Ricerca binaria e mergesort

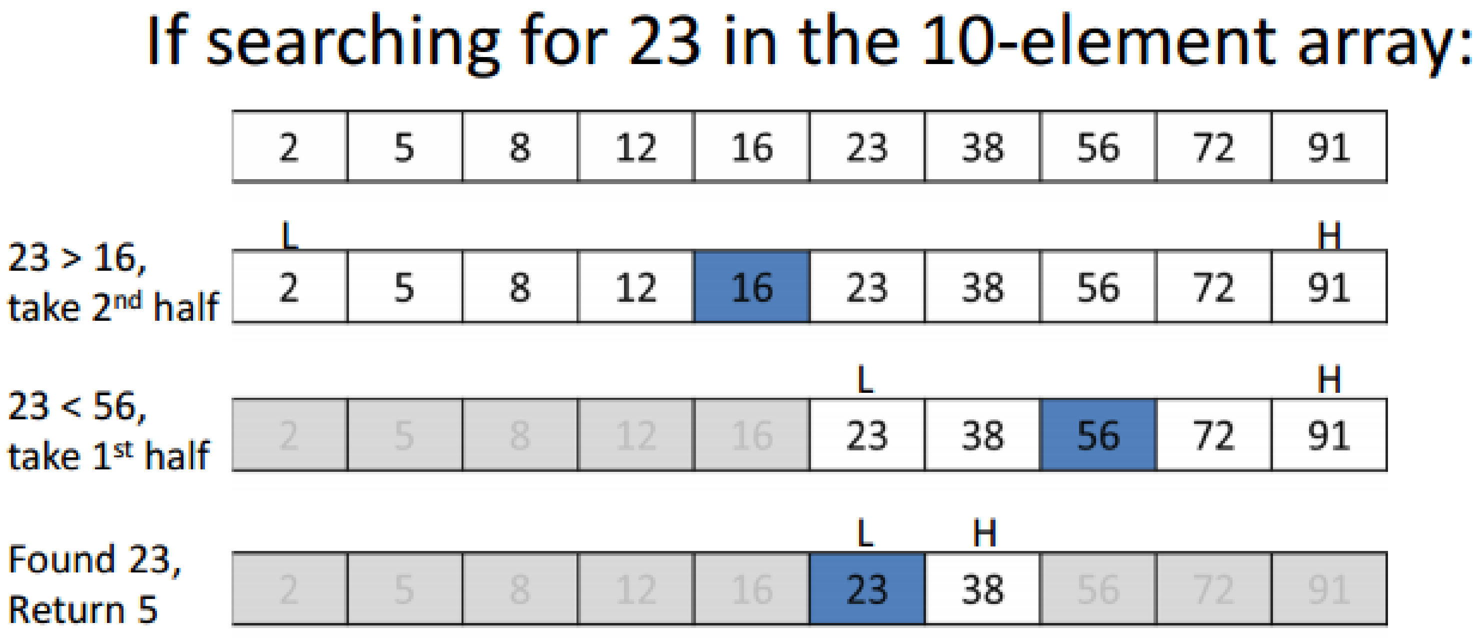
La ricerca binaria

```
int ricercaBinariaAux(int inizio, int fine, int array[], int elem)
{
if (inizio==fine)
   {
   if (array[inizio]==elem)
      return inizio;
   else
      return -1;
   }

int mezzo = (inizio+fine)/2;
if (array[mezzo]==elem) return mezzo;

if (elem > array[mezzo])
   return ricercaBinariaAux(mezzo+1,fine,array,elem);
else
   return ricercaBinariaAux(inizio,mezzo-1,array,elem);
}
```

La ricerca binaria

```
int ricercaBinaria(int dim, int array[], int elem)
{
return ricercaBinariaAux(0, dim-1, array, elem);
}
```

La ricerca binaria



Analisi della ricerca binaria

Θ (log n)

nel caso peggiore

Θ (1)

nel caso migliore

Idea della procedura mergesort

se l'array contiene un solo elemento, return
altrimenti
{
riordina la prima metà dell'array chiamando mergesort
riordina la seconda metà dell'array chiamando mergesort
fondi le due metà così ordinate in un unico array (merge)
}

