# Fondamenti di Informatica (Elettronici)

merge-sort e quicksort– Lezione 33

3 gennaio 2021

# Algoritmi fondamentali: merge-sort e quicksort

Section 1

# Algoritmo di ordinamento per fusione (merge sort)

# Algoritmo di fusione

Normalmente una procedura di fusione prevede due collezioni ordinate di ingresso e una collezione di uscita.

Nel seguito ne daremo una versione leggermente diversa, necessaria pero' per l'algoritmo MergeSort.

In particolare si supporra' che un array di nome `A` sia una variabile globale, e che due suoi sottoinsiemi adiacenti di indici iniziali e finali `(i:k)` e `(k+1:j)` debbano essere fusi restituendo ordinato il sottoinsieme di componenti indicizzate in `(i:j)`

A questo scopo si userà un array locale `out` sulle cui copie ricorsive si eseguira' la fusione, che sara' infine ricopiata nell'output array `out`.

# ordinamento per fusione

Merge sort Algoritmo ricorsivo

CASO BASE :

CASO RICORSIVO :

# ordinamento per fusione

Merge sort Algoritmo ricorsivo

CASO BASE :

- Se l'array contiene due elementi allora ordina direttamente

CASO RICORSIVO :

# ordinamento per fusione

Merge sort Algoritmo ricorsivo

CASO BASE :

- Se l'array contiene due elementi allora ordina direttamente

CASO RICORSIVO :

- Dividi l'array in due array di lunghezza simile

# ordinamento per fusione

Merge sort Algoritmo ricorsivo

CASO BASE :

- Se l'array contiene due elementi allora ordina direttamente

CASO RICORSIVO :

- Dividi l'array in due array di lunghezza simile
- ordina entrambi i due array parziali non ordinati

# ordinamento per fusione

Merge sort Algoritmo ricorsivo

CASO BASE :

- Se l'array contiene due elementi allora ordina direttamente

CASO RICORSIVO :

- Dividi l'array in due array di lunghezza simile
- ordina entrambi i due array parziali non ordinati
- fondi i due array parziali ordinati

Section 2

# merge sort: Prima versione

# Merge-Sort: Prima versione

Si utilizza una ricorsione indiretta, con una funzione `mergesort` non ricorsiva che chiama una funzione `mSort` ricorsiva.

Funzioni ricorsive definiscono sul record di attivazione le variabili locali e i parametri formali.

```
function mergesort(T)
    msort(T, 1, m)
end
```

# Merge-Sort: pseudocode 1/2

```
# ordina le comp. di T comprese tra i e j
function msort(T, i, j)
    # caso base
    if # T contiene solo una componente#
        # return T
    else
    # caso ricorsivo
        # dividi (i:j) in due sottointerv. (i:mid) e (mid+1:j)
        # msort( sottointervallo (i:mid) )
        # msort( sottointervallo (mid+1:j) )
        # fondi (merge) i due sottointervalli ordinati
    end
    return il risultato della fusione
end

function mergesort(T)
    m = length(T)
    T = msort(T, 1, m)
    return T
end
```

# Merge-Sort: Versione finale

```
function mergesort( A::Array, debug=false )
  if length(A) <= 1 return A end      # caso base
  mid = div(length(A), 2)             # caso ricorsivo
  lpart = mergesort( A[1:mid], debug );      if debug  @show lpart end
  rpart = mergesort( A[mid+1:end], debug );   if debug @show rpart end
  out = similar(A)                    # algoritmo di merge
  i = ri = li = 1
  while li <= length(lpart) && ri <= length(rpart)
    if lpart[li] <= rpart[ri]
      out[i] = lpart[li]; li += 1
    else
      out[i] = rpart[ri]; ri += 1
    end
    i += 1
  end
  if li <= length(lpart)
    copyto!(out, i, lpart, li);       if debug @show out end
  else
    copyto!(out, i, rpart, ri);       if debug @show out end
  end
  return out
end
```

Section 3

# merge sort: esempio

# Merge-Sort: Esempio

```
julia> v = rand(-10:10, 10);

julia> println("# unordered: $v\n -> ordered: ", mergesort(v))
# unordered: [6, -2, 8, -10, -2, -3, -6, -6, -2, -3]
 -> ordered: [-10, -6, -6, -3, -3, -2, -2, -2, 6, 8]

julia> v = rand(-10:10, 10);

julia> println("# unordered: $v\n -> ordered: ", mergesort(v))
# unordered: [1, -7, 5, -6, 5, -2, -4, 10, -6, 2]
 -> ordered: [-7, -6, -6, -4, -2, 1, 2, 5, 5, 10]

julia> v = rand(-10:10, 10);

julia> println("# unordered: $v\n -> ordered: ", mergesort(v))
# unordered: [6, -6, -8, 1, -7, -7, -6, -10, 9, 0]
 -> ordered: [-10, -8, -7, -7, -6, -6, 0, 1, 6, 9]
```

