

Esercizi Divide-et-impera

- ➊ **Punto fisso:** Scrivere un algoritmo che prenda in input un vettore ordinato A contenente n interi distinti e restituisca **true** se e solo se esiste un indice i tale che $A[i] = i$, in tempo $O(\log n)$.
- ➋ **Chi manca?:** Scrivere un algoritmo che prenda in input un vettore ordinato $A[0 \dots n - 1]$ contenente n elementi interi distinti appartenenti all'intervallo $0 \dots n$ e restituisca in tempo $O(\log n)$ l'unico intero dell'intervallo $0 \dots n$ che non compare in A .
- ➌ **Vettori unimodulari:** Un vettore di interi distinti A è detto **unimodulare** se esiste un indice h tale che $A[0] > A[1] > \dots > A[h - 1] > A[h]$ e $A[h] < A[h + 1] < A[h + 2] < \dots < A[n - 1]$, dove n è la dimensione del vettore. Scrivere un algoritmo che prenda in input un vettore unimodulare e restituisca il valore minimo del vettore in tempo $O(\log n)$.

Discutere correttezza e complessità degli algoritmi proposti.

Samarcanda

Nel gioco di Samarcanda, ogni giocatore è figlio di una nobile famiglia della Serenissima, il cui compito è di partire da Venezia con una certa dotazione di denari, arrivare nelle ricche città orientali, acquistare le merci preziose al prezzo più conveniente e tornare alla propria città per rivenderle.

Scrivere un algoritmo che prenda in input un vettore P contenente n interi in cui $P[i]$ è il prezzo di una certa merce al giorno i e restituisca il guadagno massimo $P[y] - P[x]$ che si può ottenere comprando la merce nel giorno x e rivendendola il giorno y , con $x < y$.

Discutere correttezza e complessità dell'algoritmo proposto.

Spoiler alert!

Esercizi Divide-et-Impera

Una spiegazione più approfondita del funzionamento di `missing()`, `fixedPoint()`, `vum()` si trova nel file 12 - Divide-et-impera che si trova a questo indirizzo:

<http://cricca.disi.unitn.it/montresor/teaching/asd/materiale/esercizi/esercizi-argomento/>

Le soluzioni di questi tre problemi si basano su ricerca dicotomica, quindi hanno complessità pari a $O(\log n)$.

Punto fisso

```
boolean fixedPoint(int[] A, int n)
```

```
return fixedPointRec(A, 0, n - 1)
```

```
boolean fixedPointRec(int[] A, int start, int end)
```

```
if start > end then
```

```
    return false
```

```
else
```

```
    int mid =  $\lfloor (start + end) / 2 \rfloor$ 
```

```
    if A[mid] == mid then
```

```
        return true
```

```
    else if A[mid] < mid then
```

```
        return fixedPointRec(A, mid + 1, end)
```

```
    else
```

```
        return fixedPointRec(A, start, mid - 1)
```

Chi manca?

```
int missing(int[] A, int n)
```

```
if A[n - 1] == n - 1 then
```

```
    | return n
```

```
else
```

```
    | return missingRec(A, 0, n - 1)
```

```
int missingRec(int[] A, int start, int end)
```

```
if start == end then
```

```
    | return start
```

```
else
```

```
    | int mid =  $\lfloor (start + end) / 2 \rfloor$ 
```

```
    | if A[mid] == mid then
```

```
        | return missingRec(A, mid + 1, end)
```

```
    | else
```

```
        | return missingRec(A, start, mid)
```

Vettori unimodulari

```
int vum(int[] A, int n)
```

```
return vumRec(A, 0, n - 1)
```

```
int vumRec(int[] A, int start, int end)
```

```
if start == end then
```

```
    | return A[start]
```

```
else
```

```
    | int mid =  $\lfloor (start + end) / 2 \rfloor$ 
```

```
    | if A[mid] < A[mid + 1] then
```

```
        | return vumRec(A, start, mid)
```

```
    | else
```

```
        | return vumRec(A, mid + 1, end)
```

Struttura della soluzione:

- Si divide il vettore a metà
- Si applica ricorsivamente la soluzione nelle due metà, ottenendo la migliore soluzione nella metà sinistra e nella metà destra
- Da questo approccio, non vengono considerate le soluzioni in cui si acquista nella metà sinistra del vettore, si vende nella metà destra
- Si calcola quindi la differenza fra il massimo a destra e il minimo a sinistra, utilizzando zero nel caso tale differenza sia negativa
- Caso base: un elemento solo, che dà origine a zero come massimo guadagno

Samarcanda

```
int samarcanda(int[] P, int start, int end)
if start == end then
    return 0
else
    int mid = ⌊(start + end)/2⌋
    int maxDiffL = samarcanda(P, start, mid)
    int maxDiffR = samarcanda(P, mid + 1, end)
    int minL = min(P, start, mid)
    int maxR = max(P, mid + 1, end)
    int maxDiffAcross = max(0, maxR - minL)
    return max(maxDiffL, maxDiffR, maxDiffAcross)
```

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases} = \Theta(n \log n)$$

- La soluzione così proposta non è quella ottima, perché il problema può essere risolto in tempo $\Theta(n)$
- Un possibile approccio consiste nel calcolare il vettore delle differenze fra le quotazioni, e applicare la somma massimale vista alla prima lezione. Costo: $\Theta(n)$.
- Altrimenti, è facile progettare una soluzione ad-hoc per il problema

Samarcanda

```
int samarcanda(int[] P, int n)
```

```
int[] diff = new int[0...n - 2]  
for i = 0 to n - 2 do  
    | diff[i] = P[i + 1] - P[i]  
return maxsum4(diff, n - 1)
```

$$T(n) = \Theta(n)$$

Samarcanda

```
int, int, int samarcanda(int[] P, int start, int end)
if start == end then
    return (P[start], P[start], 0)
else
    int mid = ⌊(start + end)/2⌋
    int minL, maxL, maxDiffL = samarcanda(P, start, mid)
    int minR, maxR, maxDiffR = samarcanda(P, mid + 1, end)
    int maxDiffAcross = max(0, maxR - minL)
    return ( min(minL, minR), max(maxL, maxR),
            max(maxDiffL, maxDiffR, maxDiffAcross) )
```

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + 1 & n > 1 \end{cases}$$
$$= \Theta(n)$$