Ordinamento di dati

Ordinamento: Motivazioni

- Un problema molto frequente nella elaborazione dei dati consiste nell'***ordinamento*** dei dati stessi, secondo un criterio prestabilito
  - numeri in ordine crescente (o decrescente),
    stringhe in ordine lessicografico (crescente o decrescente),
    conti bancari in ordine di saldo (crescente o decrescente)…

- Una sequenza di n "oggetti" **confrontabili**

$$(a_0, a_1, a_2, \ldots a_{n-1})$$

si dice ***ordinata*** quando

$$a_i <= a_{i+1} \text{ per ogni } i=0,\ldots n-2$$

Ordinamento Motivazioni

- Tra i più studiati problemi in informatica
  - Fucina di idee algoritmiche
  - Moltissime soluzioni
- Analisi di complessità computazionale
  - Caso migliore/peggiore/medio
- L'ordinamento è un passaggio chiave su cui molti altri algoritmi sono costruiti.

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Utilizziamo l'ordinamento per essere piu' efficienti nel…

2	5	9	12	17	25	29	29	42	57	57

- **Cercare un valore** (pensate al dizionario)
- Trovare min/max o k-th min/max
- Ricerca della coppia di elementi più vicina
  - Verificare unicità di un elemento (o trovare duplicati)
- Distribuzione di frequenza degli elementi
- Intersezione/unione di array ordinati

Ordinamento: Motivazioni

- **Studieremo** tre algoritmi di ordinamento:
  - Selection sort,
  - Mergesort
  - Insertion sort

- Altri ne vedrete a Dati e Algoritmi 1

Ordinamento: Modalità

- In generale ci occuperemo di ordinamento di dati presenti in un ***contenitore (finora abbiamo visto solo gli array)***

- Nei problemi di ordinamento si parla di
  - Ordinamento "**sul posto**" quando il metodo di ordinamento riceve un contenitore e ne ordina il contenuto
    - Non restituisce nulla
    - Il metodo invocante si ritrova il contenitore ordinato
  - Ordinamento "**non sul posto**" quando il metodo di ordinamento riceve un contenitore e ne restituisce **un altro** contenente gli stessi elementi, ma in ordine; il contenitore ricevuto non viene modificato

- Noi vedremo quelli "sul posto"

Metodi per l'ordinamento

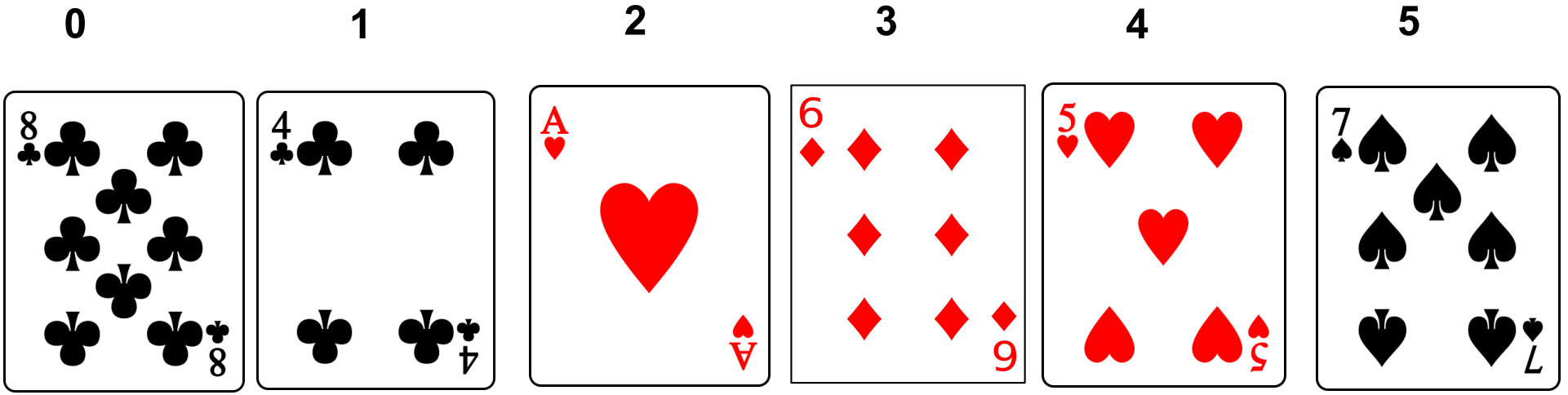
- I metodi che ordinano array sono **metodi statici**
  - Devono ricevere un array come parametro, ma questo non può essere il parametro implicito del metodo, perché non esiste la "classe array" in cui definire il metodo di ordinamento

- Ordineremo array "pieni" (dimensione logica = dimensione fisica)

- Con modifiche minime, gli stessi algoritmi possono ordinare:
  - array "riempiti solo in parte", fornendo la dimensione logica come parametro aggiuntivo
  - porzioni di array, fornendo due parametri aggiuntivi: indice iniziale (incluso) e indice finale (escluso), in analogia con il metodo **substring** della classe **String**

Ordinamento per selezione

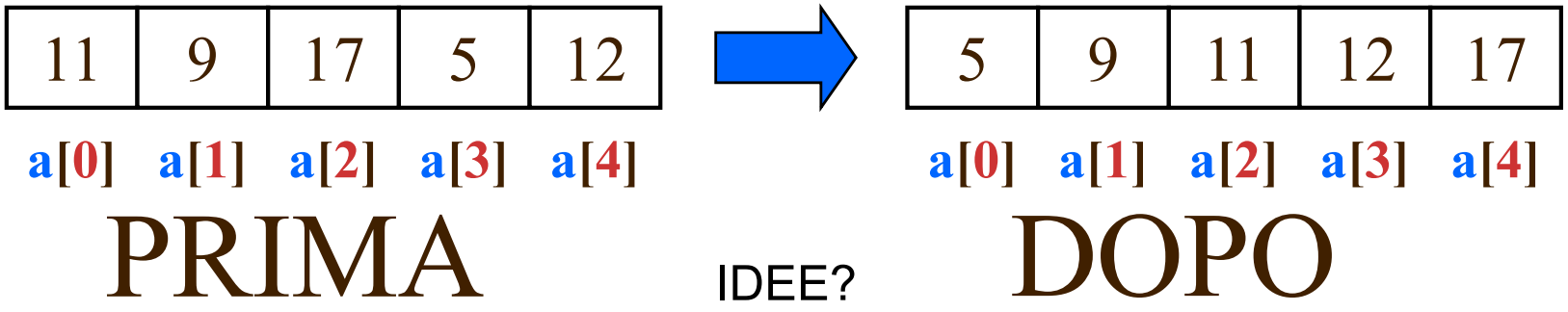
ATTENZIONE: non cerchiamo il modo migliore per risolvere il problema dell'ordinamento, semplicemente analizziamo uno dei tanti algoritmi che risolvono questo problema; successivamente vedremo algoritmi "migliori"


0
1
2
3
4
5

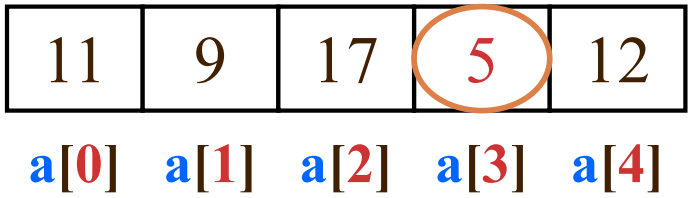
Ordinamento per selezione

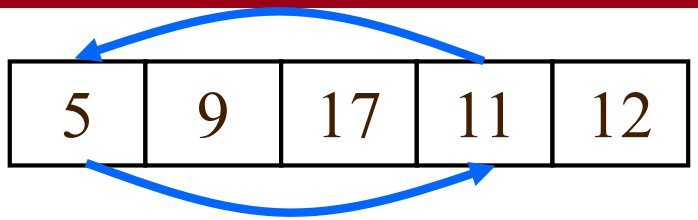
- Per semplicità, analizzeremo prima algoritmi per ***ordinare numeri*** memorizzati in un array

- Prendiamo in esame un array **a** da ordinare ***in senso crescente*** (meglio: non decrescente in caso di duplicati)
  - Cioè al crescere dell'indice di cella, cresce (meglio, non decresce) il valore in essa memorizzato



Ordinamento per selezione





- Per prima cosa, bisogna trovare ***la posizione dell'elemento minimo presente nell'intero array***, come sappiamo già fare
  - in questo caso l'elemento minimo è **5** in posizione 3
- Essendo l'elemento minimo, la sua ***posizione finale corretta*** nell'array ordinato è 0
  - in **a**[0] è però memorizzato il numero **11**, da spostare
  - non sappiamo quale sarà la posizione finale di **11**
    - lo spostiamo temporaneamente in **a**[3]
  - quindi, *scambiamo* **a**[3] con **a**[0]

Ordinamento per selezione

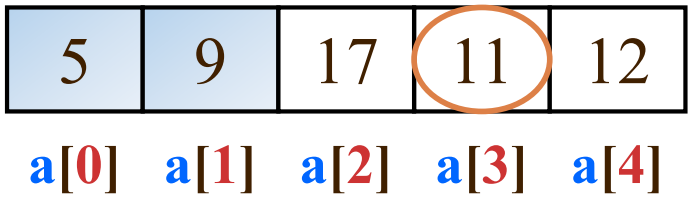
5	9	17	11	12
a[0]	a[1]	a[2]	a[3]	a[4]

la parte colorata dell'array è già ordinata

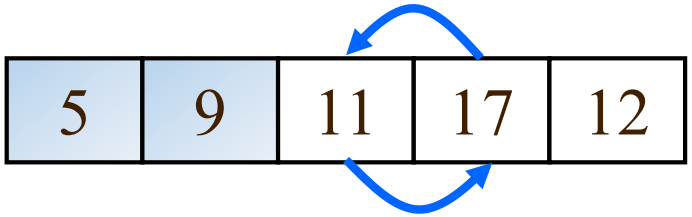
- La parte sinistra dell'array è già ordinata e non sarà più considerata, dobbiamo ordinare la parte destra
- Ordiniamo la parte destra ***con lo stesso algoritmo***
  - cerchiamo l'elemento minimo nella porzione non colorata: è **9** in posizione **1**
  - dato che ***è già nella prima posizione della parte da ordinare, non c'è bisogno di fare scambi***
  - **la porzione ordinata cresce…**