# Merge-Sort: Debug print (I)

```
julia> v = rand(-10:10, 10);

julia> println("# unordered: $v\n -> ordered: ", mergesort(v))
# unordered: [-10, 10, 6, -2, 8, 6, -7, 1, 10, 0]
 -> ordered: [-10, -7, -2, 0, 1, 6, 6, 8, 10, 10]

julia> println("# unordered: $v\n -> ordered: ", mergesort(v, true))
lpart = [-10]
rpart = [10]
out = [-10, 10]
lpart = [-10, 10]
lpart = [6]
lpart = [-2]
rpart = [8]
out = [-2, 8]
rpart = [-2, 8]
out = [-2, 6, 8]
rpart = [-2, 6, 8]
out = [-10, -2, 6, 8, 10]
lpart = [-10, -2, 6, 8, 10]
```

# Merge-Sort: Debug print (I)

```
lpart = [6]
rpart = [-7]
out = [-7, 6]
lpart = [-7, 6]
lpart = [1]
lpart = [10]
rpart = [0]
out = [0, 10]
rpart = [0, 10]
out = [0, 1, 10]
rpart = [0, 1, 10]
out = [-7, 0, 1, 6, 10]
rpart = [-7, 0, 1, 6, 10]
out = [-10, -7, -2, 0, 1, 6, 6, 8, 10, 10]
# unordered: [-10, 10, 6, -2, 8, 6, -7, 1, 10, 0]
 -> ordered: [-10, -7, -2, 0, 1, 6, 6, 8, 10, 10]
```

Section 4

# merge sort: complessità

# Analisi di complessità: Esempio di esecuzione 1/2

chiamate ricorsive nel sottoinsieme di destra

```
1  2  3  4  5 6  7 8
23 12 2 18 24 9 32 5

mSort (1 , 8)

      mSort(1,4)
mSort(1,2) mSort(3,4)

1  2  3  4  5 6  7 8
12 23 2 18 24 9 32 5

merge(1,2,4)

1  2  3  4  5 6  7 8
2 12 18 23 24 9 32 5
```

# Analisi di complessità: Esempio di esecuzione 2/2

chiamate ricorsive nel sottoinsieme di sinistra

```
      mSort(5,8)
mSort (5,6) mSort (7,8)
```

```
1  2  3  4 5 6  7  8
2 12 18 23 9 24 5 32
```

```
merge (5,6,8)
```

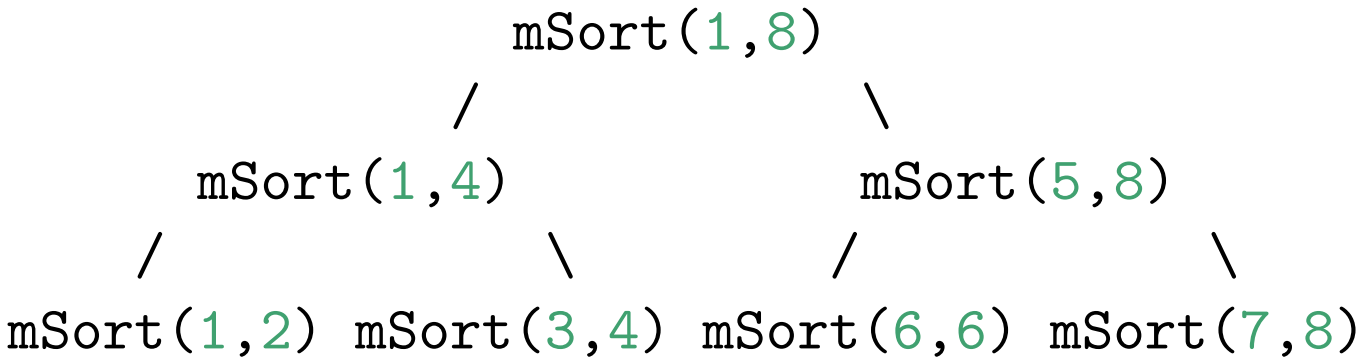
```
1  2  3  4 5 6  7  8
2 12 18 23 5 9 24 32
```

```
merge (1,4,8)
```

```
1 2 3  4  5  6  7  8
2 5 9 12 18 23 24 32
```

