Un ordinamento con tempo di esecuzione in $\Theta(n \lg n)$

Fondi (merge)

Utilizzeremo una variante della funzione fondi che abbiamo visto

Fondi(L,R,A)

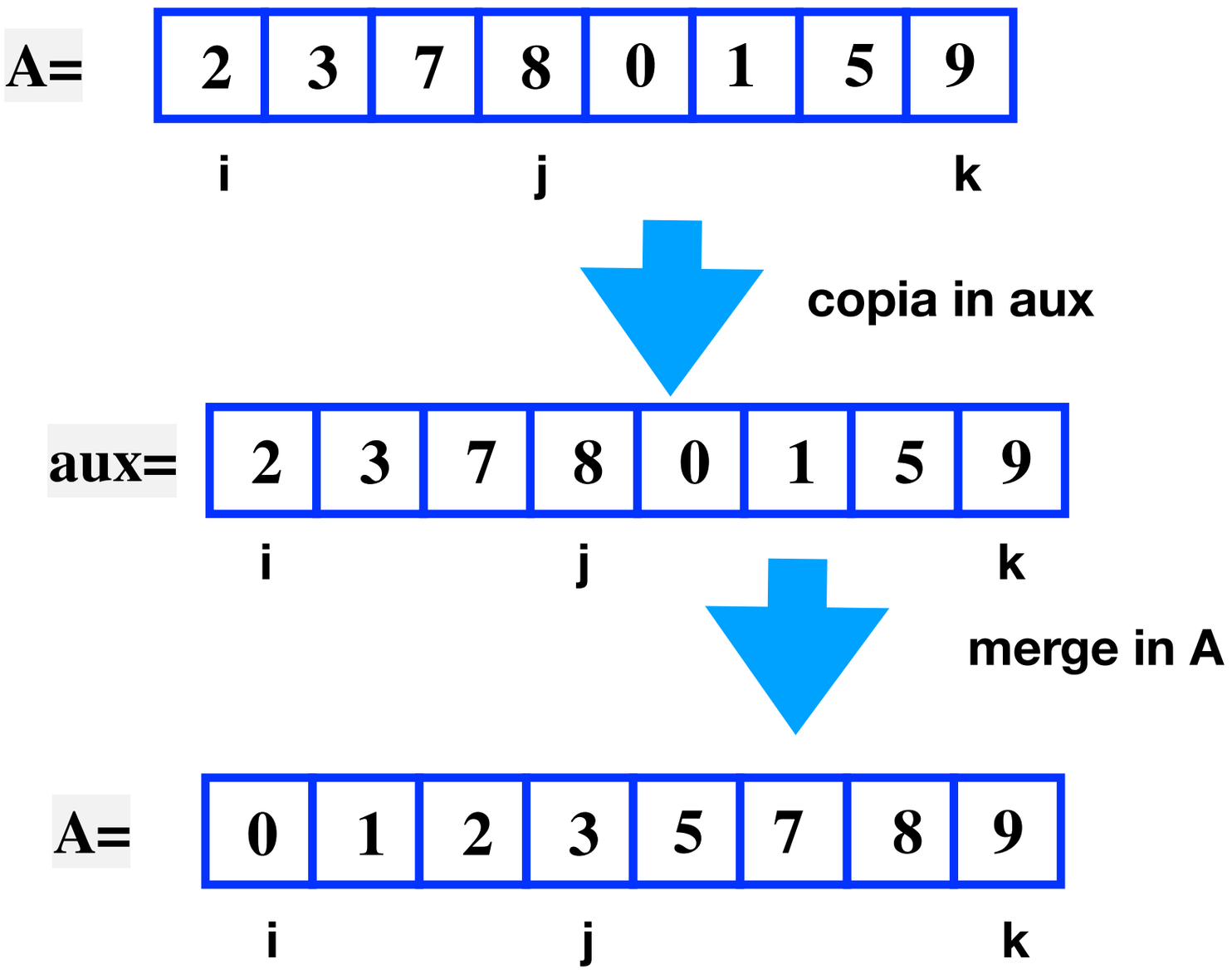
Input: L,R ed A sono array di m, n e n+m elementi
prec: L e R sono ordinati, cioè
L[0] ≤ ...≤ L[m-1] e R[0]≤…≤R[n-1]
postc: A contiene gli elementi di L e di R e
A[0] ≤ ...≤ A[m+n-1]

