algoritmi e strutture di dati

ricorsione, iterazione e complessità

m.patrignani

nota di copyright

- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

sommario

- ricorsione ed iterazione
- formule di ricorrenza
  - teorema dell'esperto
- strategie algoritmiche
  - algoritmi greedy
  - algoritmi divide et impera
- richiami su algoritmi di ordinamento
  - selection sort
  - merge sort

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	3
variabile sum	0
variabile i	0

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(0)

istruzione	5
variabile f	1
variabile i	

SUM-OF-FACT(3)

istruzione	3
variabile sum	0
variabile i	0

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(0)

~~istruzione~~	~~8~~
~~variabile f~~	~~1~~
~~variabile i~~	~~2~~

SUM-OF-FACT(3)

istruzione	3
variabile sum	0
variabile i	0

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	3
variabile sum	1
variabile i	0

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	2
variabile sum	1
variabile i	1

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(1)

istruzione	8
variabile f	1
variabile i	2

SUM-OF-FACT(3)

istruzione	3
variabile sum	1
variabile i	1

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	3
variabile sum	2
variabile i	1

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	2
variabile sum	2
variabile i	2

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(2)

~~istruzione~~	~~8~~
~~variabile f~~	~~2~~
~~variabile i~~	~~3~~

SUM-OF-FACT(3)

istruzione	3
variabile sum	2
variabile i	2

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	3
variabile sum	4
variabile i	2

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	2
variabile sum	4
variabile i	3

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(3)

~~istruzione~~	~~8~~
~~variabile f~~	6
~~variabile i~~	~~4~~

SUM-OF-FACT(3)

istruzione	3
variabile sum	4
variabile i	3

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	3
variabile sum	10
variabile i	3

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

istruzione	2
variabile sum	10
variabile i	4

effetti di una chiamata a funzione

```
SUM-OF-FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM-OF-FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM-OF-FACT(3)

~~istruzione~~	~~4~~
~~variabile sum~~	~~10~~
~~variabile i~~	~~4~~

funzioni ricorsive

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- abbiamo già visto che l'algoritmo iterativo FACT per il calcolo del fattoriale ha complessità $\Theta(n)$
- il calcolo del fattoriale può essere facilmente realizzato anche tramite un algoritmo ricorsivo

esecuzione di funzioni ricorsive

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire FACT-RIC(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT-RIC(3)

istruzione	4
variabile f	0

esecuzione di funzioni ricorsive

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire FACT-RIC(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT-RIC(2)

istruzione	4
variabile f	0

FACT-RIC(3)

istruzione	4
variabile f	0

esecuzione di funzioni ricorsive

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire FACT-RIC(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT-RIC(1)

istruzione	4
variabile f	0

FACT-RIC(2)

istruzione	4
variabile f	0

FACT-RIC(3)

istruzione	4
variabile f	0

esecuzione di funzioni ricorsive

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire FACT-RIC(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT-RIC(0)

istruzione	2
variabile f	1

FACT-RIC(1)

istruzione	4
variabile f	0

FACT-RIC(2)

istruzione	4
variabile f	0

FACT-RIC(3)

istruzione	4
variabile f	0

esecuzione di funzioni ricorsive

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire FACT-RIC(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT-RIC(1)

istruzione	4
variabile f	1

FACT-RIC(2)

istruzione	4
variabile f	0

FACT-RIC(3)

istruzione	4
variabile f	0

esecuzione di funzioni ricorsive

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire FACT-RIC(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT-RIC(2)

~~istruzione~~	~~4~~
~~variabile f~~	~~2~~

FACT-RIC(3)

istruzione	4
variabile f	0

esecuzione di funzioni ricorsive

```
FACT-RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire FACT-RIC(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT-RIC(3)

~~istruzione~~	~~4~~
~~variabile f~~	~~6~~

costo di FACT-RIC

```
FACT-RIC(n)
1. if n == 0
2.     f = 1
3. else
4.     f = n * FACT-RIC(n-1)
5. return f
```