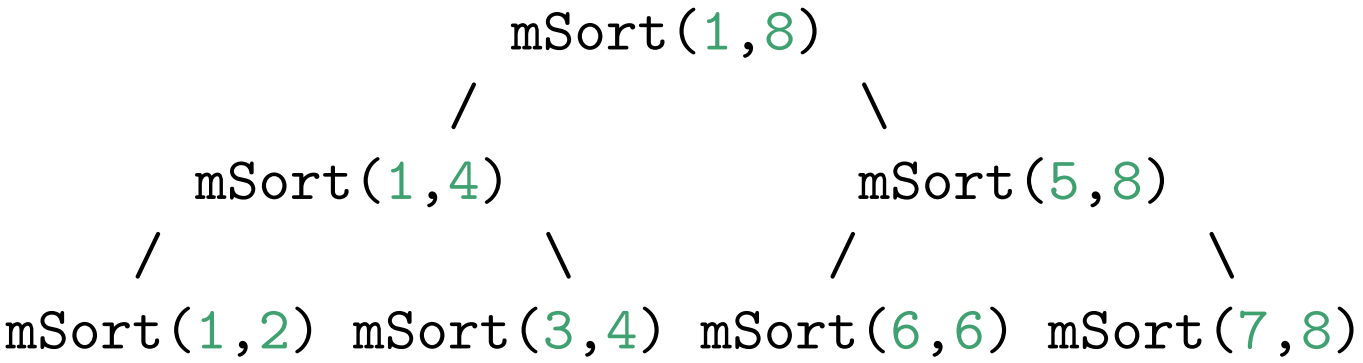
# Analisi di complessità

Occorre contare il numero di attivazioni della procedura ricorsiva `msort`



# Analisi di complessità

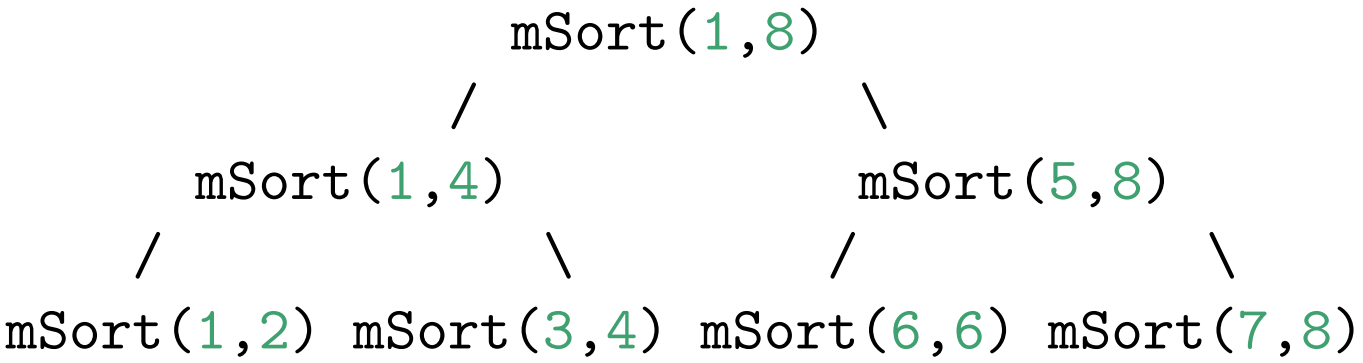
Occorre contare il numero di attivazioni della procedura ricorsiva `msort`