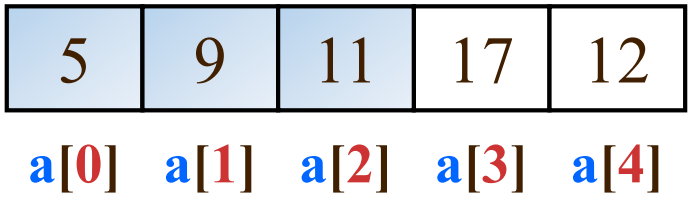
5	9	17	11	12
a[0]	a[1]	a[2]	a[3]	a[4]

Ordinamento per selezione

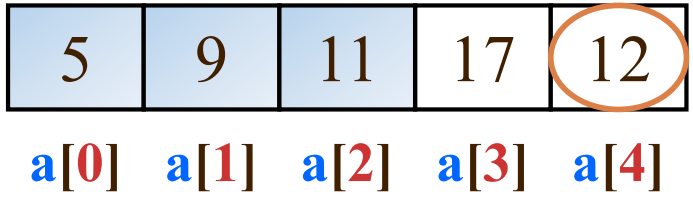


- Proseguiamo per ordinare la parte di array che contiene gli elementi **a[2]**, **a[3]** e **a[4]**
  - l'elemento minimo è il numero **11** in posizione **3**
  - scambiamo **a[3]** con **a[2]**

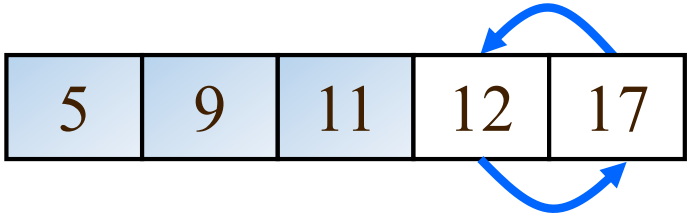


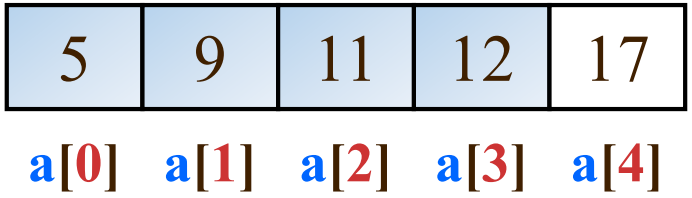


Ordinamento per selezione



- Ora l'array da ordinare contiene **a[3]** e **a[4]**
  - l'elemento minimo è il numero **12** in posizione **4**
  - scambiamo **a[4]** con **a[3]**





- A questo punto *la parte da ordinare contiene un solo elemento*, quindi ***è ovviamente ordinata***

Ordinamento per selezione: implementazione

```
public static void selectionSort(double[] a)
{
    if (a == null) // è un metodo "void"
       return;     // non serve lanciare un'eccezione

    for (int i = 0; i < a.length - 1; i++) // tranne l'ultimo
    {
        int minPos = findMinPosFrom(a, i); // cerco l'elemento

        if (minPos != i)        // che deve andare in posizione i
          swap(a,minPos,i);

    } // ok anche nei casi degeneri, a.length 0 e 1,

}     // nei quali non c'è niente da fare... e non fa niente!
```

Ordinamento per selezione: implementazione

```
private static void swap(double[] a, int i, int j)
{  double temp = a[i]; // non verifico precondizioni
   a[i] = a[j];        // perché è un metodo privato
   a[j] = temp;
}

private static int findMinPosFrom(double[] a, int from)
{
   int pos = from;

   for (int i = pos + 1; i < a.length; i++)
      if (a[i] < a[pos])
         pos = i;

   return pos;

} // è diverso da quello che avevamo progettato...
  // cerca da una certa posizione in poi
```

Metodi ausiliari privati

- Senza i metodi ausiliari il codice è meno immediato
- Con i metodi ausiliari, se si usano nomi appropriati (come è obbligatorio fare!), il codice "si commenta da solo"

```
public static void selectionSort(double[] a)
{  if (a == null) return;
   for (int i = 0; i < a.length - 1; i++)
   {  int pos = i;
      for (int j = pos + 1; j < a.length; j++)
         if (a[j] < a[pos])
            pos = j; // findMinPosFrom...
      if (pos != i)
      {  double temp = a[i];
         a[i] = a[pos];    // swap(a, i, pos)
         a[pos] = temp;
      }
   }
}
```

Ordinare array riempiti in parte

- Per ordinare array riempiti solo in parte basta aggiungere il parametro "dimensione logica", aSize, da usare nel metodo al posto della lunghezza fisica dell'array

- I nuovi metodi possono "convivere" con i precedenti, che diventano sovraccarichi

Ordinare array riempiti in parte

```
public static void selectionSort(double[] a, int aSize)
{
   if (a == null) return;
   if (aSize > a.length) throw new IllegalArgumentException();

   for (int i = 0; i < aSize - 1; i++)
   {  int minPos = findMinPosFrom(a, aSize, i);
      if (minPos != i){
         swap(a, minPos, i); // swap non richiede modifiche
      }
   }
}

private static int findMinPosFrom(double[] a,int aSize,int from)
{  int pos = from;
   for (int i = pos + 1; i < aSize; i++){
      if (a[i] < a[pos]){
         pos = i;
      }
   }
   return pos;
}
```

Ordinamento per selezione

- Possiamo ora sostituire il metodo che ordina un array "pieno" con uno ***stub*** che invoca quello che ordina array riempiti nella parte iniziale

```
public static void selectionSort(double[] a) // stub
{  if (a == null) return;
   selectionSort(a, a.length);
}
```

- Quindi, findMinPos con due soli parametri non serve più

Ordinamento per selezione su array riempiti solo in parte

```
public static void selectionSort(double[] a, int aSize)
{  if (a == null) return; if (aSize > a.length) ...
   for (int i = 0; i < aSize - 1; i++)
   {  int minPos = findMinPosFrom(a, aSize, i);
      if (minPos != i) swap(a, minPos, i);
   }
}
public static void selectionSort(double[] a) // stub
{  if (a == null) return;
   selectionSort(a, a.length);
}
private static void swap(double[] a, int i, int j)
{  double temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
private static int findMinPosFrom(double[] a, int aSize, int from)
{  int pos = from;
   for (int i = pos + 1; i < aSize; i++)
      if (a[i] < a[pos]) pos = i;
   return pos;
}
```

Ordinamento decrescente

- Per ordinare, invece, in senso decrescente, è sufficiente
  - Definire e usare un metodo **`findMaxPosFrom`**, che identifichi di volta in volta la posizione dell'elemento **massimo**
  - In tale metodo, usare il confronto **>** anziché **<**

```
private static int findMaxPosFrom(double[] a, int aSize,
                                  int from)
{  int pos = from;
   for (int i = pos + 1; i < aSize; i++)
      if (a[i] > a[pos])
         pos = i;
   return pos;
}
```

Ordinamento *decrescente* per selezione

```
public static void reverseSelectionSort(double[] a, int aSize)
{  if (a == null) return;
   for (int i = 0; i < aSize - 1; i++)
   {  int minPos = findMaxPosFrom(a, aSize, i);
      if (minPos != i) swap(a, minPos, i);
   }
}
public static void reverseSelectionSort(double[] a) // stub
{  if (a == null) return;
   reverseSelectionSort(a, a.length);
}
private static void swap(double[] a, int i, int j)
{  double temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
private static int findMaxPosFrom(double[] a, int aSize, int from)
{  int pos = from;
   for (int i = pos + 1; i < aSize; i++)
      if (a[i] > a[pos]) pos = i;
   return pos;
}
```

Ordinamento di altri tipi di dati primitivi

Bisogna scrivere altri metodi, ***identici e sovraccarichi*** ai precedenti

```
public static void selectionSort(char[] a, int aSize){
    if (a == null)
        return;
     for (int i = 0; i < aSize - 1; i++){
        int minPos = findMinPosFrom(a, aSize, i);
        if (minPos != i) swap(a, minPos, i);
    }
}
public static void selectionSort(char[] a) // stub
{    if (a == null)
        return;
     selectionSort(a, a.length);
}
private static void swap(char[] a, int i, int j){
    char temp = a[i];    // temp è l'unica variabile di tipo
    a[i] = a[j];         // uguale ai dati ordinati, tutte le altre
    a[j] = temp;         // variabili sono indici, quindi int
}
private static int findMinPosFrom(char[] a, int aSize, int from{
    int pos = from;
    for (int i = pos + 1; i < aSize; i++)
       if (a[i] < a[pos]) pos = i;
    return pos;
}
```

Ordinamento di oggetti, es. Stringhe

Per ordinare array di **stringhe**, usando **`compareTo`** per i confronti

```
public static void selectionSort(String[] a, int aSize)
{  if (a == null) return;
   for (int i = 0; i < aSize - 1; i++)
   {  int minPos = findMinPosFrom(a, aSize, i);
      if (minPos != i) swap(a, minPos, i);
   }
}
public static void selectionSort(String[] a)
{  if (a == null) return;
   selectionSort(a, a.length);
}
private static void swap(String[] a, int i, int j)
{  String temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
private static int findMinPosFrom(String[] a, int aSize, int from)
{  int pos = from;
   for (int i = pos + 1; i < aSize; i++)
      if (a[i].compareTo(a[pos]) < 0) // era a[i] < a[pos]
         pos = i;
   return pos;
}
```

Complessita' di selection sort

- Sia n la dimensione dell'array da ordinare
- Ho un ciclo for su indice i che va da 0 a n-2
- Ho un ciclo for annidato su indice j che va da i+1 a n-1
  - Al suo interno ho un numero costante di passi primitivi

```
public static void selectionSort(double[] a)
{  if (a == null) return;
   for (int i = 0; i < a.length - 1; i++)
   {  int pos = i;
      for (int j = pos + 1; j < a.length; j++)
         if (a[j] < a[pos])
            pos = j; // findMinPosFrom...
      if (pos != i)
      {  double temp = a[i];
         a[i] = a[pos];     // swap(a, i, pos)
         a[pos] = temp;
      }
   }
}
```

Complessita' di selection sort

- Ciclo for esterno:
  - i=0: n-1 cicli for interno + c istruzioni base
  - i=1: n-2 cicli for interno + c istruzioni base
  - i=3: n-3 cicli for interno + c istruzioni base
  - ...
  - i=n-2: 1 ciclo for interno + c istruzioni base
- $T(n) = (n-1+c)+((n-2)+c)+\ldots+(2+c)+(1+c)$

  $= (n-1)+(n-2)+\ldots+3+2+1 + (n-1)*c$

  $= n*(n-1)/2 + (n-1)*c$

  $= O(n^2)$

Cicli annidati: analisi delle prestazioni

- In generale cicli annidati di questo tipo portano a complessità quadratica

```
for (int i = 0; i < n; i++)
    //... operazioni primitive
    for (int j = i; j < n; j++)
       //... operazioni primitive
```