$T(0) = \Theta(1)$

$T(n) = T(n\text{-}1) + \Theta(1)$

- il costo di FACT-RIC(n) è
  - $\Theta(1)$ quando n è zero
  - pari al costo di FACT-RIC(n-1) + $\Theta(1)$ negli altri casi

formule di ricorrenza

- equazioni o disequazioni che descrivono una funzione in termini del suo valore su input più piccoli
  - prevedono sempre dei casi base e dei casi induttivi
- esempi

$$T(n) = \begin{cases} a & \text{per } n = 0 \\ T(n\text{-}1) + g(n) & \text{per } n > 0 \end{cases}$$

$$T(n) = \begin{cases} a & \text{per } n = 0 \text{ o } n = 1 \\ 2T(n/2) + f(n) & \text{per } n > 1 \end{cases}$$

formule di ricorrenza

- le soluzioni delle formule di ricorrenza non sempre sono facili da trovare
- quando esprimono delle complessità asintotiche talvolta i casi base vengono omessi
  - se $T(n)$ esprime il tempo di esecuzione di un algoritmo, $T(n)$ è sempre $\Theta(1)$ per n piccolo
- esempio

  $T(n) = 2T(n/2) + \Theta(n)$
  - è sottointeso che $T(n) = \Theta(1)$ per n = 0 e n = 1

soluzione di una equazione di ricorrenza

- dimostriamo che l'equazione di ricorrenza

$$T(n) = \begin{cases} a & \text{per } n = 0 \\ T(n\text{-}1) + g(n) & \text{per } n > 0 \end{cases}$$

- ammette come soluzione

$$T(n) = a + \sum_{k=1}^{n} g(k)$$

- per dimostrarlo sostituiamo la soluzione proposta a destra e sinistra dell'equazione di ricorrenza

verifica della correttezza della soluzione

- caso base per n=0

  $T(n=0) = a + \sum_{k=1}^{0} g(k) = a + 0 = a$ (verificato)

- caso induttivo

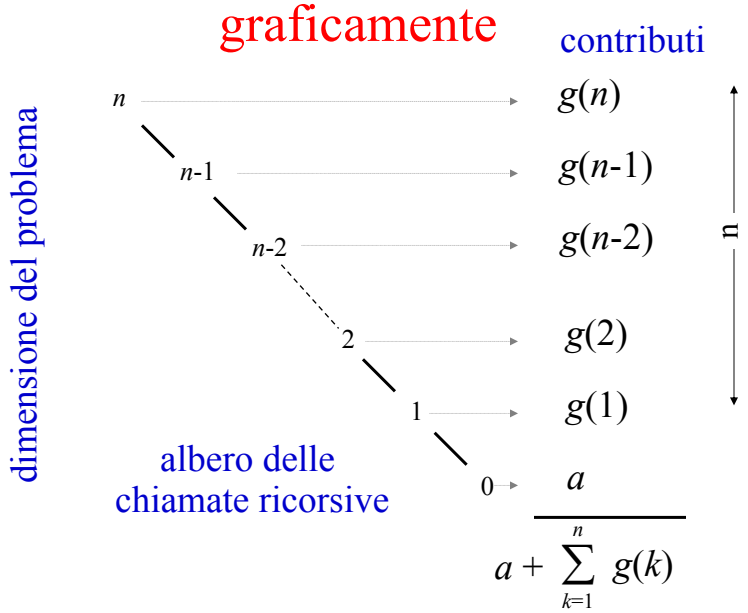
  so che $T(n\text{-}1) = a + \sum_{k=1}^{n-1} g(k)$ (ipotesi induttiva)

  $T(n) = T(n\text{-}1) + g(n)$ (dalla definizione)

  $T(n) = a + \sum_{k=1}^{n-1} g(k) + g(n)$

  $T(n) = a + \sum_{k=1}^{n} g(k)$ (verificato)

graficamente



complessità di FACT-RIC

- sappiamo che FACT-RIC ha complessità

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \\ T(n\text{-}1) + \Theta(1) & \text{per } n > 0 \end{cases}$$

- sappiamo che l'equazione di ricorrenza

$$T(n) = \begin{cases} a & \text{per } n = 0 \\ T(n\text{-}1) + g(n) & \text{per } n > 0 \end{cases}$$

- ammette come soluzione $T(n) = a + \sum_{k=1}^{n} g(k)$
- la complessità di FACT-RIC è dunque

$$T(n) = \Theta(1) + \sum_{k=1}^{n} \Theta(1) = \Theta(n)$$

algoritmi ricorsivi per l'ordinamento

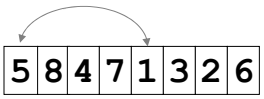
- richiameremo due algoritmi di ordinamento
  - selection sort
  - merge sort
- calcoleremo la loro complessità tramite delle equazioni di ricorrenza
- ciò ci consentirà di considerare due tecniche algoritmiche diverse
  - tecnica greedy
  - tecnica divide et impera