La differenza sta nel fatto che il nostro array L è la porzione di A delimitata da due indici, i e j, L= A[i..j], e R la porzione dell'array A tra gli indici j+1 e k, A[j+1..k], infine la fusione di queste due parti di A già ordinate avviene in A, utilizzando un array di appoggio per le due porzioni.

Fondi (merge)

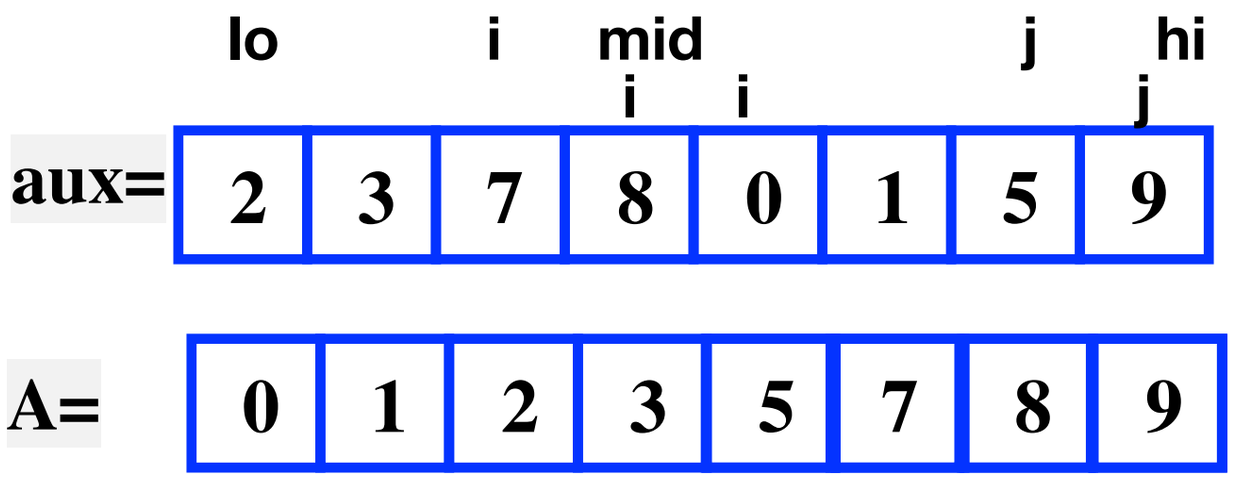
Possiamo copiare prima le due porzioni da fondere in un array ausiliario e scrivere i risultati della fusione in A.

Chiamiamo merge(A,aux,i,j,k) questa funzione che fonde, ricopiandoli da aux in A, gli elementi di A[i..j] e A[j+1..k] da aux.



merge

```
merge(A,aux, lo, mid, hi)
input: due array A e aux, con lo stesso numero di elementi. Tre interi non negativi
prec: 0≤lo≤mid≤hi<A.lenght, A[lo]≤…≤A[mid] e  A[mid+1]≤…≤A[hi]
output: A[lo]≤…≤A[mid]≤ A[mid+1]≤…≤A[hi]
{
  for (k = lo; k ≤ hi; k++) // copia gli elementi da lo a hi in aux
        aux[k] = A[k];
  i = lo, j = mid+1;
  for (k = lo; k ≤ hi; k++)
  { // fusione
     if (i > mid)  then     A[k] = aux[j++] else  //la prima porzione è vuota
     if (j > hi   then     A[k] = aux[i++] else // la seconda porzione è vuota
     if (aux[j] ≤ aux[i])   then     A[k] = aux[j++]
                              else     A[k] = aux[i++]
  }
}
```



mergeSort iterativo

L’idea è:

fondi sotto arrays consecutivi di quello iniziale raddoppiando ad ogni passo la dimensione dei sotto arrays da fondere, fino ad arrivare ad aver ordinato tutto l’array iniziale. La prima dimensione è 1 perchè ogni elemento isolato è un insieme ordinato.

