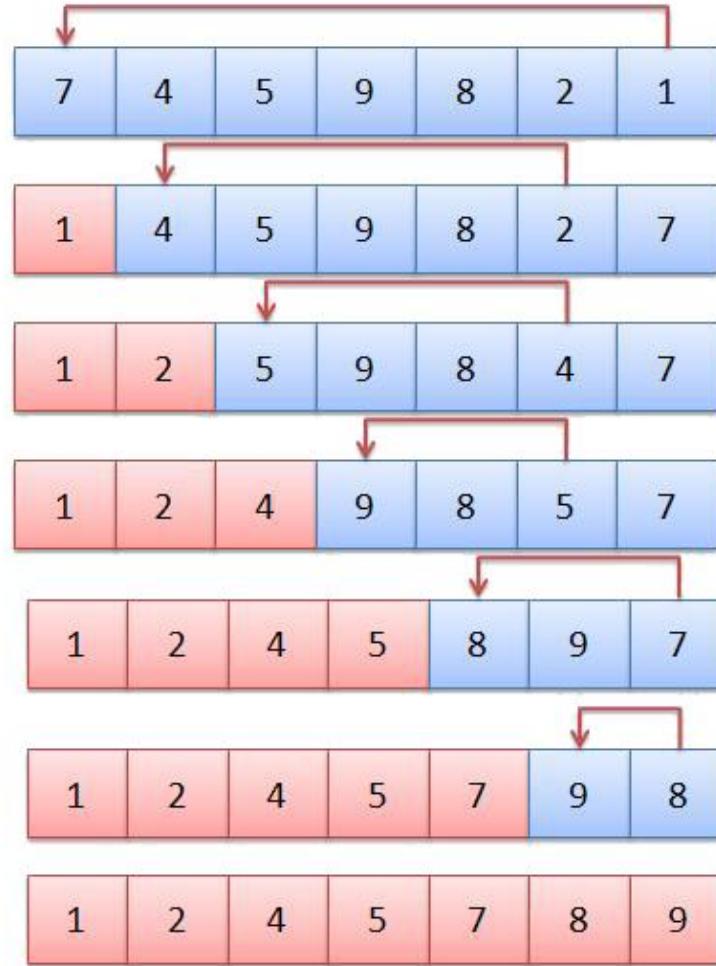
- Determina il valore dell'elemento rimanente più piccolo partendo dal secondo elemento (5) e lo scambia con il valore 56 del secondo elemento

4 **5** 34 10 77 51 93 30 56 52

- Quando l'algoritmo termina, l'array è ordinato

4 5 10 30 34 51 52 56 77 93

Ordinamento per selezione



Implementare ordinamento per selezione

```
1 // fig13_01.c
2 // Algoritmo di ordinamento per selezione.
3 #define SIZE 10
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 void selectionSort(int array[], size_t length);
9 void swap(int array[], size_t first, size_t second);
10 void printPass(int array[], size_t length, int pass, size_t index);
11
12 int main(void) {
13     int array[SIZE] = {0}; // dichiara l'array di int da ordinare
14
15     srand(time(NULL)); // fornisce il seme alla funzione rand
16
17     for (size_t i = 0; i < SIZE; i++) {
18         array[i] = rand() % 90 + 10; // assegna un valore a ogni elemento
19     }
20
21     puts("Unsorted array:");
22
23     for (size_t i = 0; i < SIZE; i++) { // stampa l'array
24         printf("%d ", array[i]);
25     }
26
27     puts("\n");
28     selectionSort(array, SIZE);
29     puts("Sorted array:");
30
31     for (size_t i = 0; i < SIZE; i++) { // stampa l'array
32         printf("%d ", array[i]);
33     }
34
35     puts("");
36 }
```

Implementare ordinamento per selezione

```
29 // funzione che ordina per selezione l'array
30 void selectionSort(int array[], size_t length) {
31     // esegui l'iterazione su length - 1 elementi
32     for (size_t i = 0; i < length - 1; i++) {
33         size_t smallest = i; // primo indice dell'array rimanente
33
34         // trova l'indice dell'elemento piu' piccolo
35         for (size_t j = i + 1; j < length; j++) {
36             if (array[j] < array[smallest]) {
37                 smallest = j;
38             }
39         }
39
40         swap(array, i, smallest); // scambia con l'elemento piu' piccolo
41         printPass(array, length, i + 1, smallest); // stampa il passo
42     }
43 }
```