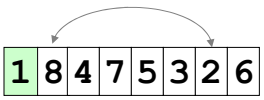
gli algoritmi greedy

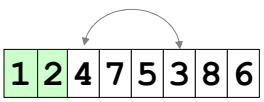
- gli algoritmi greedy (golosi) costruiscono una soluzione scegliendo sempre l'alternativa che al momento sembra più appetibile

algoritmo selection sort

- utilizza una tecnica greedy per ordinare un array
- strategia generale
  - seleziona l'elemento più piccolo e mettilo al primo posto
  - seleziona l'elemento più piccolo dei rimanenti e mettilo al secondo posto
  - …







versione iterativa SELECTION-ITER

```
SELECTION-ITER(A)
1.  for i = 0 to A.length-2
2.      min = i          ▷ indice elemento minimo in A[i..n-1]
3.      for j = i + 1 to A.length-1    ▷ scorro l'array
4.          if A[j] < A[min]           ▷ devo aggiornare min
5.          then   min = j
6.      temp = A[i]                    ▷ scambio A[i] con A[min]
7.      A[i] = A[min]
8.      A[min] = temp
```

- i valori dell'input non modificano il numero delle iterazioni del ciclo esterno e del ciclo interno
  - quindi il caso migliore, il caso peggiore ed il caso medio hanno la stessa complessità

complessità del SELECTION-ITER

```
SELECTION-ITER(A)
1.  for i = 0 to A.length-2
2.      min = i          ▷ indice elemento minimo in A[i..n-1]
3.      for j = i + 1 to A.length-1    ▷ scorro l'array
4.          if A[j] < A[min]           ▷ devo aggiornare min
5.          then   min = j
6.      temp = A[i]                    ▷ scambio A[i] con A[min]
7.      A[i] = A[min]
8.      A[min] = temp
```

- l'algoritmo esegue $O(n)$ cicli esterni e $O(n)$ cicli interni
  - dunque SELECTION-ITER ha complessità $O(n^2)$
- la riga 4 viene eseguita $(n-1)+(n-2)+...+1 = [n(n-1)]/2$ volte
  - dunque SELECTION-ITER ha complessità $\Omega(n^2)$
- il tempo di esecuzione dell'algoritmo è $\Theta(n^2)$

versione ricorsiva SELECTION-RIC

```
SELECTION(A)                    ▷ ordina A
1.  SELECTION-RIC(A,0)
```

```
SELECTION-RIC(A,i)              ▷ ordina A da i a A.length-1
1.  if i < A.length-1           ▷ altrimenti è già ordinato
2.      min = i                 ▷ indice elemento minimo in A[i..n-1]
3.      for j = i + 1 to A.length-1     ▷ scorro l'array
4.          if A[j] < A[min]            ▷ devo aggiornare min
5.          then   min = j
6.      temp = A[i]                     ▷ scambio A[i] con A[min]
7.      A[i] = A[min]
8.      A[min] = temp
9.      SELECTION-RIC(A,i+1)
```


complessità di SELECTION-RIC

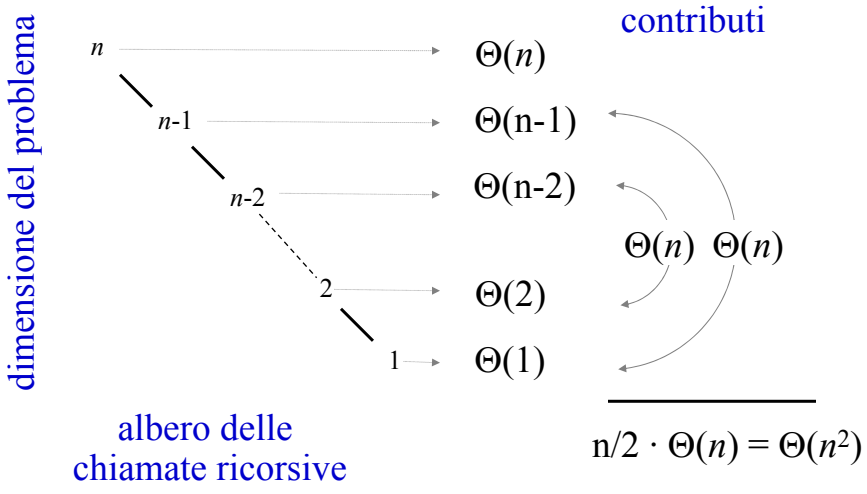
- possiamo scrivere la seguente equazione di ricorrenza, in cui n è il numero degli elementi di A ancora da ordinare