- Per stimare il tempo di esecuzione dobbiamo stimare il numero di operazioni primitive eseguite nel ciclo interno
  - Per i=0 vengono eseguite n volte
  - Per i=1 vengono eseguite n-1 volte
  - Il numero totale è quindi

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

  ovvero $O(n^2)$

Analisi di complessita'

- In un'analisi dettagliata della complessità si dovrà tener conto di tutte le possibilità:
  - caso migliore, tempo richiesto dai casi più fortunati
  - caso medio (Average case), ovvero il tempo medio di esecuzione dell'algoritmo
  - caso peggiore (Worst case), ovvero il tempo richiesto nei casi più "sfortunati"

Analisi di complessita'

- Il ciclo for interno devo eseguirlo in ogni caso fino alla fine della sequenza, quindi non cambia nulla

- Nella nostra analisi abbiamo indicato con c il numero di passi base costanti fatti dentro al ciclo for con indice i
- Questi passi base nel caso peggiore sono
  - Assegnamento pos=i
  - Confronto dell'IF: i != pos
  - 3 istruzioni per lo swap

Selection sort nel caso migliore

- Il caso migliore si ha quando i != pos è sempre falso (=sequenza ordinata in partenza) e risparmio lo swap
- Si tratta di un numero costante di operazioni
  - Caso peggiore: 5 istruzioni
  - Caso migliore: 2 istruzioni
  - Non cambia nulla dal punto di vista asintotico!
- L'ordinamento per selezione è, quindi, **quadratico anche nel caso migliore**, oltre che nel caso peggiore
  - Non trae (**asintoticamente**) alcun vantaggio dal fatto che l'array da ordinare sia già ordinato! Anche se, ovviamente, le prestazioni "a cronometro" migliorano un po'

Selezione nel "caso medio"

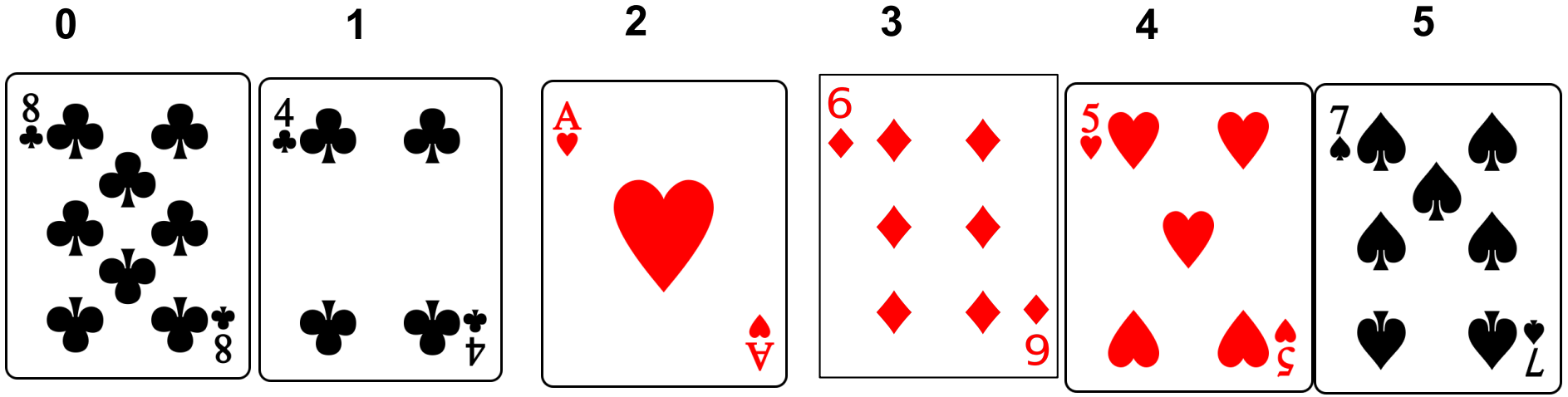
- Di solito le prestazioni nel "caso migliore" hanno poco interesse (meglio le prestazioni nel "caso peggiore" e nel "caso medio")
- Non è semplice definire le condizioni di "caso medio"
  - in teoria, bisogna fare la **media** del numero di accessi necessario per ordinare **tutte le possibili permutazioni dei dati**, cioè **tutte le possibili situazioni iniziali**
  - in pratica, spesso si arriva a una definizione adeguata in modo più semplice, con l'esperienza e la matematica…
  - in questo caso, **dato che l'ordinamento per selezione è quadratico sia nel caso migliore sia nel caso peggiore, non può che essere quadratico anche nel caso medio!**

Take home message

- Il selectionsort è un algoritmo di ordinamento in cui per ogni posizione i dell'array
  - Individuo la posizione *min* dell'elemento minimo da i+1 in poi
  - Se diversa da i, scambio gli elementi in posizione i e min

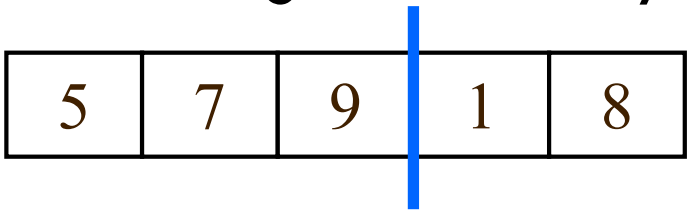
- Complessità $O(n^2)$
  - L'ordine degli elementi in input non influenza le prestazioni
    - Caso migliore = caso peggiore = caso medio

Ordinamento per fusione (*MergeSort*)


0
1
2
3
4
5

MergeSort

- Presentiamo ora un algoritmo di ordinamento (***MergeSort***) che vedremo avere ***prestazioni migliori*** di quelle viste finora
- Dovendo ordinare il seguente array

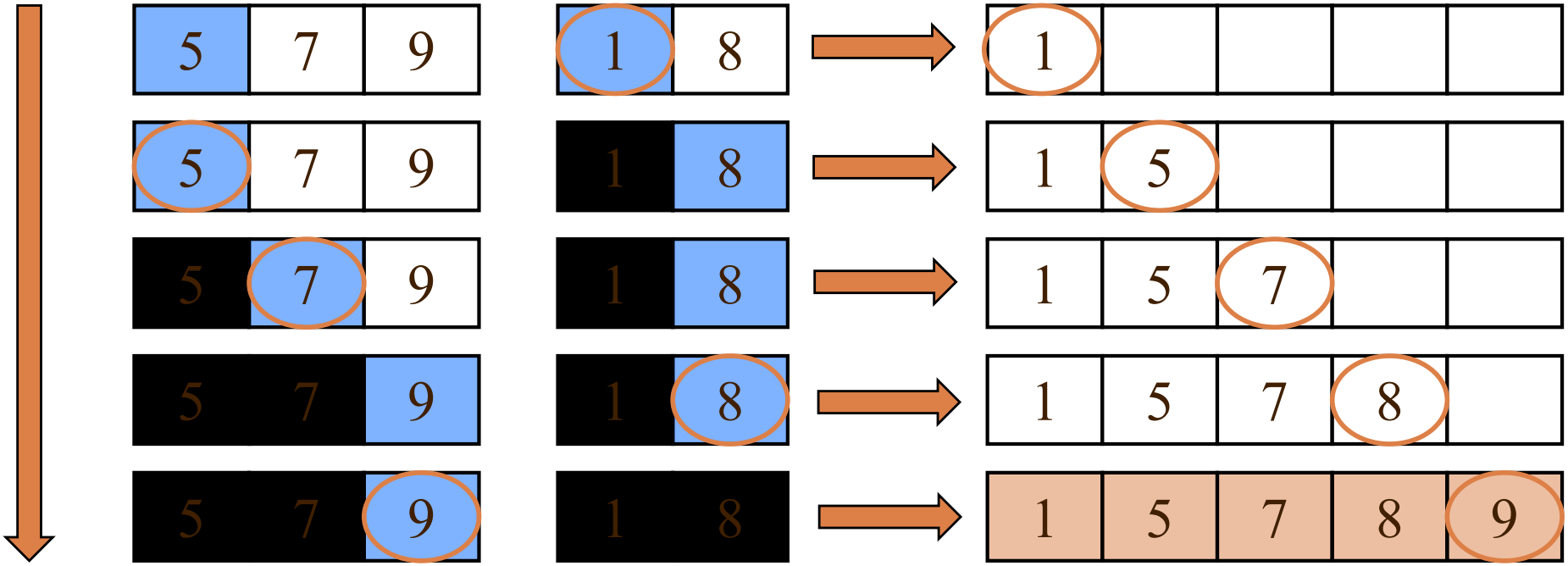


lo dividiamo in due parti di dimensioni (circa) uguali

MergeSort

La fusione NON opera sul posto

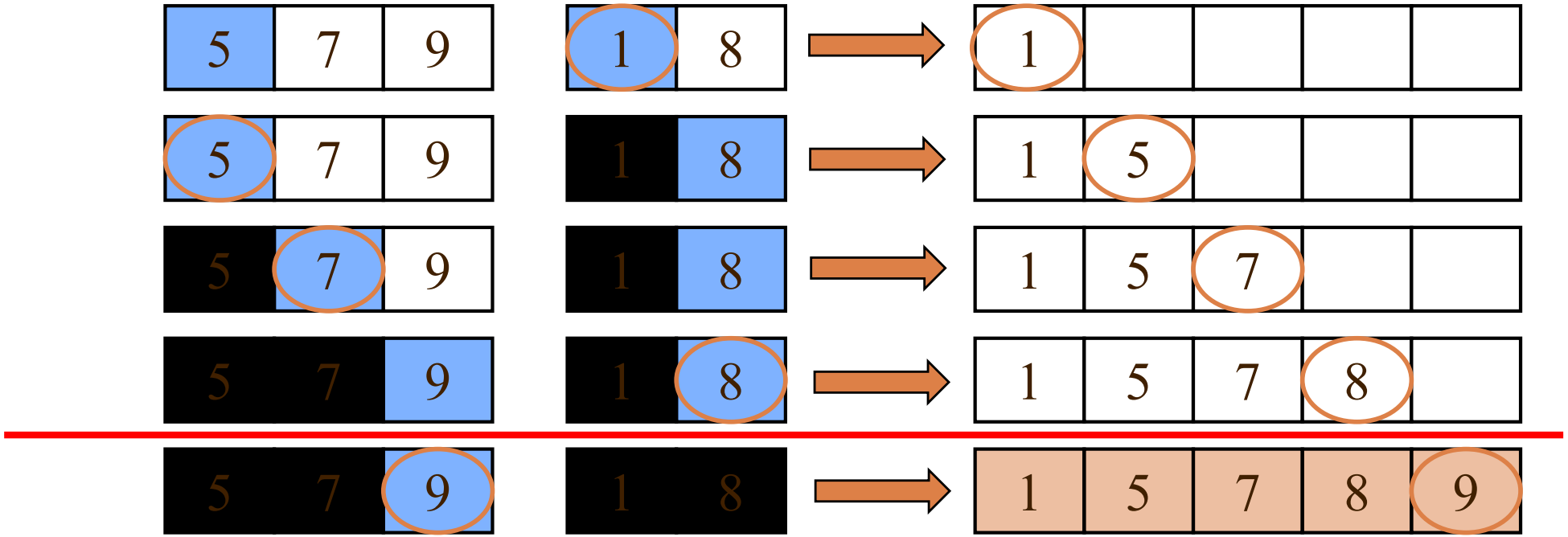
- Se supponiamo che, come in questo caso, le due parti siano già ordinate, è facile costruire l'array ordinato, togliendo sempre ***il primo elemento da uno dei due array***, scegliendo ***il più piccolo***



