# *Algoritmi e Strutture Dati – 31/08/2021*

## Esercizio A1 – Complessità – Punti $\geq 8$

È facile vedere che $T(n)$ è $\Omega(n^2)$, per via della parte non ricorsiva. Verifichiamo se $T(n)$ è anche $O(n^2)$.

- **Caso base**: per $n = 1, \ldots, 5$, si deve dimostrare che $T(n) = 1 \leq cn^2$; questa disequazione è vera per qualunque $c \geq 1$, $c \geq 1/4$, $c \geq 1/9$, $c \geq 1/16$, $c \geq 1/25$ rispettivamente; è quindi vera per ogni $c \geq 1$.
- **Ipotesi induttiva**: ipotizziamo che per ogni valore $k$, $1 \leq k < n$: $T(k) \leq k^2$;
- **Passo induttivo**: consideriamo i casi con $n \geq 6$, per i quali $\lfloor n/2 \rfloor$ e $\lfloor n/6 \rfloor$ sono maggiori o uguali a 1 (il primo caso base); applichiamo la sostituzione e otteniamo:

$$\begin{aligned} T(n) &= 3T(\lfloor n/2 \rfloor) + 8T(\lfloor n/6 \rfloor) + n^2 & \\ &\leq 3c\lfloor n/2 \rfloor^2 + 8c\lfloor n/6 \rfloor^2 + n^2 & \text{Sostituzione} \\ &\leq 3cn^2/4 + 8cn^2/36 + n^2 & \text{Eliminazione } \lfloor\,\rfloor \\ &= \frac{35}{36}cn^2 + n^2 \stackrel{?}{\leq} cn^2 & \text{Passaggio algebrico} \end{aligned}$$

L'ultima disequazione è vera per $c \geq 36$.

Abbiamo quindi dimostrato che $T(n) = O(n^2)$, per $m = 1$ e $c = 36$. Poiché $T(n)$ è anche $\Omega(n^2)$, abbiamo che $T(n) = \Theta(n^2)$.

## Esercizio A2 – ContainsZero – Punti $\geq 10$

**Soluzione $O(n^3)$:** Soluzione banale, con tre cicli annidati: due cicli per scorrere tutte le coppie di indici $i, j$, con $i \leq j$, un ciclo per sommare tutti gli elementi all'interno di $A[i \ldots j]$. Corrisponde alla Versione 1 di Somma Massimale.

**Soluzione $O(n^2)$:** Basata su due cicli annidati in cui la somma viene mano a mano calcolata durante il ciclo interno. Corrisponde alla Versione 2 di Somma Massimale:

```
boolean containsZero(int[] A, int n)
for i = 1 to n do
    int tot = 0
    for j = i to n do
        tot = tot + A[i]
        if tot == 0 then
            return true
return false
```

**Soluzione $O(n \log n)$:** Soluzione basata sulle somme parziali e sull'ordinamento delle stesse. Sia $S[i] = \sum_{k=1}^{i}$ la somma dei primi $i$ valori del vettore, con $i \geq 0$. Se due elementi $S[i]$ e $S[j]$, con $i < j$, hanno lo stesso valore, si può dedurre che la somma dei valori compresi fra $i + 1$ e $j$ è pari a zero, e possiamo quindi restituire **true**. Si noti che con questa definizione, si cattura anche la possibilità che una somma parziale sia 0, che troverebbe una corrispondenza nella somma dei primi zero elementi.

```
boolean containsZero(int[] A, int n)
int[] S = new int[0 ... n]
S[0] = 0
for i = 1 to n do
    S[i] = S[i − 1] + A[i]
sort(S, 0, n)
for i = 1 to n do
    if S[i] = S[i − 1] then
        return true
return false
```

Il costo è $\Theta(n \log n)$, dominato dall'ordinamento.

Si noti che in nessuna delle tre soluzioni precedenti abbiamo sfruttato il fatto che il vettore sia ordinato. Nelle prime due soluzioni, è possibile modificare il codice interrompendo la ricerca qualora la somma ottenuta sia positiva, in quanto sommando ulteriori termini positivi può solo crescere, ma la complessità nel caso pessimo resta la stessa.

**Soluzione $O(n)$:** Utilizziamo un approccio ad-hoc, simile al divide-et-impera, in cui eliminiamo un elemento alla volta ottenendo un sottoproblema più piccolo.
Consideriamo il sottoproblema $A[i \dots j]$, con $i \leq j$ e cerchiamo di ridurlo a un problema più piccolo. Il problema iniziale è $A[1 \dots n]$. Sia $tot = \sum_{k=i}^{j} A[k]$ la somma del sottovettore. Possono darsi tre casi:

- se $tot = 0$, abbiamo trovato il sottovettore cercato;
- se $tot < 0$, qualunque sottovettore $A[i \dots k]$, con $k < j$, avrà somma negativa per via dell'ordinamento (se gli elementi "in fondo" al vettore sono positivi, la somma diminuisce; se sono negativi, sono rimasti solo valori negativi). Quindi $A[i]$ può essere escluso dalla ricerca incrementando $i$;
- se $tot > 0$, qualunque sottovettore $A[k \dots j]$, con $k > i$, avrà somma positiva per via dell'ordinamento (se gli elementi "in testa" al vettore sono negativi, la somma aumenta; se sono positivi, sono rimasti solo valori positivi); quindi $A[j]$ può essere escluso dalla ricerca decrementando $j$.