Implementare ordinamento per selezione

```
44 // funzione che scambia due elementi nell'array
45 void swap(int array[], size_t first, size_t second) {
46     int temp = array[first];
47     array[first] = array[second];
48     array[second] = temp;
49 }
```

Implementare ordinamento per selezione

```
50 // funzione che stampa un passo dell'algoritmo
51 void printPass(int array[], size_t length, int pass, size_t index) {
52     printf("After pass %2d: ", pass);
52
53     // stampa gli elementi fino all'elemento selezionato
54     for (size_t i = 0; i < index; i++) {
55         printf("%d ", array[i]);
56     }
56
57     printf("%d* ", array[index]); // indica l'elemento scambiato
57
58     // completa la stampa dell'array
59     for (size_t i = index + 1; i < length; i++) {
60         printf("%d ", array[i]);
61     }
61
62     printf("%s", "\n"); // per l'allineamento
62
63     // indica la porzione dell'array che e' ordinata
64     for (int i = 0; i < pass; i++) {
65         printf("%s", "-- ");
66     }
66
67     puts(""); // aggiungi un newline
68 }
```

Implementare ordinamento per selezione

Unsorted array:

72	34	88	14	32	12	34	77	56	83	
After pass 1:	12	34	88	14	32	72*	34	77	56	83
	--									
After pass 2:	12	14	88	34*	32	72	34	77	56	83
	--	--								
After pass 3:	12	14	32	34	88*	72	34	77	56	83
	--	--	--							
After pass 4:	12	14	32	34*	88	72	34	77	56	83
	--	--	--							
After pass 5:	12	14	32	34	34	72	88*	77	56	83
	--	--	--	--	--					
After pass 6:	12	14	32	34	34	56	88	77	72*	83
	--	--	--	--	--					
After pass 7:	12	14	32	34	34	56	72	77	88*	83
	--	--	--	--	--					
After pass 8:	12	14	32	34	34	56	72	77*	88	83
	--	--	--	--	--					
After pass 9:	12	14	32	34	34	56	72	77	83	88*
	--	--	--	--	--					
After pass 10:	12	14	32	34	34	56	72	77	83	88*
	--	--	--	--	--					

Sorted array:

12	14	32	34	34	56	72	77	83	88
----	----	----	----	----	----	----	----	----	----

Efficienza ordinamento per selezione

- L'algoritmo di ordinamento per selezione
 - Si esegue in un tempo $O(n^2)$
- Funzione selectionSort
 - Il ciclo esterno elabora i primi $n - 1$ elementi
 - Inserisce l'elemento più piccolo rimanente nella sua posizione ordinata
 - Il ciclo interno cerca l'elemento più piccolo tra quelli rimanenti
 - Viene eseguito $n - 1$ volte nella prima iterazione del ciclo esterno
 - $n - 2$ volte nella seconda iterazione
 - Così via fino all'ultima iterazione
 - Il numero totale di iterazioni del ciclo interno sono $\frac{n(n-1)}{2} = (n^2-n)/2$
- Nella notazione O grande i termini più piccoli e le costante vengono ignorate, ottenendo un tempo di esecuzione $O(n^2)$

Ordinamento per inserzione

- L'ordinamento per inserzione (*insertion sort*) è un altro algoritmo di ordinamento semplice ma inefficiente
 - La prima interazione prende il secondo elemento dell'array e, se è minore del primo elemento, allora li scambia
 - La seconda iterazione considera il terzo elemento e lo inserisce nella posizione corretta rispetto ai primi due elementi
 - Alla i -esima iterazione di questo algoritmo, vengono ordinati i primi i elementi dell'array originario

Ordinamento per inserzione

- Considera il seguente array:

34 56 4 10 77 51 93 30 5 52

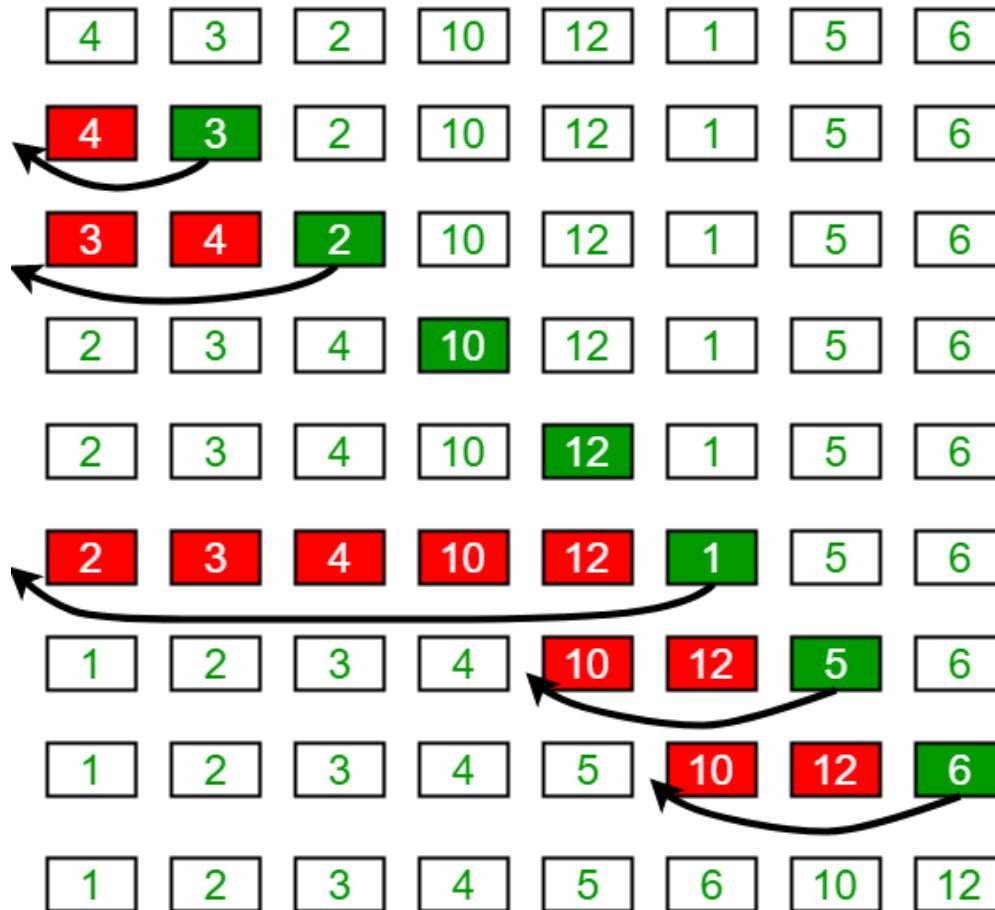
- Prima iterazione: confronta 34 e 56 (già ordinati)
- Seconda iterazione: considera il terzo elemento (4), sposta 34 e 56 a destra, inserisce 4

4 34 56 10 77 51 93 30 5 52

- Terza iterazione: memorizza 10 in una variabile temporanea, confronta 10 con 56 e sposta 56 a destra di una posizione, confronta 10 con 34 e sposta 34 a destra, confronta 10 con 4 e mette 10 in seconda posizione
- Alla i -esima iterazione, i primi $i+1$ elementi sono ordinati tra loro ma potrebbero non essere nella loro posizione finale
 - ci potrebbero essere elementi più piccoli negli elementi seguenti dell'array

Ordinamento per inserzione

Insertion Sort Execution Example



Implementare ordinamento per inserzione

```
1  / fig13_02.c
2  // Algoritmo di ordinamento per inserzione.
3  #define SIZE 10
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  // prototipi di funzioni
9  void insertionSort(int array[], size_t length);
10 void printPass(int array[], size_t length, int pass, size_t index);
11
12 int main(void)
13     int array[SIZE] = {0}; // dichiara l'array di int da ordinare
14
15     srand(time(NULL)); // fornisci il seme alla funzione rand
16
17     for (size_t i = 0; i < SIZE; i++) {
18         array[i] = rand() % 90 + 10; // assegna un valore a ogni elemento
19     }
20
21     puts("Unsorted array:");
22
23     for (size_t i = 0; i < SIZE; i++) { // stampa l'array
24         printf("%d ", array[i]);
25     }
26
27     puts("\n");
28     insertionSort(array, SIZE);
29     puts("Sorted array:");
30
31     for (size_t i = 0; i < SIZE; i++) { // stampa l'array
32         printf("%d ", array[i]);
33     }
34
35     puts("");
36 }
```