Array destinato a contenere il risultato della fusione

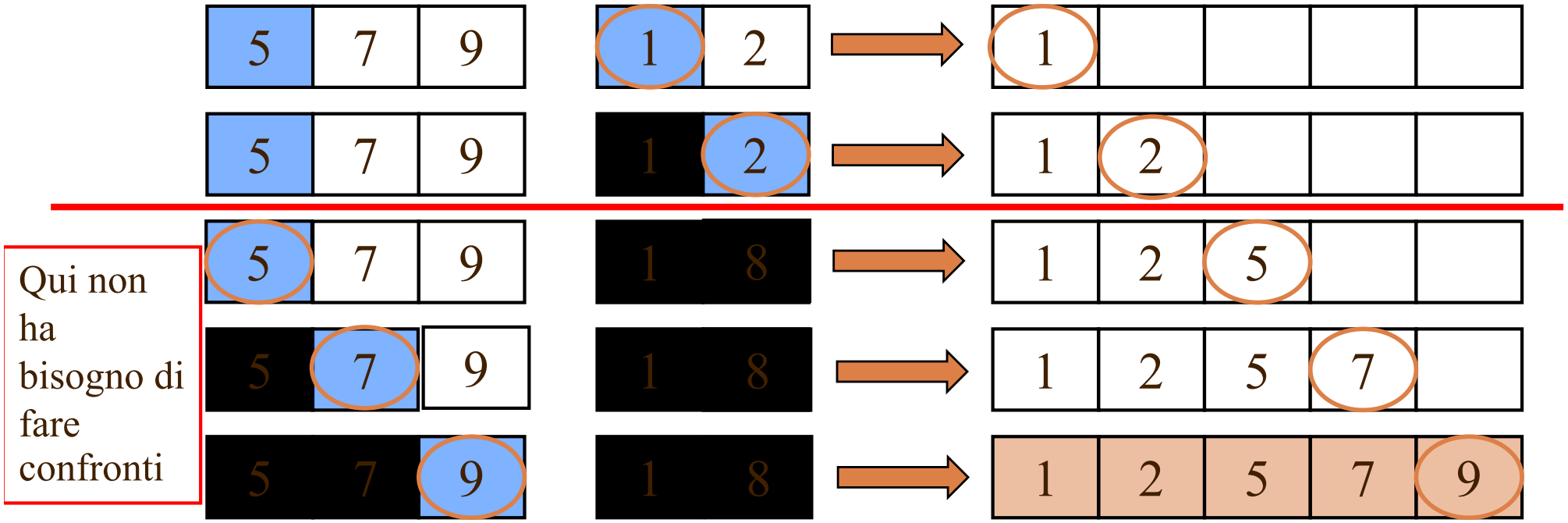
MergeSort

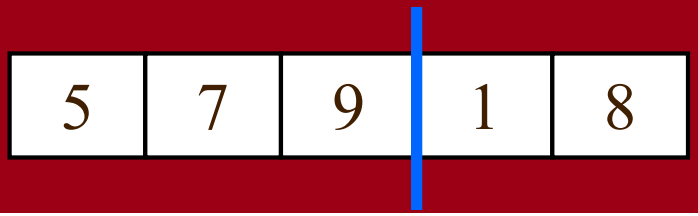
- L'algoritmo di fusione opera in due modalità diverse nelle sue due fasi: nella prima fase effettua confronti, nella seconda (**quando uno dei due semi-array è rimasto vuoto**) non ha più bisogno di fare confronti



MergeSort

- Vediamo un altro esempio di fusione: la seconda modalità di funzionamento dell'algoritmo può coinvolgere anche più di un elemento





MergeSort

- Ovviamente, nel caso generale le due parti dell'array non saranno ordinate
- Possiamo però *ordinare ciascuna parte ripetendo il seguente processo*
  - dividiamo il vettore in due parti (circa) uguali
  - ***chiediamo "a qualcuno" di ordinare entrambe le parti***
  - ***uniamo le due parti ordinate*** nel modo appena visto
    - questa fase si chiama ***fusione*** (***merge***)
- Continuando così, arriveremo certamente a una situazione in cui *le due parti da ordinare sono già ordinate*
  - ***quando contengono un solo elemento!***

Non confondere la **fusione** (*merge*) con l'**ordinamento per fusione** (*mergesort*): la fusione è una fase dell'ordinamento per fusione

MergeSort

- Si delinea quindi un algoritmo ***ricorsivo*** per ordinare un array, chiamato ***MergeSort***
  - Caso base: se l'array contiene meno di due elementi, è già ordinato
  - altrimenti
    - si divide l'array in due parti (circa) uguali
    - si ordina la prima parte (usando ***MergeSort***)
    - si ordina la seconda parte (usando ***MergeSort***)
    - si fondono le due parti **ordinate** usando l'algoritmo di ***fusione*** (***merge***)
- Nota: la realizzazione di ***MergeSort*** in forma iterativa è significativamente più complessa, anche se è possibile

Mergesort: implementazione

```
public static void mergesort(int[] a)
{  if (a == null) return; // è void...
   if (a.length < 2) return; // caso base

   // divido circa a metà
   int mid = a.length / 2;
   int[] left = new int[mid];
   int[] right = new int[a.length - mid];
   System.arraycopy(a, 0, left, 0, left.length);
   System.arraycopy(a, mid, right, 0, right.length);

// passi ricorsivi per due problemi più semplici, si
// tratta di ricorsione multipla (doppia)
   mergesort(left);
   mergesort(right);

// fusione (metodo ausiliario)
   merge(a, left, right);
}
// ordina "sul posto", però usa array temporanei
```

Mergesort : metodo merge

```
private static void merge(int[] a, int[] b, int[] c)
{  int ia = 0, ib = 0, ic = 0; //
   while (ib < b.length && ic < c.length)
      if (b[ib] < c[ic])          // per "cancellare" un
         a[ia++] = b[ib++];       // elemento da b o c
      else                        // incremento il
         a[ia++] = c[ic++];       // relativo indice
   // attenzione ai due cicli che seguono...
   while (ib < b.length)
      a[ia++] = b[ib++];
   while (ic < c.length)
      a[ia++] = c[ic++];
} // NON è un metodo ricorsivo,
  // è solo ausiliario di mergesort
```

```
a[ia++] = b[ib++];
```

```
a[ia] = b[ib];
ia++;
ib++;
```

La variabile viene PRIMA usata, POI incrementata

Mergesort

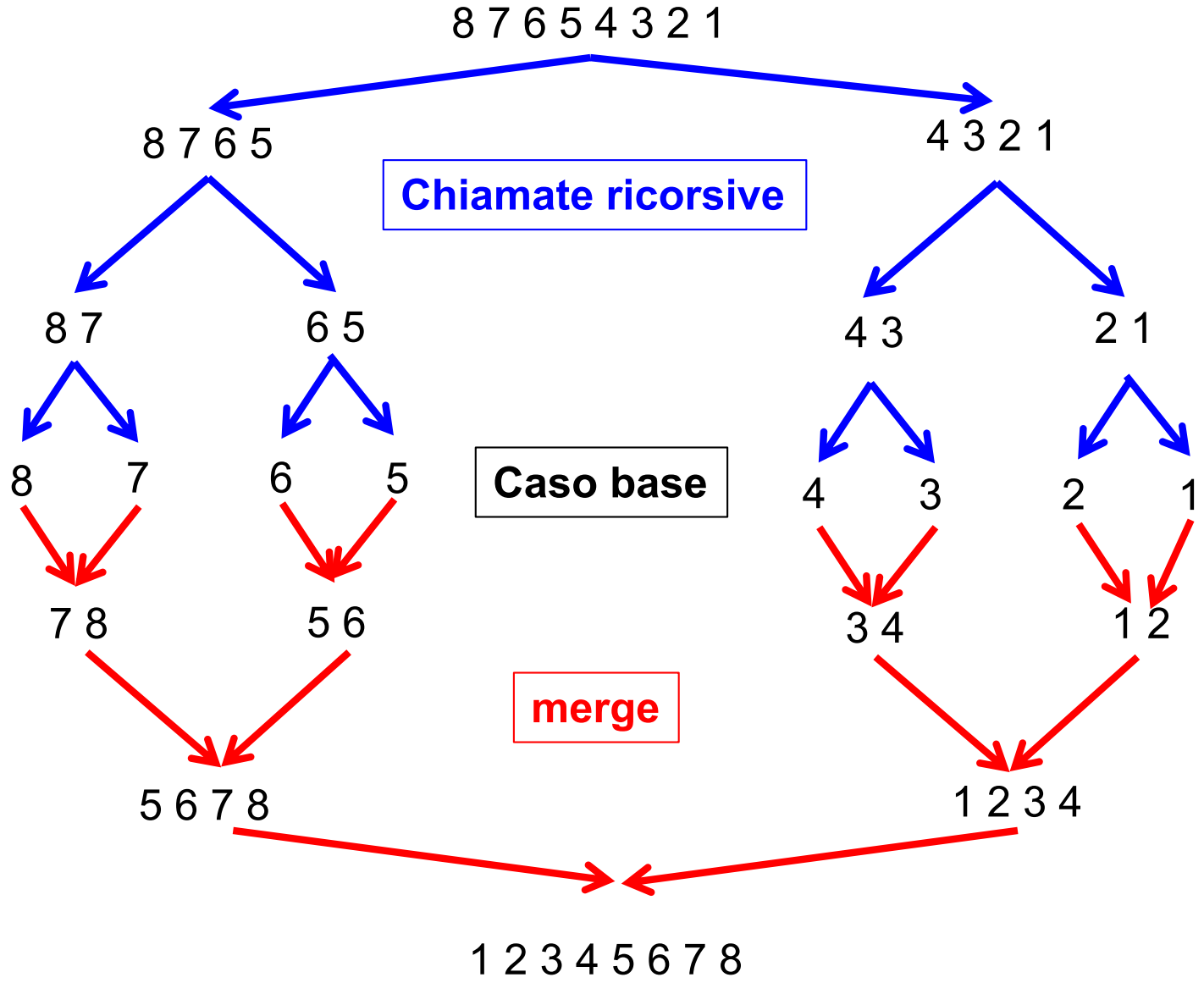
```
private static void merge(int[] a, int[] b, int[] c)
{  int ia = 0, ib = 0, ic = 0;
   while (ib < b.length && ic < c.length)
      if (b[ib] < c[ic]) a[ia++] = b[ib++];
      else a[ia++] = c[ic++];
   // se il ciclo è terminato, uno dei due array
   // (b o c) è certamente "vuoto" dal punto di vista
   // logico, mentre l'altro certamente NON lo è
   // (togliendo un elemento per volta, non possono
   // esaurirsi contemporaneamente)
   // quindi, viene eseguito (almeno una volta)
   // il corpo di UNO E UNO SOLO dei cicli seguenti
   while (ib < b.length) a[ia++] = b[ib++];
   while (ic < c.length) a[ia++] = c[ic++];
}
```