m =merge

dim = 1

merge(A, aux, 0, 0, 1)

merge(A, aux, 2, 2, 3)

merge(A, aux, 4, 4, 5)

merge(A, aux, 6, 6, 7)

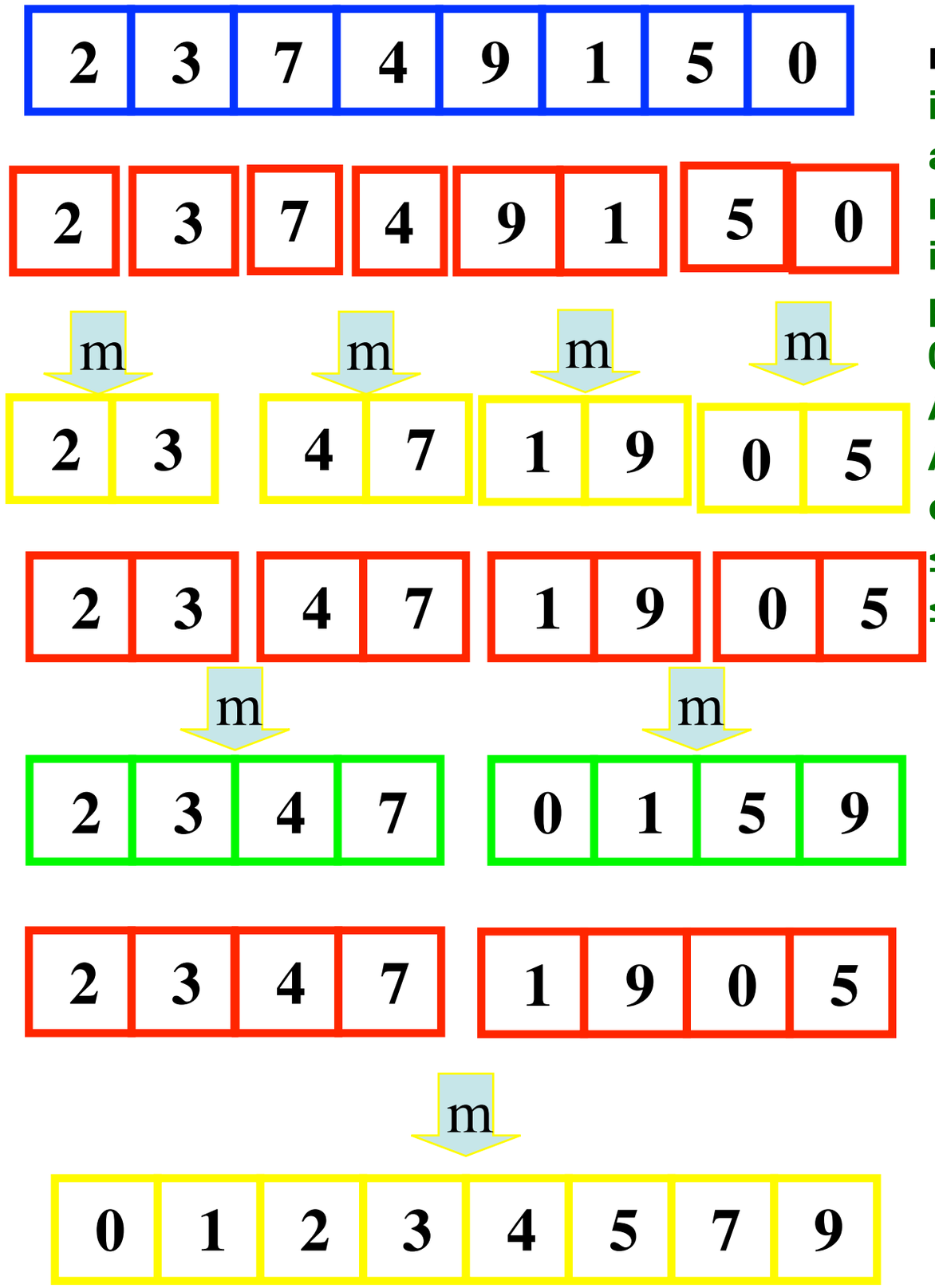
dim = 2

merge(A, aux, 0, 1, 3)