Implementare ordinamento per inserzione

```
30 // funzione che ordina l'array
31 void insertionSort(int array[], size_t length) {
32     // esegui l'iterazione su length - 1 elementi
33     for (size_t i = 1; i < length; i++) {
34         size_t moveItem = i; // posizione in cui inserire l'elemento
35         int insert = array[i]; // contiene l'elemento da inserire
36
37         // cerca la posizione dove mettere l'elemento corrente
38         while (moveItem > 0 && array[moveItem - 1] > insert) {
39             // sposta l'elemento a destra di una posizione
40             array[moveItem] = array[moveItem - 1];
41             --moveItem;
42         }
43
44         array[moveItem] = insert; // inserisci l'elemento al suo posto
45         printPass(array, length, i, moveItem);
46     }
}
```

Implementare ordinamento per inserzione

```
47 // funzione che stampa un passo dell'algoritmo
48 void printPass(int array[], size_t length, int pass, size_t index) {
49     printf("After pass %2d: ", pass);
50
51     // stampa gli elementi fino all'elemento selezionato
52     for (size_t i = 0; i < index; i++) {
53         printf("%d ", array[i]);
54     }
55
56     printf("%d* ", array[index]); // indica l'elemento inserito
57
58     // termina di stampare l'array
59     for (size_t i = index + 1; i < length; i++) {
60         printf("%d ", array[i]);
61     }
62
63     printf("%s", "\n"); // per l'allineamento
64
65     // indica la porzione di array che e' ordinata
66     for (size_t i = 0; i <= pass; i++) {
67         printf("%s", "-- ");
68     }
69
70     puts(""); // aggiungi un newline
71 }
```

Implementare ordinamento per inserzione

Unsorted array:

72 16 11 92 63 99 59 82 99 30

After pass 1: 16* 72 11 92 63 99 59 82 99 30

 -- --

After pass 2: 11* 16 72 92 63 99 59 82 99 30

 -- -- --

After pass 3: 11 16 72 92* 63 99 59 82 99 30

 -- -- -- --

After pass 4: 11 16 63* 72 92 99 59 82 99 30

 -- -- -- -- --

After pass 5: 11 16 63 72 92 99* 59 82 99 30

 -- -- -- -- -- --

After pass 6: 11 16 59* 63 72 92 99 82 99 30

 -- -- -- -- -- -- --

After pass 7: 11 16 59 63 72 82* 92 99 99 30

 -- -- -- -- -- -- -- --

After pass 8: 11 16 59 63 72 82 92 99 99* 30

 -- -- -- -- -- -- -- -- --

After pass 9: 11 16 30* 59 63 72 82 92 99 99

 -- -- -- -- -- -- -- -- -- --

Sorted array:

11 16 30 59 63 72 82 92 99 99

Efficienza ordinamento per inserzione

- L'algoritmo di ordinamento per selezione
 - Si esegue in un tempo $O(n^2)$
- Funzione *insertionSort*
 - Il ciclo esterno effettua l'iterazione SIZE-1($=n-1$) volte
 - Inserisce un elemento nella posizione corretta tra quelli ordinati fino a quel punto
 - Il ciclo interno effettua l'iterazione sugli elementi già ordinati dell'array
 - Nel caso peggiore richiede $n - 1$ confronti
 - Ogni ciclo richiede tempo $O(n)$
 - Il numero totale di iterazioni del ciclo interno sono $O(n^2)$ perché il ciclo è eseguito $O(n)$ volte
- Nella notazione O grande i termini più piccoli e le costante vengono ignorate, ottenendo un tempo di esecuzione $O(n^2)$