Idea della procedura merge

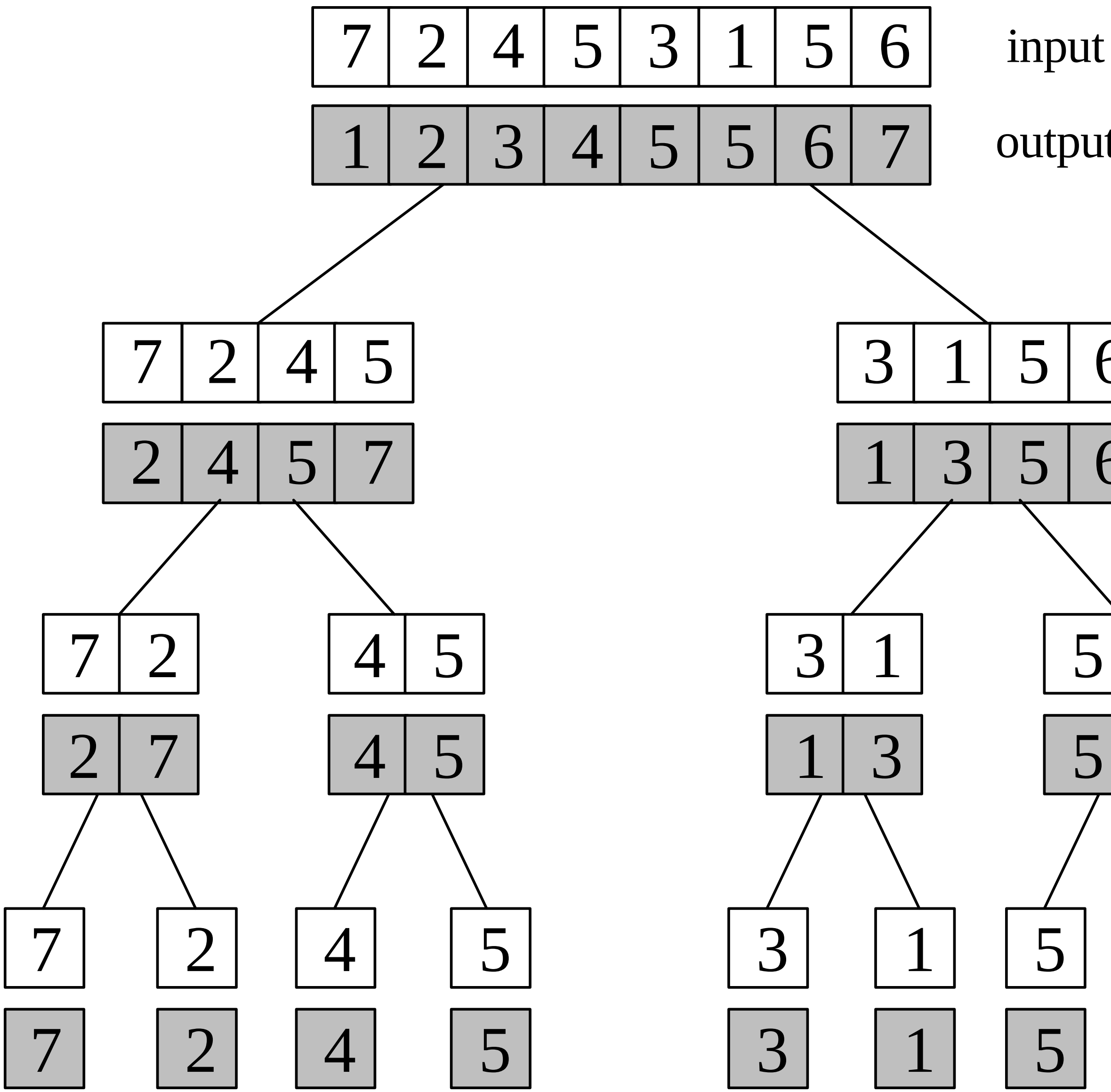
Pseudocode for Merge:

```
C = output [length = n]
A = 1st sorted array [n/2]
B = 2nd sorted array [n/2]
i = 1
j = 1
```

```
for k = 1 to n
        if A(i) < B(j)
                C(k) = A(i)
                i++
        else [B(j) < A(i)]
                C(k) = B(j)
                j++
end
```

(ignores end cases)

Albero delle chiamate ricorsive



Codice C++

```
void mergeSort(vector<int>& v)
{
   ms(v, 0, v.size()-1);
}


void ms(vector<int>& v, unsigned int inizio, unsigned int fine)
{
    if (inizio < fine)
      {
          unsigned int centro = (inizio+fine)/2;
          ms(v, inizio, centro);
          ms(v, centro+1, fine);
          fondi(v, inizio, centro, fine);
      }
}
```