merge(A, aux, 4, 5, 7)

dim = 4

merge(A, aux, 0, 3, 7)



merge(A,aux, lo, mid, hi)
input: due array A e aux, con lo stesso numero di elementi. Tre interi non negativi
prec: 0≤lo≤mid≤hi<A.lenght, A[lo]≤…≤A[mid] e A[mid+1]≤…≤A[hi]
output: A[lo]≤… ≤A[mid]≤ A[mid+1]≤… ≤A[hi]

dim = 1

merge(A, aux, 0, 0, 1)

merge(A, aux, 2, 2, 3)

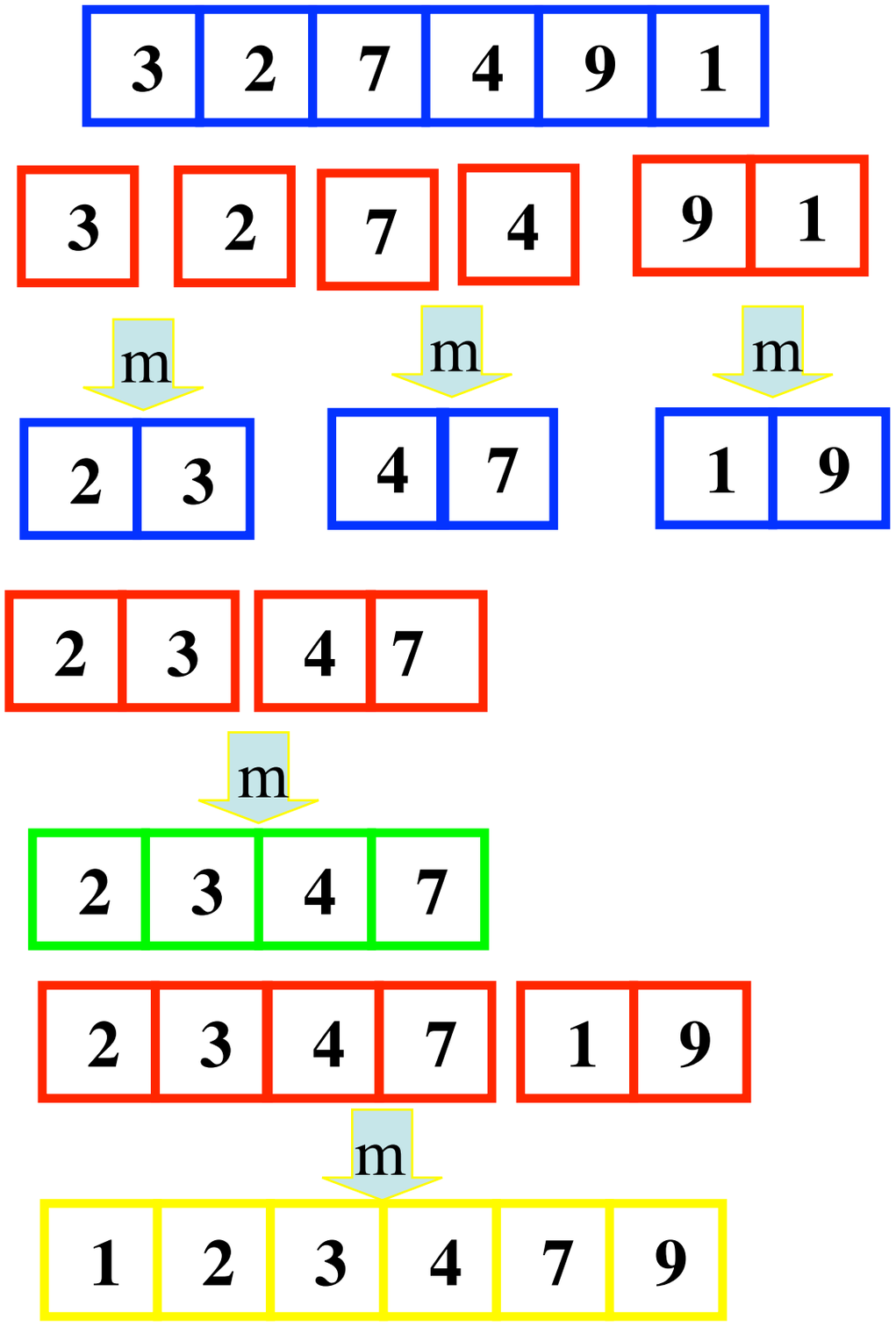
merge(A, aux, 4, 4, 5)