Per calcolare la somma dei sottovettori, calcoliamo la somma totale del vettore completo e poi rimuoviamo mano a mano gli elementi che vengono esclusi.
In base a queste considerazioni, è possibile realizzare un algoritmo ricorsivo la cui complessità é $\Theta(n)$, in quanto ad ogni passo si elimina un estremo del vettore e all'inizio si effettua una somma di tutti i valori.

```
boolean zeroRec(int[] A, int i, int j, int tot)
    if i > j or A[i] > 0 or A[j] < 0 then
        return false
    else if tot == 0 then
        return true
    else if tot < 0 then
        return zeroRec(A, i + 1, j, tot − A[i])
    else
        return zeroRec(A, i, j − 1, tot − A[j])
```

```
boolean containsZero(int[] A, int n)
    return zeroRec(A, 1, n, sum(A))
```

Si noti che le condizioni addizionali $A[i] > 0$ e $A[j] < 0$ servono a stoppare la ricorsione quando siamo certi che non sia possibile trovare un sottovettore di somma zero: infatti,

- se $A[i] > 0$, per via dell'ordinamento tutti gli elementi successivi sono positivi e non è possibile trovare un sottovettore di somma zero non vuoto;
- simmetricamente, se $A[j] < 0$, tutti gli elementi precedenti sono negativi e non è possibile trovare un sottovettore di somma zero non vuoto.

Tuttavia, se tali condizioni non fossero presenti, l'algoritmo terminerebbe comunque quando $i > j$.
È anche possibile trasformare questo algoritmo in una versione iterativa, sempre di costo $\Theta(n)$.

```
boolean containsZero(int[] A, int n)
    int i = 1
    int j = n
    int tot = sum(A)
    while tot ≠ 0 and A[i] ≤ 0 and A[j] ≥ 0 and i ≤ j do
        if tot < 0 then
            tot = tot − A[i]
            i = i + 1
        else
            tot = tot − A[j]
            j = j − 1
    return tot == 0 and i ≤ j
```

**Nota** "Degra" e Riccardo Benevelli mi fanno notare che se invece di sommare tutti i valori e poi rimuovere, cerco il valore più vicino a zero (con una ricerca dicotomica) e inizio ad "allargare" il vettore invece di ridurlo, ottengo un algoritmo che è $O(n)$ (nel caso lo zero non sia presente oppure il sottovettore che dà origine a zero è il vettore completo) ma $\Omega(\log n)$ nel caso il vettore contenga un valore 0.

## Esercizio A3 – Radice quadrata – Punti $\geq 12$

Utilizziamo la tecnica divide-et-impera per calcolare la radice quadrata di $n$. Cerchiamo il più piccolo intero $m$ tale per cui $m^2 \geq n$. Il valore $n$ è un quadrato perfetto se e solo se $n = m^2$.
Siano dati due indici $i, j$ tale per cui il valore cercato sia sicuramente compreso nell'intervallo $[i, j]$ (estremi inclusi). Se $i = j$, allora restituiamo quel valore. Altrimenti, calcoliamo il valore mediano $m$; possono darsi tre casi:

- Se $m^2 = n$, allora abbiamo trovato la radice quadrata intera di $n$ e possiamo restituire tale valore;
- se $m^2 > n$, allora restringiamo l'intervallo di ricerca a $[i, m]$, in quanto $m$ potrebbe essere il valore più piccolo;
- altrimenti, se $m^2 < n$, allora restringiamo l'intervallo di ricerca a $[m + 1, j]$, in quanto il valore cercato è sicuramente più grande di $m$.