Codice C++

```
void fondi(vector<int>& v, unsigned int inizio, unsigned int centro,
unsigned int fine)
{
   vector<int> vsinistra, vdestra;

   for (unsigned int i=inizio; i<=centro; ++i)
      vsinistra.push_back(v[i]);

   for (unsigned int i=centro+1; i<=fine; ++i)
      vdestra.push_back(v[i]);

   unsigned int indicesinistra = 0;
   unsigned int maxsin = vsinistra.size();
   unsigned int indicedestra = 0;
   unsigned int maxdes = vdestra.size();
```

Codice C++

```
for (unsigned int i=inizio; i<=fine; ++i)
   {
      if (indicesinistra < maxsin && indicedestra < maxdes)
         {
         if (vsinistra[indicesinistra]<vdestra[indicedestra])
            {
             v[i] = vsinistra[indicesinistra];
             indicesinistra++; continue;
             }
         else
            {
            v[i] = vdestra[indicedestra];
            indicedestra++; continue;
            }
         }
```

Codice C++

```
if (indicesinistra == maxsin && indicedestra < maxdes)
        {
          v[i] = vdestra[indicedestra];
          indicedestra++; continue;
        }

      if (indicedestra == maxdes && indicesinistra < maxsin)
        {
          v[i] = vsinistra[indicesinistra];
          indicesinistra++; continue;
        }
  }
}
```

Costo della procedura merge

Indichiamo con $costo_{merge}$ una funzione che va da N in R^+

$costo_{merge}: N \rightarrow R^+$ t.c. $costo_{merge}(n)$ = il numero di operazioni necessarie a fondere due array, ciascuno lungo n/2, per ottenere un array lungo n.

$costo_{merge}(n) = \Theta(n)$

Costo della procedura merge

Da dove deriva $\mathbf{costo_{merge}(n) = \Theta(n)}$ **?**

Inizializzazione dei due cursori **i** e **j** (2 operazioni)

per **n** volte ripeto

{

gestione della guardia del for (2 operazioni: una per l'inizializzazione o l'incremento di **k**, una per il confronto di **k** con **n**)

confronto tra **A(i)** e B**(j)** (1 operazione)

qualunque sia l'esito del confronto, faccio altre 2 operazioni (una assegnazione di un valore a **C(k)** e un incremento di un cursore)

}

Costo della procedura merge

$costo_{merge}(n) = 2 + 5n$

posso trascurare l'addendo 2 e la costante moltiplicativa 5, quindi ottengo un andamento lineare

Costo delle operazioni effettuate al livello j dell'albero della ricorsione

Al livello **j** dell'albero si affrontano $\mathbf{2^j}$ (“2 elevato alla j”) sottoproblemi di tipo “merge”, ognuno di dimensione $\mathbf{n/2^j}$ (“n fratto 2 alla j”)

Il costo delle operazioni effettuate al livello j dell'albero è

numero_problemi * costo_ciascun_problema =

$2^j * \text{costo}_{merge}(n/2^j) = \Theta(2^j * (n/2^j)) = \Theta (n)$

Ad ogni livello, faccio un numero di operazioni in $\Theta (n)$

Costo della procedura mergesort

Indico con $costo_{mergesort}(n)$ il numero di operazioni per ordinare un array di n elementi usando l'algoritmo mergesort

$$costo_{mergesort}(n) =$$

costo operazioni ad ogni livello dell'albero della ricorsione * numero livelli

Il livello più profondo è il livello M in cui i problemi hanno dimensione 1.
Ad un livello j, i problemi hanno dimensione $n/2^j$

Qual è il livello M tale che $n/2^M = 1$?

$n = 2^M$ **sse** $M = \log_2 n$

L'albero della ricorsione ha $\log_2 n + 1$ livelli

Costo della procedura mergesort

L'albero della ricorsione ha $\log_2 n + 1$ livelli

Ad ogni livello dell'albero si eseguono $\Theta(n)$ operazioni.

Trascuro il “+1” nel numero dei livelli: non ha impatto sull'analisi asintotica di complessità

Quindi

$$\mathbf{costo_{mergesort}(n) =}$$

$$\mathbf{\log_2 n \; * \; \Theta(n) =}$$

$$\mathbf{\Theta(n \log n)}$$

Analisi della procedura mergesort

Θ (n log n)

sia nel caso migliore che nel caso peggiore