Intervallo



Fonte: chatGPT



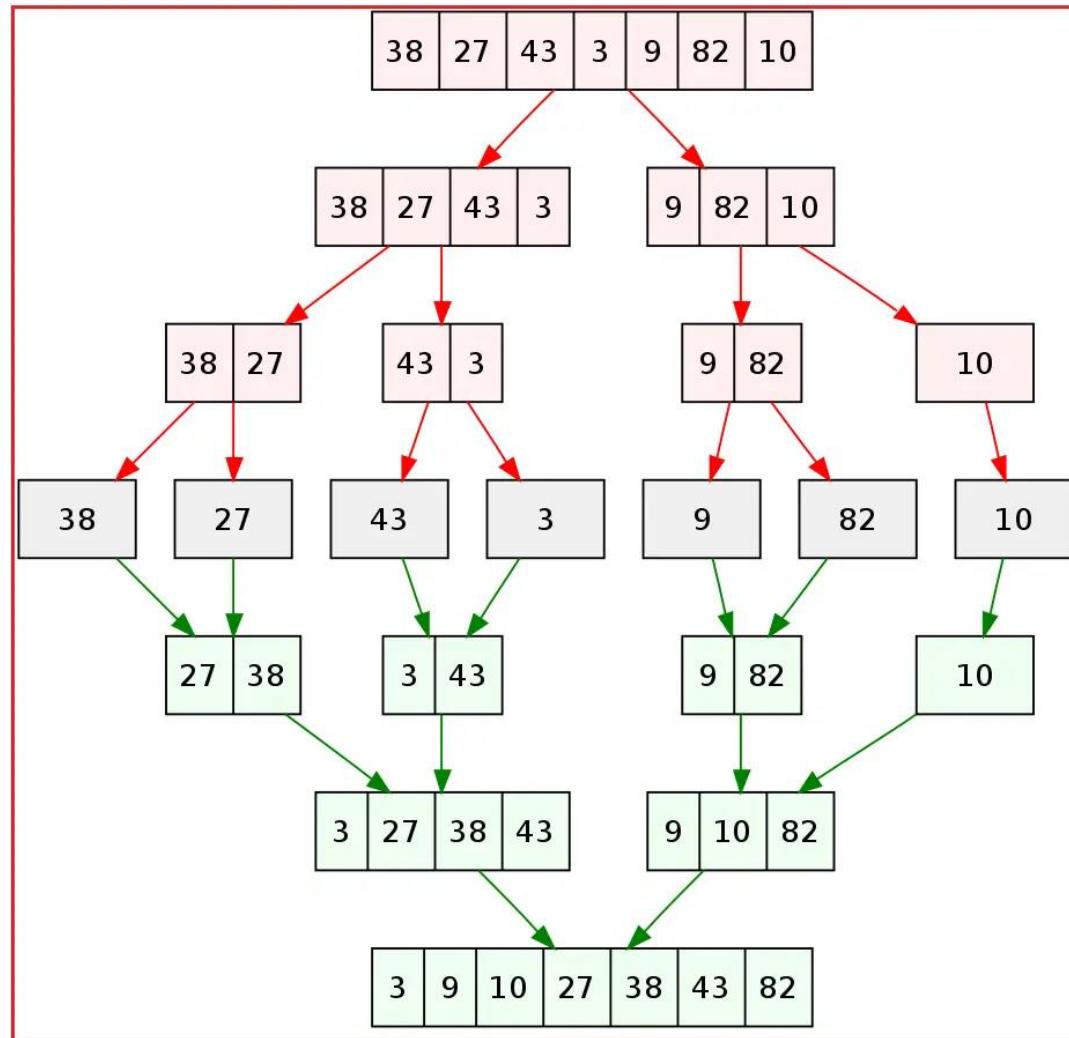
Ordinamento per fusione

- L'ordinamento per fusione (*merge sort*) è un algoritmo di ordinamento efficiente ma complesso
- L'algoritmo ordina un array
 - suddividendolo in due sottoarray della stessa dimensione
 - Ordina ognuno dei sottoarray
 - Li fonde insieme in un array più grande
- L'implementazione presentata è ricorsiva
 - Il caso base è dato da un array con un solo elemento (sempre ordinato)
 - Il passo di ricorsione suddivide un array di due o più elementi in due array, li ordina ricorsivamente, e li fonde

Ordinamento per fusione

- Supponiamo che l'algoritmo abbia già creato gli array ordinati
- A: 4 10 34 56 77
- B: 5 30 51 52 93
- Gli array vengono ordinati in un unico array
- Per determinare l'elemento più piccolo vengono confrontati gli elementi più piccoli di A (4) e B (5)
- 4 è il primo elemento dell'array
- L'algoritmo confronta l'elemento più piccolo rimanente in A (10) e B (5)
- 5 è il secondo elemento dell'array
- Si continua così fino ad esaurire tutti gli elementi di A e B