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 1 \\ T(n\text{-}1) + \Theta(n) & \text{per } n > 0 \end{cases}$$

- la complessità di SELECTION-RIC è dunque

$$T(n) = \Theta(1) + \sum_{k=1}^{n} \Theta(k) = \Theta(n^2)$$

graficamente



la tecnica divide et impera

- detta anche “divide and conquer”
- consiste nel suddividere il problema in diversi sottoproblemi
  - i sottoproblemi sono dello stesso tipo del problema originale
    - ma di dimensioni più piccole
  - i sottoproblemi possono essere risolti in maniera ricorsiva
    - suddividendoli a loro volta
  - caso base
    - quando i sottoproblemi sono di dimensioni ridottissime la loro soluzione è banale

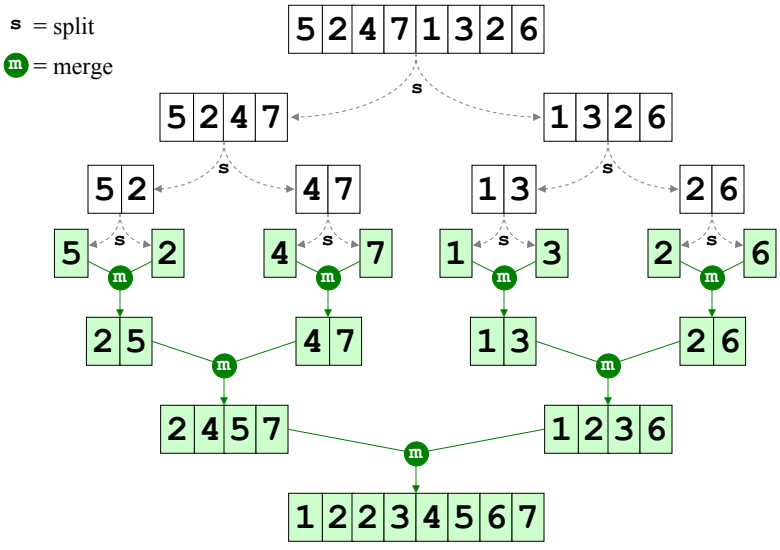
ricorsione del divide et impera

- a ciascun passo della ricorsione
  - divide
    - l'istanza corrente viene divisa in due o più istanze più piccole
  - impera
    - l'algoritmo viene lanciato sulle istanze più piccole
  - combina
    - le soluzioni delle istanze più piccole vengono utilizzate per produrre una soluzione dell'istanza corrente

merge sort

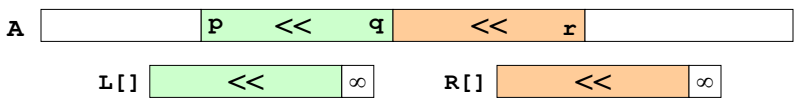
- osservazione elementare
  - due sequenze ordinate possono essere fuse in un'unica sequenza ordinata molto facilmente
- un possibile algoritmo
  - dividere la sequenza di input in due sottosequenze
  - ordinare le due sottosequenze
    - tramite lo stesso merge sort
  - fondere le due sottosequenze ordinate
- caso base
  - un array di un solo elemento è ordinato per definizione

merge sort con input 5 2 4 7 1 3 2 6



fusione: l'algoritmo MERGE

```
MERGE(A,p,q,r)
1.  n1 = q - p + 1          ▷ lunghezza della prima sequenza
2.  n2 = r - q              ▷ lunghezza della seconda sequenza
3.      ▷ creo array L[0…n1] e R[0…n2] (con una casella in +)
4.  for i = 0 to n1-1
5.      L[i] = A[p+i]       ▷ copio la 1ª sequenza
6.  for j = 0 to n2-1
7.      R[j] = A[q+j+1]     ▷ copio la 2ª sequenza
8.  L[n1] = ∞               ▷ chiudo con "infinito"
9.  R[n2] = ∞               ▷ chiudo con "infinito"
                …(continua nella prossima slide)…
```