- Su $n$ elementi abbiamo $n - 1$ attivazioni

# Analisi di complessità

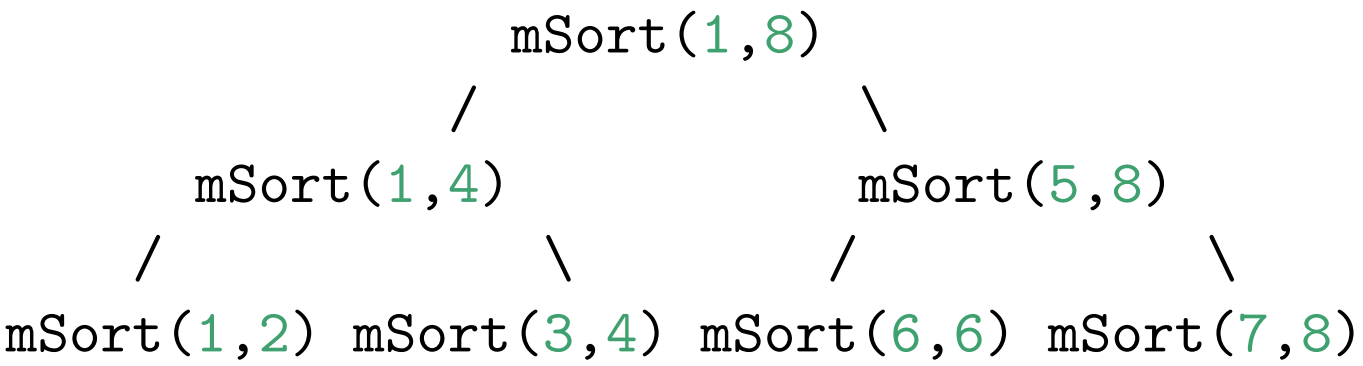
Occorre contare il numero di attivazioni della procedura ricorsiva `msort`




- Su $n$ elementi abbiamo $n-1$ attivazioni
- Su ogni livello il costo complessivo delle fusioni è $O(n)$.

# Analisi di complessità

Occorre contare il numero di attivazioni della procedura ricorsiva `msort`




- Su $n$ elementi abbiamo $n-1$ attivazioni
- Su ogni livello il costo complessivo delle fusioni è $O(n)$.
- Per il numero $k = \log n$ dei livelli, si avra' un costo complessivo delle fusioni:

$$O(n \log n)$$

.

Section 5

# Algoritmo Quicksort

# Algoritmo Quicksort

Pseudo-codice :

Quicksort, noto anche come ordinamento con scambio di partizioni, utilizza questo metodo:

1. Scegli qualsiasi elemento dell'array come perno (pivot).

# Algoritmo Quicksort

Pseudo-codice :

Quicksort, noto anche come ordinamento con scambio di partizioni, utilizza questo metodo:

1. Scegli qualsiasi elemento dell'array come perno (pivot).
2. Dividi tutti gli altri elementi (eccetto il perno) in due partizioni.

# Algoritmo Quicksort

Pseudo-codice :

Quicksort, noto anche come ordinamento con scambio di partizioni, utilizza questo metodo:

1. Scegli qualsiasi elemento dell'array come perno (pivot).
2. Dividi tutti gli altri elementi (eccetto il perno) in due partizioni.
   a) Tutti gli elementi inferiori al pivot devono trovarsi nella prima partizione.

# Algoritmo Quicksort

Pseudo-codice :

Quicksort, noto anche come ordinamento con scambio di partizioni, utilizza questo metodo:

1. Scegli qualsiasi elemento dell'array come perno (pivot).
2. Dividi tutti gli altri elementi (eccetto il perno) in due partizioni.
   a) Tutti gli elementi inferiori al pivot devono trovarsi nella prima partizione.
   b) Tutti gli elementi maggiori del pivot devono trovarsi nella seconda partizione.

# Algoritmo Quicksort

Pseudo-codice :

Quicksort, noto anche come ordinamento con scambio di partizioni, utilizza questo metodo:

1. Scegli qualsiasi elemento dell'array come perno (pivot).
2. Dividi tutti gli altri elementi (eccetto il perno) in due partizioni.
   a) Tutti gli elementi inferiori al pivot devono trovarsi nella prima partizione.
   b) Tutti gli elementi maggiori del pivot devono trovarsi nella seconda partizione.
3. Usa la ricorsione per ordinare entrambe le partizioni.

# Algoritmo Quicksort

Pseudo-codice :

Quicksort, noto anche come ordinamento con scambio di partizioni, utilizza questo metodo:

1. Scegli qualsiasi elemento dell'array come perno (pivot).
2. Dividi tutti gli altri elementi (eccetto il perno) in due partizioni.
   a) Tutti gli elementi inferiori al pivot devono trovarsi nella prima partizione.
   b) Tutti gli elementi maggiori del pivot devono trovarsi nella seconda partizione.
3. Usa la ricorsione per ordinare entrambe le partizioni.
4. Unisci la prima partizione ordinata, il pivot e la seconda partizione ordinata.

# Quicksort: complessità

Il pivot migliore crea partizioni di uguale lunghezza (o lunghezze diverse di uno).