Ordinamento per fusione



Implementare ordinamento per fusione

```
1 // fig13_03.c
2 // Algoritmo di ordinamento per fusione.
3 #define SIZE 10
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 void mergeSort(int array[], size_t length);
9 void sortSubArray(int array[], size_t low, size_t high);
10 void merge(int array[], size_t left, size_t middle1,
11           size_t middle2, size_t right);
12 void displayElements(int array[], size_t length);
13 void displaySubArray(int array[], size_t left, size_t right);
14
15 int main(void) {
16     int array[SIZE] = {0}; // dichiara l'array di int da ordinare
17
18     srand(time(NULL)); // fornisce il seme alla funzione rand
19
20     for (size_t i = 0; i < SIZE; i++) {
21         array[i] = rand() % 90 + 10; // assegna un valore a ogni elemento
22     }
23
24     puts("Unsorted array:");
25     displayElements(array, SIZE); // stampa l'array
26     puts("\n");
27     mergeSort(array, SIZE); // ordina per fusione l'array
28     puts("Sorted array:");
29     displayElements(array, SIZE); // stampa l'array
30     puts("");
31 }
```

Implementare ordinamento per fusione

```
31 // funzione che ordina per fusione l'array
32 void mergeSort(int array[], size_t length) {
33     sortSubArray(array, 0, length - 1);
34 }
34
35 // funzione che ordina una porzione dell'array
36 void sortSubArray(int array[], size_t low, size_t high) {
37     // effettua il test per il caso di base: la dimensione dell'array è 1
38     if ((high - low) >= 1) { // se non si tratta del caso di base...
39         size_t middle1 = (low + high) / 2;
40         size_t middle2 = middle1 + 1;
41
42         // stampa il passo di suddivisione
43         printf("%s", "split:  ");
44         displaySubArray(array, low, high);
45         printf("\n        ");
46         displaySubArray(array, low, middle1);
47         printf("\n        ");
48         displaySubArray(array, middle2, high);
49         puts("\n");
50
51         // dividi l'array a metà e ordina ciascuna metà ricorsivamente
52         sortSubArray(array, low, middle1); // prima metà
53         sortSubArray(array, middle2, high); // seconda metà
54
55         // fondi i due array ordinati
56         merge(array, low, middle1, middle2, high);
57     }
58 }
```

Implementare ordinamento per fusione

```
59 // fonda i due sottoarray ordinati in un sottoarray ordinato
60 void merge(int array[], size_t left, size_t middle1,
61     size_t middle2, size_t right) {
62     size_t leftIndex = left; // indice nel sottoarray sinistro
63     size_t rightIndex = middle2; // indice nel sottoarray destro
64     size_t combinedIndex = left; // indice nell'array temporaneo
65     int tempArray[SIZE] = {0}; // array temporaneo
66
67     // stampa i due sotrray prima di fonderli
68     printf("%s", "merge:  ");
69     displaySubArray(array, left, middle1);
70     printf("\n        ");
71     displaySubArray(array, middle2, right);
72     puts("");
73
74     // fonda i sotrray finche' non si raggiunge la fine di uno di loro
75     while (leftIndex <= middle1 && rightIndex <= right) {
76         // inserisci il piu' piccolo dei due elementi correnti nel risultato
77         // e spostati nella posizione seguente nel sottoarray
78         if (array[leftIndex] <= array[rightIndex]) {
79             tempArray[combinedIndex++] = array[leftIndex++];
80         }
81         else {
82             tempArray[combinedIndex++] = array[rightIndex++];
83         }
84     }
85
86     if (leftIndex == middle2) { // fine del sottoarray sinistro?
87         while (rightIndex <= right) { // copia il sottoarray destro
88             tempArray[combinedIndex++] = array[rightIndex++];
89         }
90     }
91     else { // fine del sottoarray destro?
92         while (leftIndex <= middle1) { // copia il sottoarray sinistro
93             tempArray[combinedIndex++] = array[leftIndex++];
94         }
95     }
}
```