Gli array **b** e **c** sono usati in modalità "riempiti nella parte finale" : per evitare di effettuare veramente le ripetute rimozioni degli elementi iniziali, si incrementano gli indici per "simulare" la rimozione dell'elemento iniziale

Esercizio

- Applicare manualmente l'algoritmo mergesort al seguente input:
- 87654321

Mergesort in esecuzione



Esercizio

- Mergesort(87654321)
  - n=8 mid=4
  - left=8765, right=4321
  - Mergesort (8765)
    - n=4, mid =2
    - Left = 87, right = 65
    - Mergesort(87)
      - N=2, mid =1
      - Left = 8, right =7
      - Mergesort (8) ->return
      - Mergesort(7) -> return
      - Merge(87,8,7) -> array: 78,
    - return

Esercizio

- Mergesort (8765)
  - …
  - Mergesort(65)
    - N=2, mid =1
    - Left = 6, right =5
    - Mergesort (6) ->return
    - Mergesort(5) -> return
    - Merge(65,6,5) -> array: 56
  - Return
- Merge(8765, 78, 56) -> 5678
- return

Esercizio

- Mergesort(87654321)
  - …
  - Mergesort (4321)
    - n=4, mid =2
    - Left = 43, right = 21
    - Mergesort(43)
      - N=2, mid =1
      - Left = 4, right =3
      - Mergesort (4) ->return
      - Mergesort(3) -> return
      - Merge(43,4,3) -> array: 34,
    - return

Esercizio

- Mergesort (4321)
  - …
  - Mergesort(21)
    - N=2, mid =1
    - Left = 2, right =1
    - Mergesort(2) ->return
    - Mergesort(1) -> return
    - Merge(21,2,1) -> array: 12,
  - Return
- Merge(4321,34,12) - > 1234
- return

Esercizio

- Mergesort(87654321)
  - Mergesort(8765) … FATTO!
  - Mergesort (4321) … FATTO!
  - Merge(87654321, 5678,1234) -> 12345678

Prestazioni di MergeSort

- Rilevazioni sperimentali delle prestazioni mostrano che l'ordinamento con MergeSort è molto più efficiente dell'ordinamento con selezione

n	Selection sort	Merge sort
1000	1.32	0.12
2000	5.07	0.25
4000	20.06	0.55
8000	80.02	1.13
16000	323.51	2.42
32000	1284	5.08
64000	5105	10.37
128000	20612	21.63
256000	82448	43.69

Prestazioni di MergeSort

- Rilevazioni sperimentali delle prestazioni mostrano che l'ordinamento con MergeSort è molto più efficiente dell'ordinamento con selezione
  - **La curva del tempo**, in funzione della dimensione, **cresce poco più rapidamente di una retta**
- Cerchiamo di fare una valutazione teorica, calcolando il **numero di accesi ai singoli elementi dell'array** che sono necessari per l'ordinamento
- Chiamiamo `T(n)` il numero di accessi necessari per ordinare un array di `n` elementi

Mergesort

```
public static void mergesort(int[] a)
{   if (a == null) return; // è void...
    if (a.length < 2) return; // caso base

    // divido circa a metà
    int mid = a.length / 2;
    int[] left = new int[mid];
    int[] right = new int[a.length - mid];
    System.arraycopy(a, 0, left, 0, left.length);
    System.arraycopy(a, mid, right, 0, right.length);

//  passi ricorsivi per due problemi più semplici, si
//  tratta di ricorsione multipla (doppia)
    mergesort(left);
    mergesort(right);

//  fusione (metodo ausiliario)
    merge(a, left, right);
}
// ordina "sul posto", però usa array temporanei
```

n

2n

2T(n/2)

4n caso peggiore, 3n caso migliore

Prestazioni di MergeSort

- La creazione dei due semi-array e le loro inizializzazioni di default richiedono complessivamente `n` accessi
- Le due invocazioni di **System.arraycopy** richiedono complessivamente `2n` accessi
  - Tutti gli `n` elementi devono essere letti (una volta) dall'array e scritti (una volta) in uno dei semi-array
- Ciascuna delle due invocazioni ricorsive richiede un numero di accessi uguale a `T(n/2)`
  - Il fatto che `n` sia pari o dispari è assolutamente ininfluente quando `n` è molto grande
  - La fusione richiede …

Mergesort

```
private static void merge(int[] a, int[] b, int[] c)
{  int ia = 0, ib = 0, ic = 0; //

   while (ib < b.length && ic < c.length){
      if (b[ib] < c[ic])
         a[ia++] = b[ib++];
      else
         a[ia++] = c[ic++];
   }
   while (ib < b.length){
      a[ia++] = b[ib++];
   }
   while (ic < c.length){
      a[ia++] = c[ic++];
   }
}
```

Conto il numero di accessi agli array.

La fusione richiede esattamente **n accessi per scrivere i valori finali nell'array a.**

L'individuazione del valore da scrivere richiede (al massimo, **caso peggiore**) 3 accessi: due per il confronto e uno per l'assegnazione: **3n accessi.**

In realtà se tutti gli elementi di un array sono più piccoli dell'altro (**caso migliore**) faccio: 3x(n/2) accessi nel primo while e (n/2) accessi (non serve il confronto) in uno dei due while successivi: quindi **2n accessi.**

mergesort: prestazioni

$$T(n) = 2T(n/2) + cn =$$

$$= 2(2T(n/4)+cn/2) + cn =$$

$$= 4(2T(n/8)+cn/4) + cn + cn =$$

$$= 8T(n/8)+ 3cn = 2^3T(n/2^3)+3cn$$

$$= 2^kT(n/2^k)+ kcn =$$

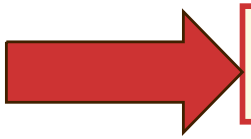
mergesort: prestazioni

$$T(n) = 2^k T(n/2^k) + kcn =$$

$$k = \log_2 n$$

$$= 2^{\log_2 n} T(n/2^{\log_2 n}) + cn\log_2 n$$

$$= n\,T(1) + cn\log_2 n = n + cn\log_2 n$$



$$O(n \log n)$$

Prestazioni di MergeSort

$$T(n) \in O(n \log_2 n)$$

- È facile verificare (ricordando la discussione fatta in merito al numero di accessi necessari per la sola fusione) che, nel caso migliore (quando l'array è già inizialmente ordinato), la costante moltiplicativa 7 diventa **6**, quindi le prestazioni asintotiche non cambiano
  - **Quando un algoritmo ha le stesse prestazioni asintotiche nel caso migliore e nel caso peggiore, non può che avere le medesime prestazioni anche nel caso medio**

Prestazioni di MergeSort

$$T(n) \in O(n \log_2 n)$$

- Come già detto, nelle notazioni "O grande" non si indica la base dei logaritmi, perché il logaritmo in una base si può trasformare nel logaritmo in un'altra base con un fattore moltiplicativo, che va ignorato

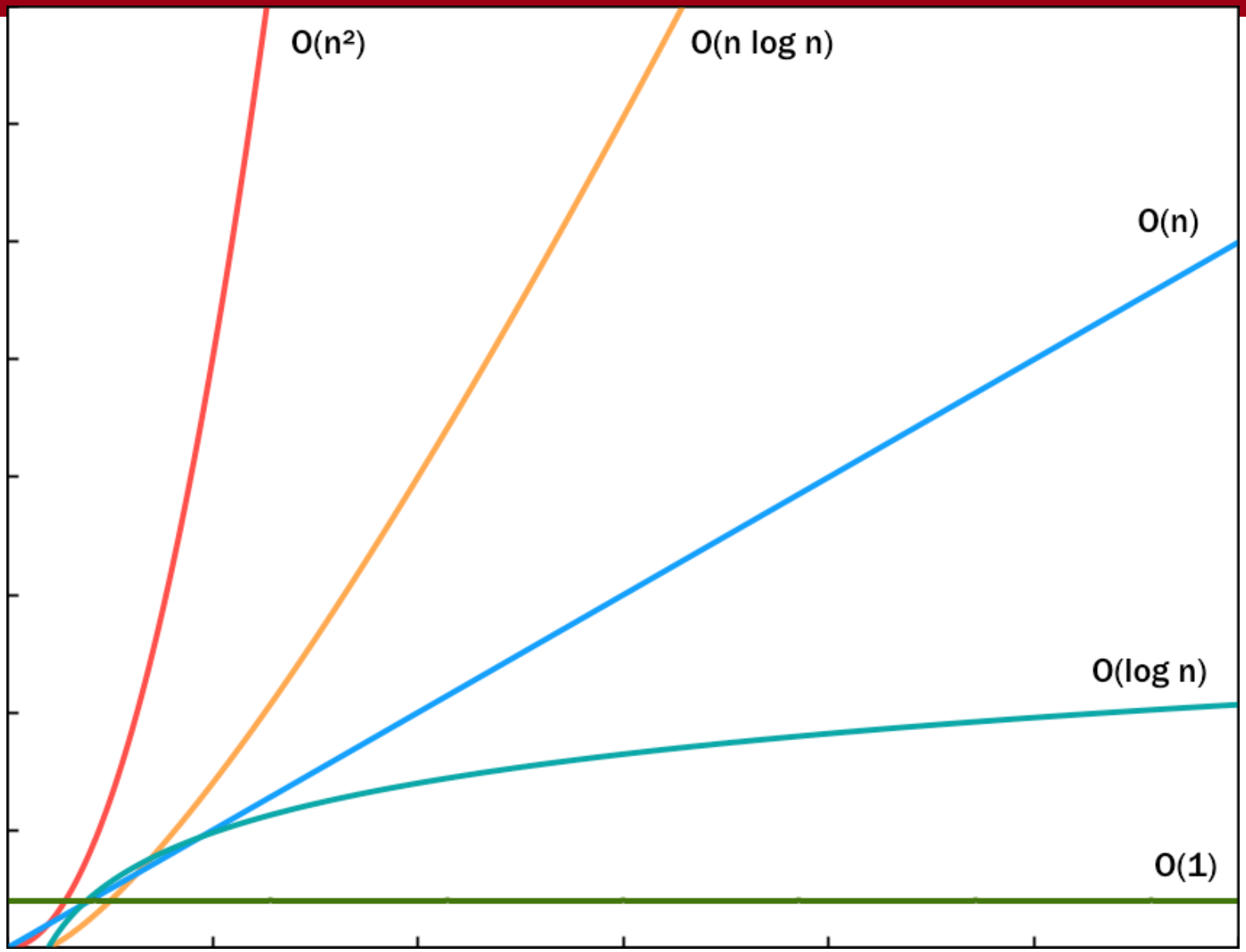
$$\log_b x = \frac{\log_k x}{\log_k b}$$