fusione (continua)

```
                    …(dalla slide precedente)…
10. i = 0           ▷ iteratore per array L
11. j = 0           ▷ iteratore per array R
12. for k = p to r
13.     if L[i] ≤ R[j] then
14.         A[k] = L[i]       ▷ pesco da L
15.         i = i + 1
16.     else
17.         A[k] = R[j]       ▷ pesco da R
18.         j = j + 1
```

- il confronto con "≤" sulla riga 13 garantisce la stabilità dell'algoritmo
  - se `L[i]=R[j]` allora `L[i]` ha la precedenza

l'algoritmo MERGE-SORT

- l'algoritmo `MERGE-SORT` esegue la parte "divide", risolve i sottoproblemi ed esegue la parte "combine"

```
MERGE-SORT(A,p,r)
1.  if p < r then              ▷ nel caso base esco subito
2.      q = ⌊(p+r)/2⌋          ▷ divido l'array in due
3.      MERGE-SORT(A,p,q)
4.      MERGE-SORT(A,q+1,r)
5.      MERGE(A,p,q,r)
```

- all'inizio della computazione lanciamo

```
MERGE(A)                ▷ ordina A
1.  MERGE-SORT(A,0,A.length-1)
```


tempo di esecuzione di merge sort

- calcoliamo il costo $T(n)$ di esecuzione del merge sort su un'istanza con n elementi
  - per comodità assumiamo che n sia una potenza di 2, in modo che la divisione produca sempre sottoarray con lo stesso numero di elementi
- caso base
  - costo $\Theta(1)$
- divide
  - calcolo di $n/2$: costo $D(n) = \Theta(1)$
- impera
  - ogni sottoproblema ha dimensione $n/2$
  - i sottoproblemi sono 2
  - costo: $2 \cdot T(n/2)$
- combina
  - l'algoritmo `MERGE` ha costo lineare: $C(n) = \Theta(n)$

tempo di esecuzione di merge sort

- complessivamente

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \text{ o } n = 1 \\ 2 \cdot T(n/2) + D(n) + C(n) & \text{per } n > 1 \end{cases}$$

- poiché $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$ si ha

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \text{ o } n = 1 \\ 2 \cdot T(n/2) + \Theta(n) & \text{per } n > 1 \end{cases}$$

- dimostreremo che questa particolare equazione di ricorrenza ammette come soluzione

$$T(n) = \Theta(n \log n)$$

master theorem

- siano $a, b \geq 1$ e sia $p(n^k)$ un polinomio di grado k
- il master theorem considera l'equazione di ricorrenza seguente

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \\ a \cdot T(n/b) + p(n^k) & \text{per } n > 0 \end{cases}$$

- si dimostra (noi non lo dimostriamo) che tale equazione di ricorrenza ammette le soluzioni seguenti
  - se $a < b^k$ allora $T(n) = \Theta(n^k)$
  - se $a = b^k$ allora $T(n) = \Theta(n^k \log n)$
  - se $a > b^k$ allora $T(n) = \Theta(n^{\log_b a})$

esempi di applicazione del master theorem

- $T(n) = 9T(n/3) + n$
  - abbiamo: $a = 9$; $b = 3$; $p(n^k) = n$; $k = 1$
  - quindi $a > b^k$
  - si ha $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 9}) = \Theta(n^2)$
- $T(n) = T(2n/3) + 1$
  - abbiamo: $a = 1$; $b = 3/2$; $p(n^k) = 1$; $k = 0$
  - quindi $a = b^k$
  - si ha $T(n) = \Theta(n^k \log n) = \Theta(n^0 \log n) = \Theta(\log n)$

complessità del merge sort

- la complessità del merge sort è data dalla formula di ricorrenza

  $T(n) = 2 \cdot T(n/2) + \Theta(n)$

- applichiamo il teorema dell'esperto
  - abbiamo: $a = 2$; $b = 2$; $p(n^k) = n$; $k = 1$
  - quindi $a = b^k$
  - si ha $T(n) = \Theta(n^k \log n) = \Theta(n \log n)$

graficamente