dim = 2

merge(A, aux, 0, 1, 3)

dim = 4

merge(A, aux, 0, 3, 5)



PseudocodiceMergesort Iterativo

```
MergeSortIter(A)
    n = A.length;
    sia aux un nuovo array di n elementi
    for (dim = 1; dim < n; dim = 2dim)
        for (lo = 0; lo < n - dim; lo = lo + 2dim)
            merge(A, aux, lo, lo+dim-1,min(lo+2dim-1, n-1));
```

L’algoritmo non lavora in loco, ma usa un array ausiliario.
Dunque usa una memoria aggiuntiva di dimensione lineare.

Analisi mergesort iterativo

Qual'è il tempo di esecuzione?
Per comodità mettiamoci nel caso che $n = 2^k$

Passo 1: si eseguono $n/2=2^{k-1}$ fusioni su elementi singoli, dim= 1
Passo 2: si eseguono $n/4=2^{k-2}$ fusioni su coppie di elementi, dim= 2
Passo 3: si eseguono $n/8=2^{k-3}$ fusioni su quadruple di elementi, dim = 2^2

…
qual'è l'ultimo passo?

passo k-1: si esegue 1 fusione su due array di 2^{k-1} elementi, dim= 2^{k-1}

Analisi mergesort iterativo

Sia $n = 2^k$

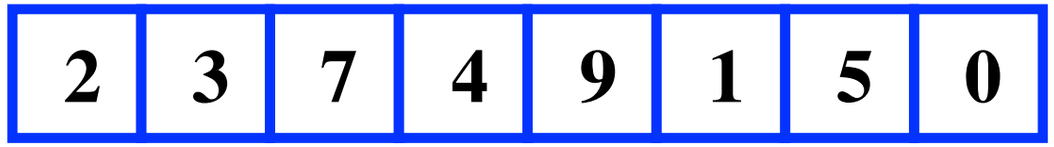
Il passo 1 si esegue in $2^{k-1} * \Theta(2) = \Theta(2^k)$
il passo 2 si esegue in $2^{k-2} * \Theta(2^2) = \Theta(2^k)$
il passo 3 si esegue in $2^{k-3} * \Theta(2^3) = \Theta(2^k)$
…

il passo k-1 si esegue in $1 * \Theta(2^k) = \Theta(2^k)$

Concludiamo che il tempo di calcolo del mergesort $T_{ms}(n) = \Theta(k * 2^k) = \Theta(n \lg n)$

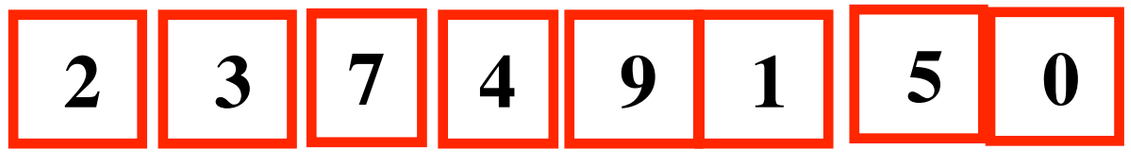
Se $2^k < n < 2^{k+1}$ allora $k2^k < T_{ms}(n) < (k+1)2^{k+1}$ e quindi concludere che in tutti i casi $T_{ms}(n) = \Theta(n \lg n)$