- Quindi, le prestazioni dell'ordinamento MergeSort hanno un andamento

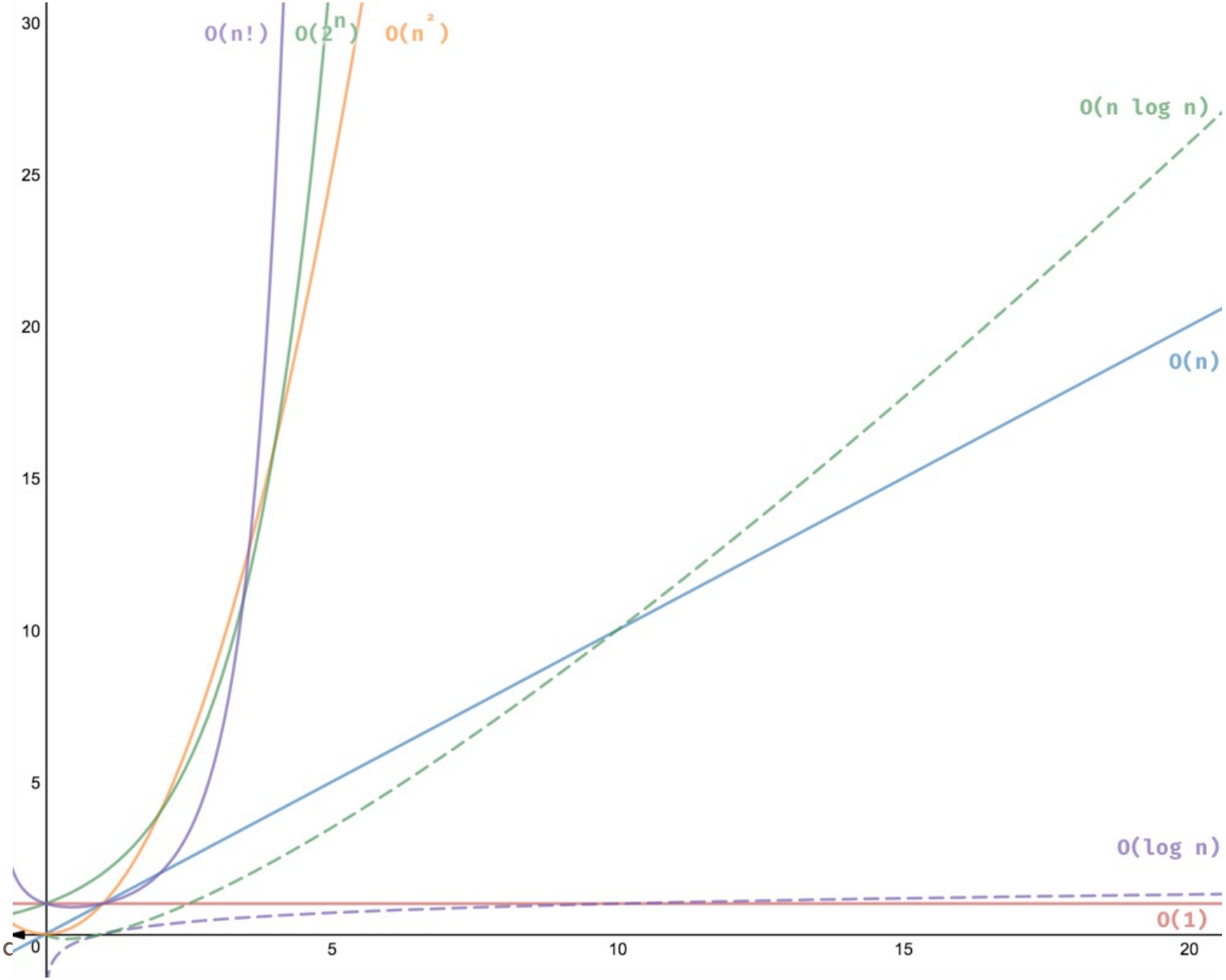
$$O(n \log n)$$

e sono, pertanto, (molto) migliori di $O(n^2)$

Andamenti asintotici



Andamenti asintotici



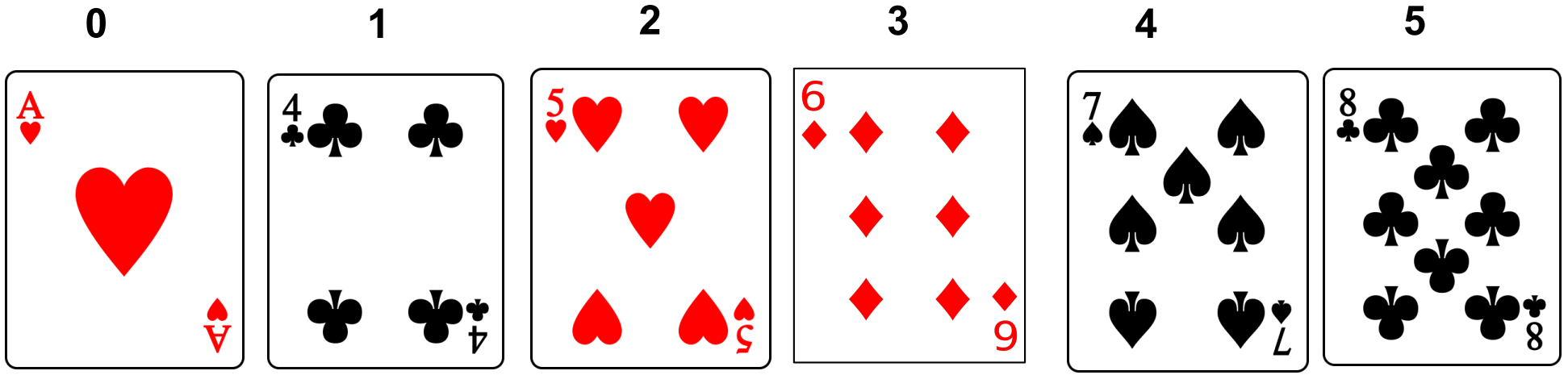
Qualche numero empirico per confrontare

n	n^2	n log n	n^2 / n log n
10	100	66.44	1.51
100	10,000	1,328.77	7.53
1,000	1,000,000	19,931.57	50.17
10,000	100,000,000	265,754.25	376.29
100,000	10,000,000,000	3,321,928.09	3,010.30
1,000,000	1,000,000,000,000	39,863,137.14	25,085.83
10,000,000	100,000,000,000,000	465,069,933.28	215,021.43

Take home message

- Il mergesort è un algoritmo di ordinamento basato sulla ricorsione
  - Divido in due l'input
  - Ordino le due parti ricorsivamente
  - Ottengo un'unica sequenza dalla fusione delle due
- Complessità O(n logn)
  - Algoritmo ottimo tra quelli basati sul confronto
  - Non è però il più veloce, esistono molti approcci diversi all'ordinamento

