Ricerca di duplicati in un array

Verifica la presenza di duplicati

La prima versione dell'esercizio chiede di verificare la presenza di duplicati in un array. Più precisamente, dato un array $A[1..n]$ si vuole verificare se ci sono due indici diversi i e j tali che $A[i] = A[j]$, restituendo conseguentemente un booleano.

L'idea è quella di realizzare una funzione di tipo divide et impera `CheckDup(A,p,r)` che verifica la presenza di duplicati in `A[p,r]` e, qualora non ci siano, ordina l'array `A[p,r]` (in stile mergesort). Le osservazioni fondamentali sono che

- un array con meno di due elementi non contiene duplicati
- un array `A[p,r]` che contenga almeno due elementi, può essere diviso in due parti `A[p,q]` e `A[q+1,r]`, e la presenza di duplicati nell'array originale può essere ridotta alla presenza di duplicati in `A[p,q]` oppure in `A[q+1,r]` oppure di duplicati "a cavallo", ovvero $i \in [p, q]$ e $j \in [q+1, r]$ tali che $A[i] = A[j]$. Per verificare la presenza di duplicati in `A[p,q]` e `A[q+1,r]` si possono effettuare delle chiamate ricorsive, mentre i duplicati "a cavallo" sono ricercati nella fase di merge.

```
CheckDup (A, p, r)
    if r-p > 1           # se l'array contiene almeno due elementi
        q = (r+p)/2
        return CheckDup (A, p, q) or    # duplicati in A[p,q]?
               CheckDup (A, q, r) or    # duplicati in A[q+1,r]?
               Merge (A, p, q, r)       # duplicati tra A[p,q] e A[q+1,r]?
    else
        return False



Merge (A, p, q, r)          # assumendo A[p,q] e A[q+1,r] ordinati, verifica la
                            # presenza di duplicati "a cavallo"
                            # se assenti, ordina l'intero sottoarray A[p,r]

    n1 = q-p+1
    n2 = r-q
    L[1..n1] = A[p..q]
    R[1..n2] = A[q+1..r]
    L[n1+1] = R[n1+1] = infty

    i = j = 1
    k = 0
    while (k <= r) and (L[i] <> R[j])   # i duplicati "a cavallo saranno
                                        # consecutivi
       if L[i] < R[j]
            A[k] = L[i]
            i +=1
        else
            A[k] = R[j]
            j +=1

    return (k <= r)                     # l'assenza di duplicati corrisponde
                                        # al completamento del merge (k <= r)
                                        # in questo caso, l'intero sottoarray
                                        # e' ordinato
```

La complessità è la stessa del mergesort, quindi $\Theta(n \log n)$.

Conta i duplicati

La seconda versione dell'esercizio, richiede di contare duplicati, ovvero le coppie di indici (i, j) con $i < j$ e $A[i] = A[j]$.

Si può raffinare la soluzione precedente osservando che se indichiamo con $dup(A, p, r)$ il numero di duplicati in `A[p..r]` ovvero

$$dup(A, p, r) = |\{(i, j) \mid p \leq i < j \leq r \ \wedge \ A[i] = A[j]\}|$$

allora

- un sottoarray con meno di due elementi, ovvero con $p \leq r$, non contiene nessun duplicato, $dup(A, p, r) = 0$;
- un array `A[p,r]` che contenga almeno due elementi, può essere diviso in due parti `A[p,q]` e `A[q+1,r]`, e il numero di duplicati nell'array originale può essere ottenuto sommando il numero di duplicati in `A[p,q]`, quelli in `A[q+1,r]` e i duplicati "a cavallo", ovvero $i \in [p, q]$ e $j \in [q + 1, r]$ tali che $A[i] = A[j]$. Indicato con

  $$crossDup(A, p, r) = |\{(i, j) \mid p \leq i < j \leq r \ \wedge \ A[i] = A[j]\}|$$

  si ha dunque

  $$dup(A, p, r) = dup(A, p, q) + dup(A, q + 1, r) + crossDup(A, p, q, r)$$

Le considerazioni fatte conducono all'algoritmo che segue:

```
CountDup (A, p, r)
    if r-p > 1           # se l'array contiene almeno due elementi
        q = (r+p)/2
        return CountDup (A, p, q) +    # duplicati in A[p,q]
               CountDup (A, q, r) or   # duplicati in A[q+1,r]
               CrossDup (A, p, q, r)   # duplicati tra A[p,q] e A[q+1,r]
    else
        return False
```