$n = 2^3$

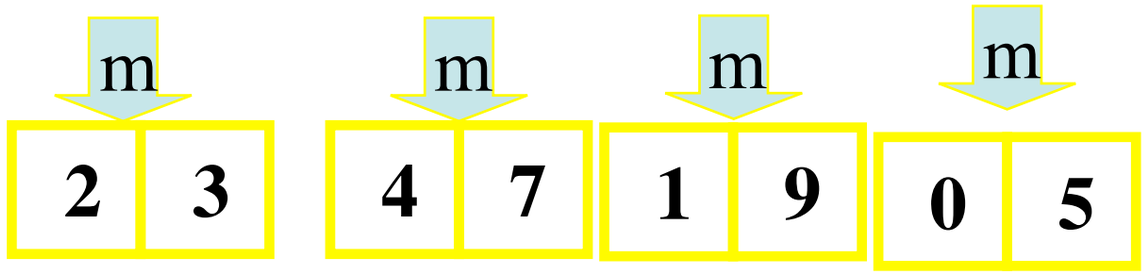


m =merge

dim = 1

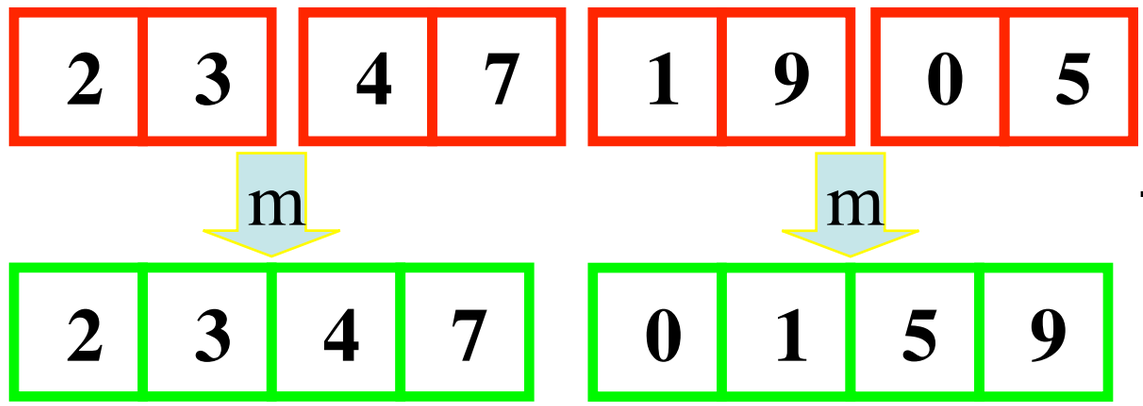


2^2 merge,
ognuna eseguita in $\Theta(2)$



tempo totale $\Theta(n)$

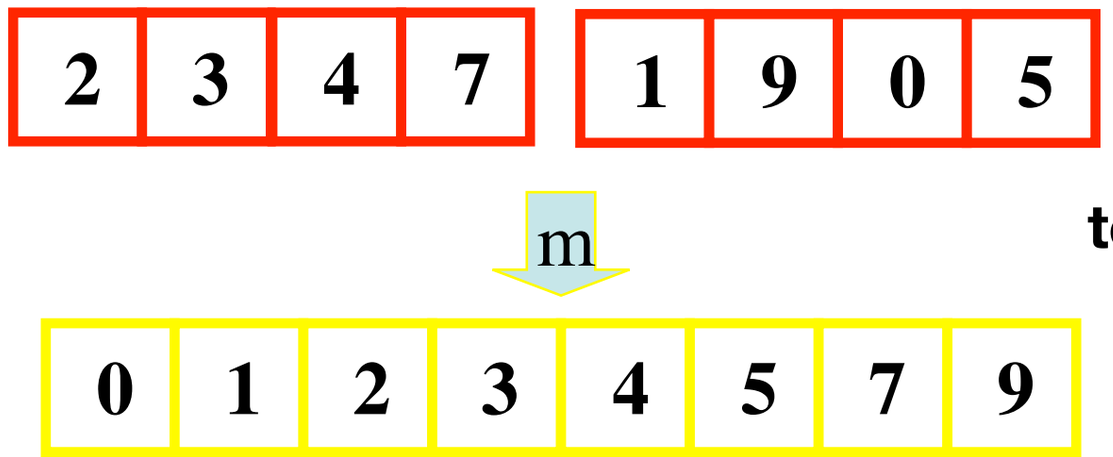
dim = 2



2 merge,
ognuna eseguita in $\Theta(2^2)$

tempo totale $\Theta(n)$

dim = 4



1 merge,
eseguita in $\Theta(2^3)$

tempo totale $\Theta(n)$

Tabella di confronto

operazioni al secondo	dimensione del problema = 1 milione	dimensione del problema = 1 milione	dimensione del problema = 1 milione	dimensione del problema = 1 miliardo	dimensione del problema = 1 miliardo	dimensione del problema = 1 miliardo
andamento algoritmo	n	nlg n	n^2	n	nlg n	n^2
10^6	secondi	secondi	settimane	ore	ore	mai
10^9	istantaneo	istantaneo	ore	secondi	secondi	decenni
10^{12}	istantaneo	istantaneo	secondi	istantaneo	istantaneo	settimane

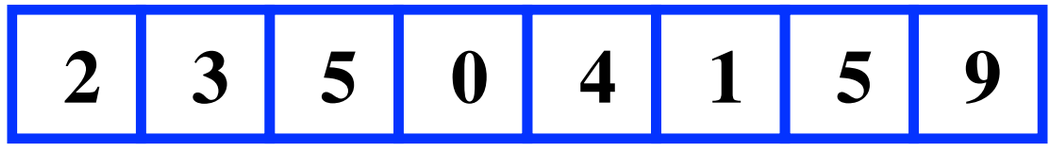
La tabella proviene dal testo di Robert Sedgewick: Algoritmi in Java 3° ed.

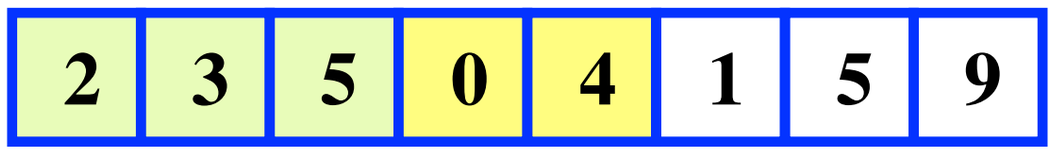
Da wikipedia:
Nel 2021 Aurora, supercomputer presso un laboratorio governativo in Illinois, sarà il primo a superare la soglia degli exaflops, cioè 10^{18} FLoating point Operations Per Second

