## Mergesort con allocazione singola

Realizzare una versione del mergesort nella quale l'array di supporto è allocato staticamente (quindi la procedura sarà del tipo `MergeSort(A,B, ...)` dove `A` è l'array di input, mentre `B` è un array della stessa dimensione utilizzato per la memorizzazione temporanea

- *Versione 1*: l'array `B` è utilizzato per memorizzare temporaneamente le parti di array già ordinate delle quali si deve effettuare il merge;
- *Versione 2*: la procedura Merge fa alternativamente il merge dei sottoarray già ordinati da `A` verso `B` e vice versa, di modo da evitare la fase di copia dei sottoarray dei quali fare il merge.
- *Versione 3*: versione iterativa: si procede facendo il merge di frammenti di array sempre più ampi, partendo dalla minima dimensione ordinata ovvero 1. Si combina agevolmente con l'ottimizzazione precedente.

**Soluzione:**

**Versione 1.** In questo caso la soluzione è molto semplice. È sufficiente allocare l'array `B` staticamente. Quindi `B` diventa un argomento della procedura `MergeSort` ricorsiva. Viene usato in `Merge` per contenere temporaneamente i due sottoarray da fondere.

```
# chiamata base:
MergeSort (A)
    n = A.length
    allocate B[1..n]            // alloca l'array B di supporto
    MergeSortRic (A, B, 1, n)


# procedura ricorsiva
MergeSortRic (A, B, p, r)
    if p < r
       q = (p+r)/2
       MergeSortRic (A, B, p, q)
       MergeSortRic (A, B, q+1, r)
       Merge (A, B, p, q, r)


# Merge: usa l'array di supporto per memorizzare i due sottoarray da fondere
Merge (A, B, p, q, r)
    for i = p to q
        B[i] = A[i]
    for j = q+1 to r
        B[j] = A[j]

    i=p
    j=q+1
    for k = p to r
        if (i <= q) and ((j > r) or (B[i] <= B[j])):
            A[k] = B[i]
            i ++
        else:
            A[k] = B[j]
            j ++
}
```

**Versione 2.** Si definisce una procedura di mergesort `MergeSortAlt(A,B,p,r,destA)` con un parametro addizionale, `destA`, un flag booleano, che indica se la destinazione dell'ordinamento è `A` (quando vero) oppure `B` (quando falso). Occorre dunque avere l'accortezza di negare il flag ad ogni passaggio ricorsivo. Inoltre, nel caso base, se la destinazione dell'ordinamento è `A` non c'è niente da fare, se invece è `B`, l'unico elemento di cui si compone il sottoarray va copiato in `B`. Il resto è immutato rispetto a prima.

```
# chiamata base:
MergeSort (A)
    n = A.length
    allocate B[1..n]            // alloca l'array B di supporto
    MergeSortAlt (A, B, 1, n, true)


# MergeSortAlt(A, B, p, r, destA)
# se  destA=true -> ordina  A[p,r] con risultato in A[p,r]
# altrimenti     -> ordina  A[p,r] con risultato in B[p,r]
MergeSortAlt (A, B, p, r, destA)
    if p < r
       q = (p+r)/2
       MergeSortAlt (A, B, p, q, not destA)
       MergeSortAlt (A, B, q+1, r, not destA)
       if destA
            Merge (B, A, p, q, r)
        else
            Merge (A, B, p, q, r)
    else
        if not destA
            B[p]=A[p]


# Merge: fa il merge di S(ource) in T(arget)
Merge (S, T, p, q, r):
    i=p
    j=q+1
    for k = p to r
        if (i <= q) and (not (j <= r) or (S[i] <= S[j]))
            T[k] = S[i]
            i ++
        else:
            T[k] = S[j]
            j ++
```

L'indicazione di quale sia la destinazione è utile solo nel caso base, quando, qualora la destinazione sia l'array `B`, occorre copiare l'elemento da `A` in `B`. Può essere dunque rimossa a patto di fare, prima di iniziare il procedimento, una copia di `A` in `B`. Potrebbe essere inutile, ma è comunque conveniente rispetto a controllare ad ogni chiamata la destinazione.

```
# chiamata base:
MergeSort (A)
    n = A.length
    B[1..n] = A[1..n]             // alloca l'array B di supporto, copia di A
    MergeSortRic (A, B, 1, n)


# MergeSort: ordina S in T (si assume che entrambi gli array
# contengano i valori da ordinare)
MergeSortAlt (S, T, p, r)
    if p < r
       q = (p+r)/2
       MergeSortAlt (T, S, p, q)
       MergeSortAlt (T, S, q+1, r)
       Merge (S, T, p, q, r)

# Merge: fa il merge di S(ource) in T(arget)
Merge (S, T, p, q, r):
    i=p
    j=q+1
    for k = p to r
        if (i <= q) and (not (j <= r) or (S[i] <= S[j]))
            T[k] = S[i]
            i ++
        else:
            T[k] = S[j]
            j ++
```

**Versione 3.** In questo caso, si procede iterativamente facendo il merge di sottoarray di dimensione sempre più grande, partendo dalla minima dimensione ordinata ovvero 1. Si presenta direttamente la versione combinata con l'ottimizzazione precedente, che fa il merge alternativamente su di un array di supporto e quello originale.

```
MergeSortIter (A)
    n = A.length
    allocate B[1..n]            # alloca l'array B di supporto

    # se  destA=true -> merge da B[p,r] in A[p,r] altrimenti da A[p,r] in B[p,r]
    destA = false

    # lunghezza dei sottoarray gia' ordinati
    length = 1

    while length <= n
        if destA
            MergeIter(B, A, n, length)
        else
            MergeIter(A, B, n, length)

        # cambia la destinazione del merge
        destA = not destA
        # raddoppia la lunghezza dei sottoarray ordinati
        length = length*2

    # se alla fine il merge e' in B, quest'ultimo deve essere il risultato
    if destA
        A = B

# fa il merge di sottoarray di dimensione length da S(ource) in T(arget)
# (dove i sottoarray ordinati avranno dimensione 2*length)

MergeIter(S, T, n, length)
        p=0                # punto di inizio del merge
        while (p + length < n)
            q = p+length   # fine del primo sottoarray (il secondo inizia da q+1)
            r = min(q+length,n) # la fine e' il minimo tra la dimensione e q+length
            # merge e' identico a prima
            Merge (A, B, p, q, r)
            p = p + 2*length
```

In implementazioni concrete, risulta conveniente che i sottoarray di dimensione al di sotto di una certa soglia siano ordinati con insertion sort (la versione iterativa con questa ottimizzazione è vicina all'algoritmo noto come TimSort).