Nella fase di merge, l'idea è di tener traccia di quanti elementi identici sono inseriti consecutivamente da L e R: un blocco di $countL$ elementi da L e di $countR$ elementi da R, determina $countL * countR$ duplicati a cavallo.

```
# funzione che dati il numero di elementi consecutivi inseriti nel
# merge da L e R rispettivamente, ritorna il corrispondente numero di
# duplicati "a cavallo", ovvero countL * countR

Check (countL, countR)
      return countL*countR



CrossDup (A, p, q, r)      # assumendo A[p,q] e A[q+1,r] ordinati, ritorna
                           # il numero di duplicati "a cavallo"
                           # e ordina l'intero sottoarray A[p,r]

    n1 = q-p+1
    n2 = r-q
    L[1..n1] = A[p..q]
    R[1..n2] = A[q+1..r]
    L[n1+1] = R[n1+1] = infty
```

```
i = j = 1

# crossDup contiene i duplicati a cavallo tra A[p,i-1] e
# A[q+1,j-1], inizialmente 0
crossDup = 0

# current contiene l'ultimo elemento inserito da L nel merge,
# mentre countL e countR indicano quanti elementi consecutivi uguali a
# current sono stati inseriti da L e da R rispettivamente

# inizialmente, dato che il processo di merge deve ancora inziare,
# current e' inizializzato ad un valore inesistente
# countL e countR sono inizializzate a zero
current= infty
countL=countR=0

for k = p to r

    if (L[i] <= R[j])        # sceglie il minimo da L e R e lo inserisce
        if L[i] == current   # se coincide con current, il numero di elementi
                             # uguali consecutivi inseriti da L aumenta
            countL++
        else
            # altrimenti aggiorna crossDup usando countL e countR
            crossDup = crossDup + check(countL, countR)

            current = L[i]   # altrimenti si aggiorna current
            countL = 1       # e si ricomincia a contare
            countR = 0

        A[k] = L[i]
        i++

    else:
        if R[j] == current:      # se da R si inserisce un elemento che coincide
                                 # con current, incremento countR
            countR++

        A[k] = R[j]
        j++

# aggiorna con i valori countL e countR calcolati nell'ultima parte del ciclo
crossDup = crossDup + check(countL, countR)

return crossDup
```

Conta gli indici e valori duplicati

Se volessimo contare gli indici duplicati ovvero

$$dupIdx(A, p, r) = |\{i \mid \exists j. j \neq i \ \wedge \ A[i] = A[j]\}|$$

è sufficiente osservare che se ho un blocco di *countL* elementi da L e di *countR* elementi da R identici

- se $countL = 0$ o $countR = 0$ non si identificano nuovi duplicati
- se $countL = 1$ e $countR = 1$, i due elementi sono due nuovi indici duplicati;

- se $countL > 1$ e $countR = 1$, gli indici in L erano già stati identificati come duplicati, mentre l'unico indice in R rappresenta un nuovo duplicato;
- se $countL = 1$ e $countR > 1$, la situazione è duale.

Pertanto, sarà sufficiente modificare la funzione `Check(countL,countR)` come segue:

```
Check(countL,countR)
if (countL==1) and (countR==1)
    res = 2
elseif ((countL==1) and (countR>1)) or ((countL>1) and (countR==1))
    res = 1
else
    res = 0
return res
```

Analogamente, se volessimo contare i valori ripetuti, ovvero

$$dupVal(A, p, r) = |\{A[i] \mid \exists j. j \neq i \ \wedge \ A[i] = A[j]\}|$$

è sufficiente osservare che se ho un blocco di $countL$ elementi da L e di $countR$ elementi da R identici, verifico la presenza di un nuovo valore duplicato solo se entrambi i blocchi hanno dimensione 1. Pertanto, sarà sufficiente modificare la funzione `Check(countL,countR)` come segue:

```
Check(countL,countR)
if (countL==1) and (countR==1)
    res = 1
else
    res = 0

return res
```

In tutti i casi la complessità resta sempre $\Theta(n \log n)$.