La maggior parte dei microprocessori moderni può eseguire 4 FLOPs per ciclo di *clock*. Pertanto teoricamente un processore single-core con un clock a 2,5 GHz ha una capacità di 10 miliardi di FLOPS, ovvero 10^{10} FLOPS.

Varianti: mergeSort “naturale”

L’idea è di sfruttare la presenza di sotto sequenze ordinate nell’array in input e applicare la merge a queste.





Si fanno più confronti per identificare le sotto sequenze già ordinate, ma in media meno chiamate della merge

TimSort di Python

Inventato da Tim Peters, è l'algoritmo standard di ordinamento in Python, OpenJDK 7 e Android JDK 1.5

- **mergeSort naturale**
- **usa l'insertionSort binario (cioè la versione con la ricerca binaria) per costruire delle sotto sequenze ordinate, se necessario.**
- **Ulteriori passi di ottimizzazione, come sfruttare la migliore prestazione di merge su array lunghi una potenza di 2 e dimensionando accuratamente le sottosequenze da ordinare con l'insertion sort binario.**

L'algoritmo ottenuto è sempre in Θ(n lg n) nel caso peggiore, ma nel caso migliore è lineare e le costanti sono ridimensionate tanto che le prestazioni sperimentali sono migliori in molti casi di altri algoritmi con tempo di esecuzione asintotico in O(n lg n).

Conclusione