Il pivot peggiore crea una partizione vuota (ad esempio, se il pivot à il primo o l'ultimo elemento di un array ordinato).

Il tempo di esecuzione di Quicksort varia da

$$O(n \log n)$$

con i migliori pivot, a

$$O(n^2)$$

con i peggiori pivot, dove $n$ è il numero di elementi nell'array.

# Algoritmo Quicksort

Funzione predefinita per l'ordinamento sul posto tramite `quicksort` (il codice dalla libreria standard è abbastanza leggibile :-)

```
sort!(A, alg=QuickSort)
```

Una semplice implementazione polimorfica di un quicksort ricorsivo sul posto (basato sullo pseudocodice sopra) è data nel seguito.

# Algoritmo Quicksort

```
function quicksort!(A,i=1,j=length(A);debug=false)
    if j > i
        pivot = A[rand(i:j)]      # random element of A
        left, right = i, j;                  if debug @show i,pivot,j end
        while left <= right
            while A[left] < pivot
                left += 1
            end
            while A[right] > pivot
                right -= 1
            end;                             if debug @show left,right,A end
            if left <= right
                A[left], A[right] = A[right], A[left]
                left += 1
                right -= 1;                  if debug @show left,right,A end
            end
        end
        quicksort!(A,i,right,debug);     if debug @show A[1:i], i,right end
        quicksort!(A,left,j,debug);      if debug @show A[j:end], left,j end
    end
    return A
end
```

# Quicksort : debug

```
julia> v = rand(-10:10, 10);

julia> w = copy(v);

julia> println("# unordered: $v\n -> ordered: ", quicksort!(v))
# unordered: [9, -4, 2, -3, 8, 1, 6, 10, -6, 2]
 -> ordered: [-6, -4, -3, 1, 2, 2, 6, 8, 9, 10]

julia> println("# unordered: $w\n -> ordered: ", quicksort!(w,debug=true))
(i, pivot, j) = (1, 8, 10)
(left, right, A) = (1, 10, [9, -4, 2, -3, 8, 1, 6, 10, -6, 2])
(left, right, A) = (2, 9, [2, -4, 2, -3, 8, 1, 6, 10, -6, 9])
(left, right, A) = (5, 9, [2, -4, 2, -3, 8, 1, 6, 10, -6, 9])
(left, right, A) = (6, 8, [2, -4, 2, -3, -6, 1, 6, 10, 8, 9])
(left, right, A) = (8, 7, [2, -4, 2, -3, -6, 1, 6, 10, 8, 9])
(i, pivot, j) = (1, -4, 7)
(A[1:i], i, right) = ([-6], 1, 1)
(A[j:end], left, j) = ([6, 10, 8, 9], 3, 7)
(A[1:i], i, right) = ([-6], 1, 7)
(i, pivot, j) = (8, 9, 10)
(A[1:i], i, right) = ([-6, -4, -3, 1, 2, 2, 6, 8], 8, 9)
(A[j:end], left, j) = ([10], 10, 10)
(A[j:end], left, j) = ([10], 8, 10)
# unordered: [9, -4, 2, -3, 8, 1, 6, 10, -6, 2]
 -> ordered: [-6, -4, -3, 1, 2, 2, 6, 8, 9, 10]
```

# Quicksort funzionale (carinissimo)

Su Rosetta Code Tradotto da Haskell

```
qsort(L) =  isempty(L) ?
            L :
            vcat(
              qsort(filter(x -> x < L[1], L[2:end])),
              L[1:1],
              qsort(filter(x -> x >= L[1], L[2:end]))
            )
```

# Quicksort funzionale (carinissimo)

Su Rosetta Code Tradotto da Haskell

```
qsort(L) =  isempty(L) ?
            L :
            vcat(
              qsort(filter(x -> x < L[1], L[2:end])),
              L[1:1],
              qsort(filter(x -> x >= L[1], L[2:end]))
            )
```

È una sola istruzione julia !! L'istruzione condizionale ternaria

# Algoritmo Quicksort

```
julia> A=[84,77,20,60,47,20,18,97,41,49,31,39,73,68,65,52,1,92,15,9]

julia> qsort(A)
[1,9,15,18,20,20,31,39,41,47,49,52,60,65,68,73,77,84,92,97]

julia> quicksort!(copy(A))
[1,9,15,18,20,20,31,39,41,47,49,52,60,65,68,73,77,84,92,97]

julia> qsort(A)==quicksort!(copy(A))==sort(A)==sort(A,alg=QuickSort)
true
```