Implementare ordinamento per fusione

```
96
97     // copia di nuovo i valori nell'array originario
98     for (size_t i = left; i <= right; i++) {
99         array[i] = tempArray[i];
100    }
101
102    // stampa il sottoarray combinato
103    printf("%s", "           ");
104    displaySubArray(array, left, right);
105    puts("\n");
106 }
```

Implementare ordinamento per fusione

```
Unsorted array:  
 79 86 60 79 76 71 44 88 58 23  
split:   79 86 60 79 76 71 44 88 58 23  
          79 86 60 79 76  
                  71 44 88 58 23  
split:   79 86 60 79 76  
          79 86 60  
                  79 76  
split:   79 86 60  
          79 86  
                  60  
split:   79 86  
          79  
                  86  
merge:    79  
          86  
          79 86  
merge:    79 86  
          60  
          60 79 86  
split:      79 76  
          79  
          76  
merge:      79  
          76  
          76 79  
merge:    60 79 86  
          76 79  
          60 76 79 79 86  
  
split:          71 44 88 58 23  
          71 44 88  
                  58 23  
split:          71 44 88  
          71 44  
                  88  
split:          71 44  
          71  
          44
```

Implementare ordinamento per fusione

```
split:          71 44 88 58 23
                71 44 88
                58 23
split:          71 44 88
                71 44
                88
split:          71 44
                71
                44
merge:          71
                44
                44 71
merge:          44 71
                88
                44 71 88
split:          58 23
                58
                23
merge:          58
                23
                23 58
merge:          44 71 88
                23 58
                23 44 58 71 88
merge: 60 76 79 79 86
                23 44 58 71 88
                23 44 58 60 71 76 79 79 86 88
Sorted array:
23 44 58 60 71 76 79 79 86 88
```

Efficienza merge sort

- L'ordinamento per fusione è più efficiente di quello per inserzione che per selezione
- La prima chiamata alla funzione *sortSubArray* produce
 - Due chiamate ricorsive alla stessa funzione, ognuna con un sottoarray che è circa la metà di quello originario
 - Una chiamata alla funzione *merge*
- La chiamata alla funzione *merge* richiede nel peggio dei casi $n-1$ confronti ($O(n)$)
- Le due chiamate alla funzione *sortSubArray* producono
 - Altre quattro chiamate ricorsive, con un array di circa un quarto degli elementi dell'originale
 - Altre due chiamate a *merge*
- Continua fino ad arrivare ad array a dimensione 1
- Il numero di «livelli» è logaritmico in n siccome la size si dimezza ad ogni livello
- Il tempo di esecuzione totale è pari a $O(n \log n)$

Confronto tra algoritmi di ordinamento

- Tempo di esecuzione di alcuni algoritmi di ordinamento con notazione O grande
 - Ordinamento per **inserzione** $O(n^2)$
 - Ordinamento per **selezione** $O(n^2)$
 - Ordinamento per **fusione** $O(n \log n)$