Ordinamento per inserimento

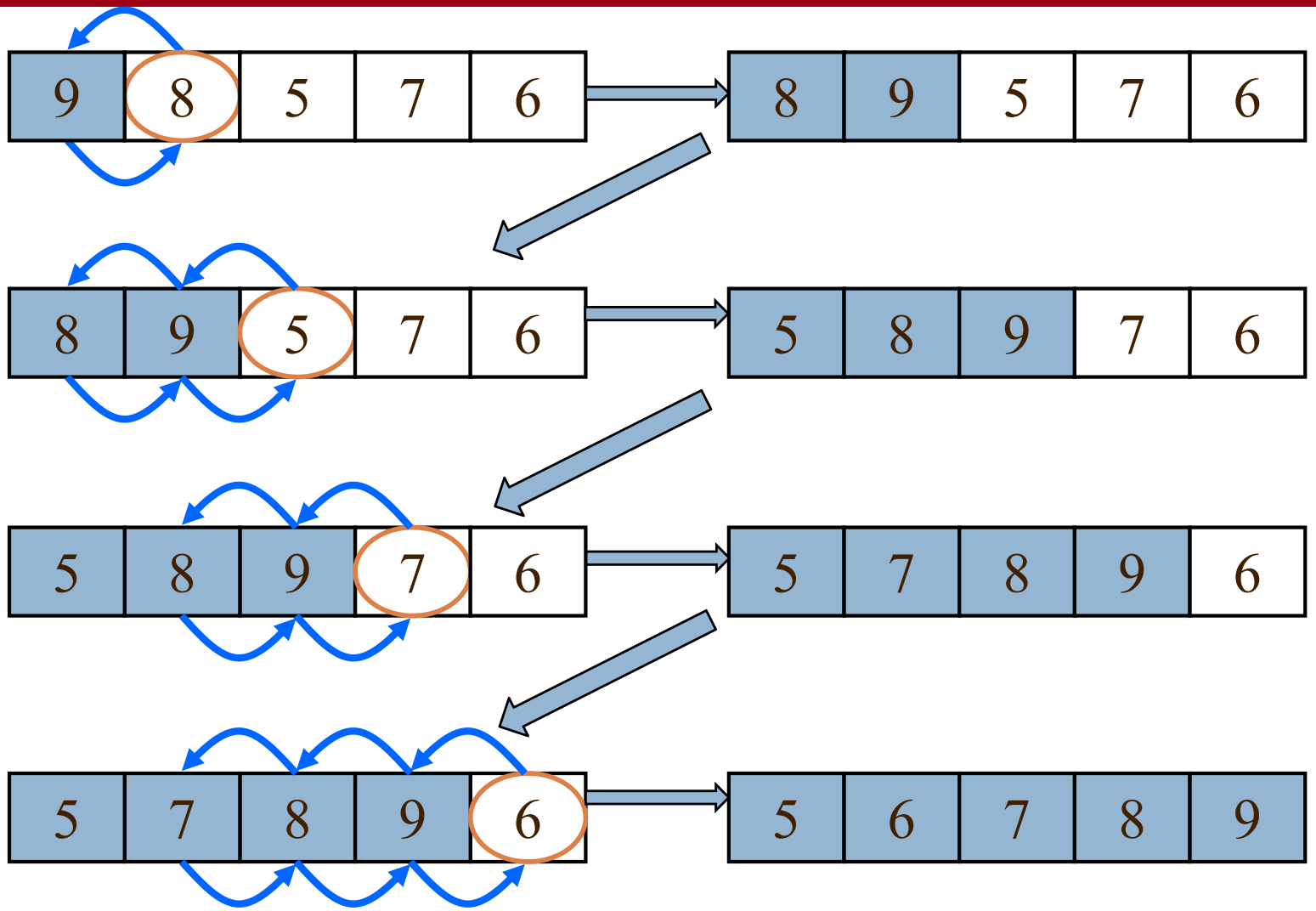

0
1
2
3
4
5

Ordinamento per inserimento

- L'algoritmo di ordinamento per inserimento
  - inizia osservando che il sotto array di lunghezza unitaria costituito dalla prima cella dell'array è ordinato (essendo, appunto, di lunghezza unitaria)
  - procede estendendo verso destra la parte ordinata, includendo nel sottoarray ordinato il primo elemento alla sua destra
    - per farlo, il nuovo elemento viene spostato verso sinistra finché non si trova nella sua posizione corretta, spostando verso destra gli elementi intermedi

9	8	5	7	6
a[0]	a[1]	a[2]	a[3]	a[4]

Ordinamento per inserimento



Ordinamento per inserimento

```
public static void insertionSort(int[] a)
{  // il ciclo inizia da 1 perché il primo
   // elemento non richiede attenzione
   for (int i = 1; i < a.length; i++)
   {  // sposto a[i] verso sinistra finché serve:
      // il primo elemento con cui confrontarlo è
      // a[i-1], quindi parto con j = i-1
      for (j = i-1; j >= 0; j--)
         if (a[j] > a[j+1])
            // devo spostare a[j+1] verso sinistra
            swap(a, j, j+1); //
         else       // non devo più spostare
            break; // è arrivato nel posto giusto
   }
}
```

Analisi teorica delle prestazioni

- Ordiniamo con inserimento un array di `n` elementi
- Il ciclo esterno esegue sempre `n-1` iterazioni (non esistono condizioni di terminazione anticipata)
  - ad ogni iterazione viene eseguito il ciclo interno
- Il ciclo interno esegue 2+4 accessi per ogni sua iterazione (i 4 accessi sono all'interno dell'invocazione di **swap**)
  - tranne **eventualmente** l'ultima iterazione che può richiedere soltanto 2 accessi (se esegue il **break**) oppure 2+4, dipende dall'esito della verifica nell'enunciato **if**
  - Quante iterazioni esegue il ciclo interno?
    - **dipende da come sono ordinati i dati!**

Ordinamento per inserimento

- **Caso migliore**: i dati sono già ordinati
- Il ciclo più interno non esegue mai iterazioni
  - Ogni volta richiede 2 soli accessi per la prima verifica, che lo fa terminare senza nessuno scambio
- Il ciclo esterno esegue `n-1` iterazioni
- $T(n) = 2*(n-1) \in O(n)$

Si dimostra che tali prestazioni valgono anche quando gli elementi **fuori ordine** sono "pochi", in particolare quando ce n'è **soltanto uno** e si trova nella posizione più a destra

$T(n) = 2*(n-2)+6*(n-1) \in O(n)$
quando, **nel caso peggiore di questa condizione particolare**, l'unico elemento fuori posto (che è all'estremo destro) deve andare a finire all'estremo sinistro

Ordinamento per inserimento

- **Caso peggiore**: i dati sono ordinati a rovescio
- Ciascun nuovo elemento "inserito" richiede lo spostamento di tutti gli elementi alla sua sinistra, perché deve essere inserito in posizione 0
- $T(n) = (6*1) + (6*2) + (6*3) + \ldots + (6*(n-1))$
  $= 6*[1+2+\ldots+(n-1)]$
  $= 6n(n-1)/2$
  $\in O(n^2)$

Ordinamento per inserimento

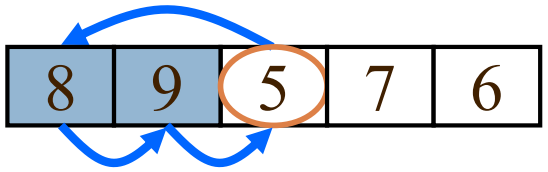
- L'ordinamento per inserimento è molto utilizzato quando si deve inserire un nuovo elemento in un array già ordinato
- Immaginiamo un array riempito solo in parte che viene conservato ordinato per effettuare ricerche, poi ogni tanto c'è bisogno di inserire un elemento nuovo e riordinare
  - Lo inseriamo nella prima posizione libera (cioè "a destra"), eventualmente ridimensionando l'array, poi per ordinarlo
    - Con selectionSort ci vuole un tempo $O(n^2)$
    - Con mergeSort ci vuole un tempo O(nlogn)
    - Con insertionSort ci vuole un tempo $O(n)$, decisamente migliore
  - Se si sa che l'unico elemento fuori posto è l'ultimo, è inutile eseguire l'intero algoritmo, perché le prime $n - 2$ iterazioni del ciclo più esterno non produrranno nessuno scambio, quindi si può "spingere verso sinistra" soltanto l'ultimo elemento, finché serve

Ordinamento per inserimento

```
public static void insertionSortLast(int[] a)
{  // ipotizza che l'unico elemento fuori posto sia
   // l'ultimo (ma non verifica tale ipotesi)
   for (int j = a.length-2; j >= 0; j--)
      if (a[j] > a[j+1])
         swap(a, j, j+1);
      else
         return;
   // ottenuto dal metodo generale presentato in
   // precedenza eseguendo, nel ciclo esterno, la
   // sola ultima iterazione, quando i = a.length-1
}
```

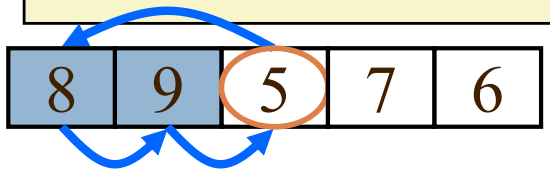
Ordinamento per inserimento

- Solitamente, poi, l'algoritmo viene implementato in un modo un po' più complicato, per renderlo un po' più veloce (anche se non asintoticamente più veloce)
- Nel ciclo interno si evita di scrivere nelle celle intermedie l'elemento che sta scorrendo verso sinistra, scrivendolo soltanto nella cella in cui termina il suo scorrimento
- Il codice è riportato nel seguito



```
public static void insertionSort(int[] a)
{  // il ciclo inizia da 1 perché il primo
   // elemento non richiede attenzione
   for (int i = 1; i < a.length; i++)
   {  // nuovo elemento da inserire
      int temp = a[i];
      // la variabile di ciclo j va definita
      // fuori dal ciclo perché il suo valore
      // finale viene usato in seguito
      int j; // analizzare con cura il significato di j
             // durante il ciclo e alla fine del ciclo
      // vengono spostati a destra di un posto
      // tutti gli elementi a sinistra di temp
      // che sono maggiori di temp, procedendo
      // da destra a sinistra
      for (j = i; j > 0 && temp < a[j-1]; j--)
         a[j] = a[j-1];
      // temp viene inserito in posizione j
      a[j] = temp;
   }
}
```

Questa versione effettua un minor numero di accessi, ma le prestazioni asintotiche non cambiano



In riferimento all'immagine precedente, il ciclo interno effettua gli spostamenti indicati con le frecce blu poste AL DI SOTTO dell'array: gli spostamenti da destra a sinistra si fanno "in un colpo solo", usando **temp**

Esercizio

- Eseguire manualmente l'algoritmo **insertionSort** sull'array 87654321 e sull'array 23456781

Take home message

- L'insertionsort è un algoritmo di ordinamento in cui per ogni posizione i-esima, a partire dalla prima, effettuo i seguenti passi
  - So che le posizioni precedenti sono ordinate
  - "Spingo" verso sinistra l'elemento i-esimo fino a che non raggiungo l'ordinamento
  - Gli elementi maggiori dell'elemento i-esimo vengono fatti scalare di una posizione a destra
- Complessità
  - $O(n^2)$ caso peggiore (si può dimostrare anche nel caso medio)
  - $O(n)$ caso migliore

Confronto di algoritmi

- Se le **notazioni O-grande** delle prestazioni temporali asintotiche di due algoritmi diversi
  - sono tra loro **diverse**, allora è possibile dire quale sia l'algoritmo migliore
  - sono tra loro **uguali**, allora per decidere quale sia l'algoritmo migliore occorre studiare con maggior cura le prestazioni effettive, eventualmente ricorrendo al numero di accessi effettivo e non al suo O-grande, per cui anche i fattori qui trascurati sono rilevanti
    - In mancanza di tale analisi, i due algoritmi si dichiarano "equivalenti" dal punto di vista delle prestazioni asintotiche

Confronto di algoritmi

	caso migliore	caso medio	caso peggiore
merge sort	$n \lg n$	$n \lg n$	$n \lg n$
selection sort	n^2	n^2	n^2
insertion sort	n	n^2	n^2

- **Se si sa che l'array è "quasi" ordinato, è meglio usare l'ordinamento per inserimento**
  - **Caso di notevole interesse:** un array che viene mantenuto ordinato, inserendo ogni tanto un nuovo elemento e poi riordinando

Algoritmi più efficienti?