Confronto tra diversi ordini

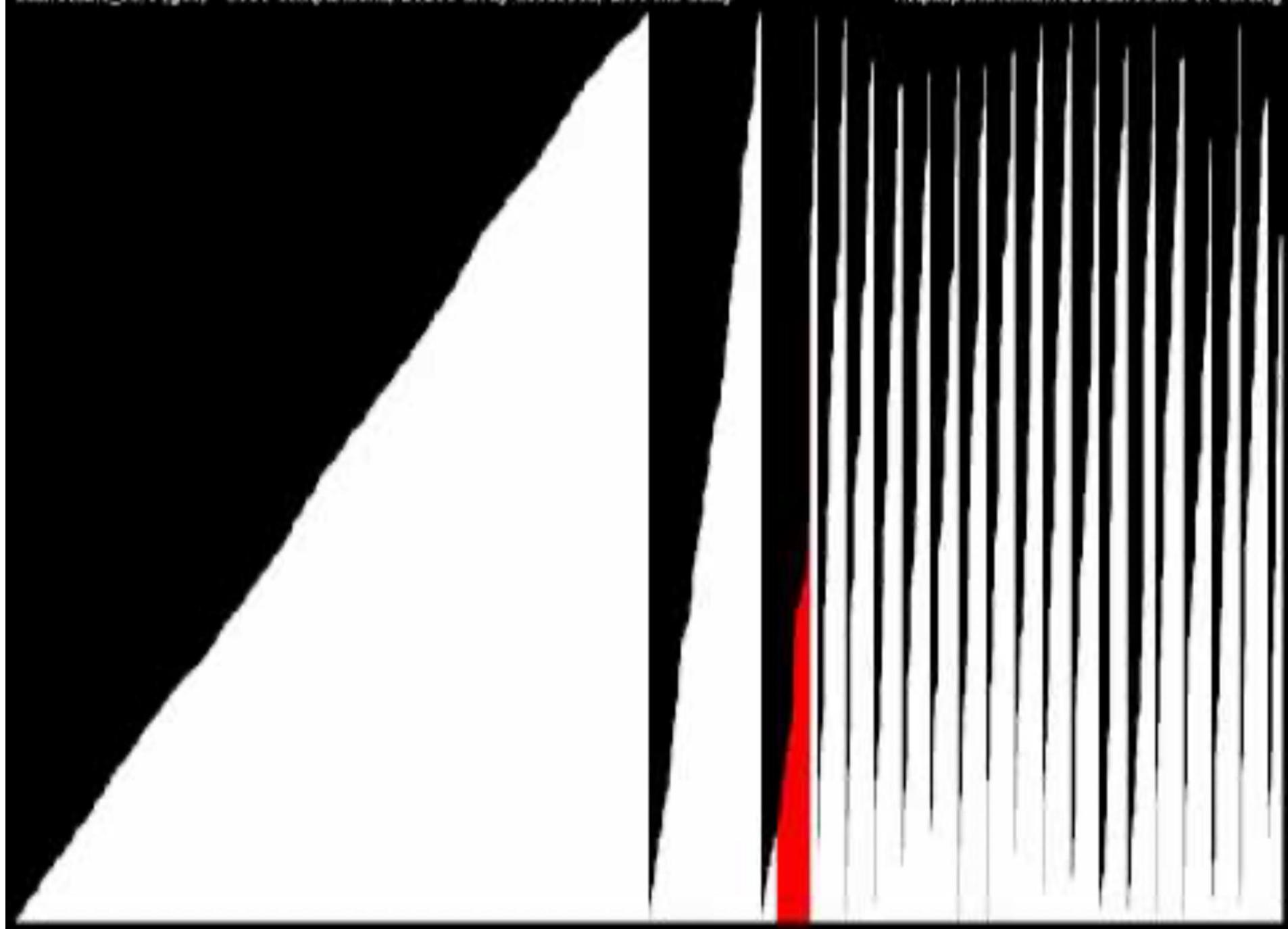
n	Valore decimale approssimato	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1.000	10	2^{10}	10×2^{10}	2^{20}
2^{20}	1.000.000	20	2^{20}	20×2^{20}	2^{40}
2^{30}	1.000.000.000	30	2^{30}	30×2^{30}	2^{60}

Altri algoritmi

- **Quick Sort:** basato sulla tecnica di partizionamento che ha una complessità di $O(n \log n)$ nel caso medio, ma nel caso peggiore può arrivare a $O(n^2)$
- **Heap Sort:** sfrutta una struttura di dati chiamata heap per garantire una complessità di $O(n \log n)$, sia nel caso migliore che peggiore, ed è particolarmente utile quando è necessario ordinare i dati senza utilizzare spazio aggiuntivo
- **Counting Sort:** è un algoritmo di ordinamento non basato su confronti che utilizza un array di conteggio per determinare la posizione di ciascun elemento nell'array ordinato, ed è particolarmente efficiente quando gli elementi da ordinare appartengono a un intervallo ristretto, con una complessità di $O(n + k)$, dove n è il numero di elementi e k è il valore massimo presente nell'array

`std::stable_sort` (gcc) - 8950 comparisons, 20268 array accesses, 1.00 ms delay

<http://panthema.net/2013/sound-of-sorting>



Recap

- la notazione O grande indica lo sforzo che un algoritmo può dover compiere per risolvere un problema
- $O(1)$ = tempo costante
- $O(n)$ = tempo lineare
- $O(n^2)$ = tempo quadratico
- Ordinamento per **selezione** richiede **$O(n^2)$**
- Ordinamento per **inserzione** richiede **$O(n^2)$**
- Ordinamento per **fusione** richiede **$O(n \log(n))$**