- **Si può dimostrare** (cfr. corso di Dati e Algoritmi 1) che qualunque algoritmo di ordinamento **che operi mediante confronti tra i valori da ordinare** richiede, **nel caso peggiore**, un tempo che è $\Omega(n \log n)$
- Quindi, ad esempio, l'ordinamento **mergesort** è sia $O(n \log n)$ sia $\Omega(n \log n)$
  - **Di conseguenza, mergesort è $\Theta(n \log n)$**
  - **Si dice anche che è OTTIMO, non si può far meglio**
- È quindi inutile cercare algoritmi ***asintoticamente*** migliori
  - Ma esistono algoritmi con le stesse prestazioni asintotiche e coefficienti moltiplicativi inferiori…
  - Poi, esistono algoritmi di ordinamento che NON fanno confronti…

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Confronto tra algoritmi di ordinamento

- https://visualgo.net/en/sorting
- https://www.toptal.com/developers/sorting-algorithms

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Algoritmi di ricerca

Ricerca sequenziale

Ricerca sequenziale

- Abbiamo già visto un algoritmo da utilizzare per individuare la posizione di un elemento che abbia un particolare valore all'interno di un array i cui elementi non siano ordinati

- Dato che bisogna esaminare tutti gli elementi, uno dopo l'altro in sequenza, si parla di ***ricerca sequenziale o lineare***

```
public static int sequentialSearch(int[] a, int v)
{   if (a == null){
       return -1; // NON TROVATO
    }
    for (int i = 0; i < a.length; i++){
       if (a[i] == v){
          return i; // TROVATO
       }
    }
    return -1;  // NON TROVATO
}
```

Ricerca sequenziale: prestazioni

- Valutiamo le prestazioni dell'algoritmo di ricerca sequenziale
  - se l'elemento cercato **non è presente** nell'array, sono **sempre** necessari `n` accessi
  - se l'elemento cercato **è presente** nell'array, il numero di accessi necessari per trovarlo dipende dalla sua posizione
    - **in media** servono(`n+1)/2` accessi ((1+2+…+n)/n)
    - **al massimo** servono `n` accessi
- **In ogni caso, quindi, le prestazioni dell'algoritmo sono sia nel caso peggiore sia nel caso medio** `O(n)`
- Si tratta, quindi, di un algoritmo con prestazioni ***lineari*** in funzione della dimensione del problema
  - Per questo motivo si parla anche di ***ricerca lineare***

Ricerca sequenziale: prestazioni

- Cosa possiamo dire in merito alle prestazioni della ricerca sequenziale nel caso migliore?
- Nel caso migliore, il primo elemento esaminato è proprio quello cercato!

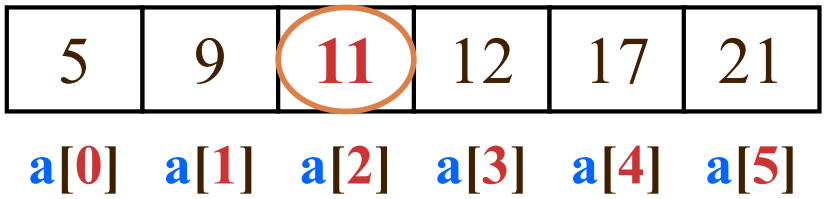
  $T(n) = 1 \in \Theta(1)$

- Ogni volta che il numero di accessi a celle dell'array NON dipende dalla dimensione dell'array, si dice che l'algoritmo viene eseguito in ***tempo costante*** e l'algoritmo è $\Theta(1)$ indipendentemente dal valore della costante stessa
  - Cioè, anche se fosse $T(n) = 5$, sarebbe comunque $T(n) \in \Theta(1)$
  - Rivedere la definizione delle notazioni asintotiche…

Ricerca per bisezione
(o "binaria")

Ricerca in un array ordinato

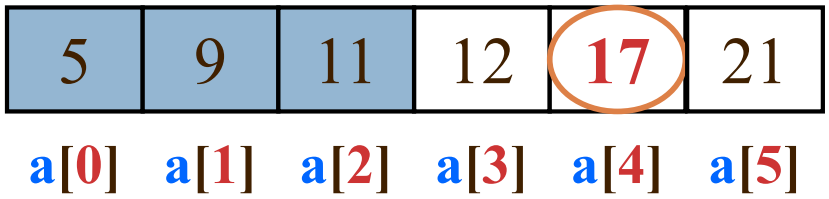
- Il problema di individuare la posizione di un elemento che contenga un particolare valore all'interno di un array può essere affrontato in modo *più efficiente se l'array è ordinato*



- Cerchiamo il valore **12**
- Confrontiamo **12** con il valore che si trova (**circa**) al centro dell'array, ad esempio `a[2]`, che è **11**. Il valore che cerchiamo è maggiore di **11**
  - *se 12 è presente nell'array, sarà a destra di 11*

Ricerca in un array ordinato

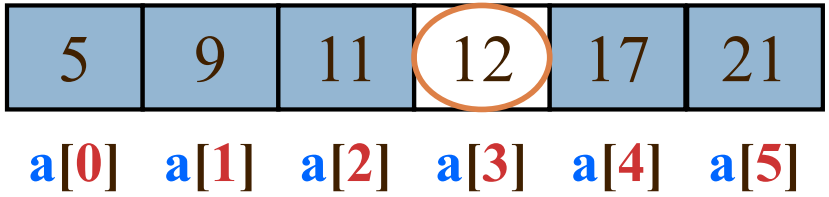
La porzione scura non verrà più esaminata



- A questo punto dobbiamo cercare il valore **12** nel solo sotto-array che si trova a destra di `a[2]`
  - **Il sotto-array che contiene le celle in posizione 0, 1 e 2 non verrà più preso in esame, perché sicuramente NON contiene il valore cercato**
- Usiamo lo stesso algoritmo, confrontando **12** con il valore che si trova al centro, `a[4]`, che è **17**
- Il valore che cerchiamo è minore di **17**
  - ***se è presente nell'array, sarà a sinistra di 17***

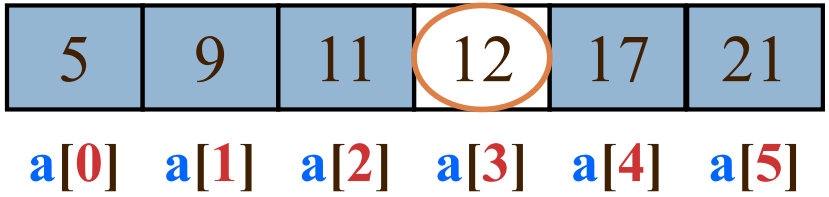
Ricerca in un array ordinato

La porzione scura non verrà più esaminata



- **Il sotto-array che contiene le celle in posizione 4 e 5 non verrà più preso in esame**
- A questo punto dobbiamo cercare il valore **12** nel sotto-array composto dal solo elemento `a[3]`
- Usiamo lo stesso algoritmo, confrontando **12** con il valore che si trova al centro, `a[3]`, che è `12`
- Il valore che cerchiamo è uguale a `12`
  - ***Il valore che cerchiamo è presente nell'array e si trova in posizione 3***

Ricerca in un array ordinato



- Se il valore da cercare fosse stato 13, dopo aver eseguito gli stessi passi, il confronto con l'elemento `a[3]` avrebbe dato esito negativo e avremmo cercato nel sotto-array ***di lunghezza zero*** a destra della posizione 3, concludendo che il valore cercato non è presente nell'array
- Questo algoritmo si chiama ***ricerca per bisezione*** (o ricerca binaria), perché ad ogni passo divide l'array ***in due parti aventi circa le stesse dimensioni*** e ***funziona soltanto se l'array è ordinato***

DIPARTIMENTO

Algoritmo di ricerca binaria

```
public class ArrayUtils
{  public static int binarySearch(int[] a, int target)
   {  return binSearch(a, 0, a.length - 1, target);
   }

private static int binSearch(int[] a,int from,int to,int v)
{
   if (from > to)
      return -1; // NON TROVATO, Caso base

   int mid = (from + to) / 2; // CIRCA IN MEZZO
   int middle = a[mid];
   if (middle == v)
      return mid; // TROVATO, Caso base
   else if (middle < v)
         // CERCA A DESTRA
         return binSearch(a, mid + 1, to, v);
   else
         // CERCA A SINISTRA
         return binSearch(a, from, mid - 1, v);
}
}
```

Ricerca binaria: prestazioni

- Valutiamo le prestazioni dell'algoritmo di ricerca binaria in un array ordinato
- osserviamo che **l'algoritmo è ricorsivo**
- Per cercare in un array di dimensione **n** bisogna
  - effettuare un confronto (con l'elemento centrale)
  - effettuare una ricerca in un array di dimensione n/2
- Quindi

$$T(n) = T(n/2) + 1$$

Ricerca binaria: prestazioni

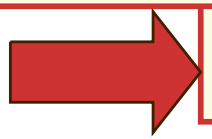
$$T(n) = T(n/2) + 1 = (T(n/4) + 1) + 1 =$$

$$= ((T(n/8)+1) + 1) + 1$$

$$= \ldots = T(n/2^k) + k$$

$$= T(n/2^{\log_2 n}) + \log_2 n$$

$$= T(1) + \log_2 n$$



$O(\log n)$

Algoritmo di ricerca binaria iterativo

```
public static int binSearch(int[] a, int target)
{  if (a == null) return -1;
   int from = 0; // from incluso, to escluso
   int to = a.length;
   // from == to equivale ad array di lunghezza zero
   while (from < to)
   {  int mid = (from + to) / 2;
      int midVal = a[mid];
      if (midVal == target)
         return mid;
      else if (midVal < target)
         from = mid + 1; // compreso
      else
         to = mid; // escluso
   }
   return -1; // la zona di ricerca
}             // ha lunghezza zero
// il comportamento di questo metodo è identico a
// quello di java.util.Arrays.binarySearch
```

Take home message

- Per cercare un valore in un array **non ordinato** posso solo fare una scansione degli elementi, che nel caso peggiore porterà alla scansione di tutto l'array
  - Ricerca sequenziale, complessità O(n)

- Per cercare un valore in un array ordinato posso utilizzare la ricerca binaria, che è molto più veloce nel caso peggiore
  - Ricerca binaria, complessità O(log)

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Riassunto del riassunto: cosa dovete sapere

- Conoscere e capire il concetto di complessità asintotica, in particolare della notazione O-grande.
  - Calcolare la complessità asintotica di piccoli pezzi di codice
  - Data una espressione in n saper dire a quale O-grande appartiene: $34n + 200 + n\log n + 3n^2 = O(n^2)$
  - Confrontare due espressioni in n a livello asintotico:
    - $f(n) = 100000n$ vs $g(n) = 3n^2$: g(n) è peggio!
  - Come funzionano e come si implementano SelectionSort, InsertionSort e MergeSort
  - Quali sono le complessità asintotiche nei casi migliore, peggiore e medio dei tre algoritmi (no le dimostrazioni)