Utilizzando il criterio di costo uniforme, il costo computazionale è $O(\log n)$. Utilizzando il criterio di costo logaritmico, supponiamo che $n$ sia rappresentato con $k$ bit, $n = O(2^k)$. Le operazioni sugli interi (confronto, somma, divisione per due) costano $O(k)$; l'elevamento al quadrato costa $O(k^2)$ con l'algoritmo banale. Il costo è quindi pari a $T(n) = O(k^2 \log n) = O(\log^3 n)$.

```
int isSquareRec(int n, int i, int j)
    if i == j then
        return i
    else
        int m = ⌊(i + j)/2⌋
        if n == m^2 then
            return m
        else if m^2 > n then
            return isSquareRec(n, i, m)
        else
            return isSquareRec(n, m + 1, j)
```

```
boolean isSquare(int n)
    int v = squareRec(n, 1, n)
    return n == v^2;
```

Si noti tuttavia che un semplice algoritmo come questo:

```
isSquare(int n)
    int i = 1
    while i^2 < n do
        i = i + 1
    return i^2 == n
```

ha costo $O(\sqrt{n})$ misurato con il criterio di costo uniforme; ha costo $O(\log^2 n \cdot \sqrt{n})$ se misurato con criterio di costo logaritmico.

## Esercizio B1 – PrintBitsK – Punti $\geq 8$

Risolviamo il problema utilizzando la tecnica di backtrack. La funzione ricorsiva prende in input:

- il vettore $S$ dove verranno memorizzate le stringhe binarie;
- l'indice $i$ che rappresenta la posizione dove scrivere il prossimo bit (inizialmente $n$, quando raggiunge zero la stringa è terminata);
- il valore $c_1$ che indica quanti bit 1 consecutivi è ancora possibile inserire (inizialmente $k$, quando raggiunge zero non è più possibile inserire bit 1);
- il valore $k$, costante, necessario per riportare $c_1$ a $k$ quando si inserisce un bit 0.

Se $i = 0$, abbiamo riempito tutti i bit (partendo dal fondo) e quindi possiamo stampare la stringa binaria. Altrimenti, si prova a inserire 0 (sempre possibile), passando il valore $k$ nel parametro $c_1$ della chiamata ricorsiva; se $c_1$ è maggiore di zero, è possibile inserire anche il valore 1, riducendo di 1 il valore $c_1$.
Nel caso $k = n$, è possibile stampare tutte le stringhe binarie di $n$ elementi, che sono $2^n$, ognuna delle quali viene stampata in tempo $O(n)$. Il costo totale è quindi $O(n2^n)$.

```
printRec(int [] S, int i, int c1, int k)
if i == 0 then
    print(S)
else
    S[i] = 0
    printRec(S, i − 1, k, k)
    if c1 > 0 then
        S[i] = 1
        printRec(S, i − 1, c1 − 1, k);
```

```
printBits(int n, int k)
int[] S = new int[1 . . . n]
printRec(S, n, k, k);
```

## Esercizio B2 – Sequenza $k$-limitata – Punti $\geq 10$

Questo problema è risolubile in maniera simile all'Esercizio B3 del 04/09/2020, anche se prende le definizioni dall'esercizio B3 del 3/7/2020.
Si utilizza la tecnica backtrack utilizzando uno stack. In questa versione, partiamo dal fondo. Se l'elemento che si sta considerando dista più di $k$ dall'ultimo valore inserito nello stack (oppure se lo stack è vuoto), abbiamo due possibilità: si può inserire oppure no nello stack. Se invece è più grande, viene saltato.
Nel caso di un vettore $V$ in cui $max(V) - min(V) \leq k$, ognuna delle $2^n$ sottosequenze deve essere stampata, con costo $\Theta(n)$. La complessità computazionale è quindi $O(n \cdot 2^n)$.

```
printKSequence(int[] A, int n, int k)
STACK S = Stack()
printRec(A,k,n,S)
```

```
printRec(int[] A, int k, int i, STACK S)
if i==0 then
    print S                                          % LIFO order
else
    if S.isEmpty() or |A[i] − S.top()| ≤ k then
        S.push(A[i])
        printRec(A, k, i − 1, S)
        S.pop()
    printRec(A, k, i − 1, S)
```

## Esercizio B3 – Somma Palindroma – Punti $\geq 12$

Le somme palindrome di lunghezza pari hanno lo stesso valore all'inizio e alla fine della sequenza. È quindi sufficiente considerare tutti i possibili valori compresi fra 1 e $n/2$, e sottrarli due volte al valore $n$, considerando quindi tutti le somme palindrome del valore $n - 2i$ con lunghezza pari.
Per effettuare il calcolo, possiamo utilizzare la programmazione dinamica. Sia $DP[i]$ il numero di somme palindrome di $i$ con lunghezza pari. $DP$ può essere calcolato in questo modo, per valori pari:

$$DP[i] = \begin{cases} 1 & i = 0 \\ \sum_{j=1}^{i/2} DP[i - 2j] & i > 0 \end{cases}$$

È facile tradurre questo algoritmo in un algoritmo di programmazione dinamica bottom-up.

```
int countSumEven(int n)
    int[] DP = new int[0...n]
    DP[0] = 1
    for i = 2 to n step 2 do
        DP[i] = 0
        for j = 1 to i/2 do
            DP[i] = DP[i] + DP[i − 2j]
    return DP[n]
```

L'algoritmo ha complessità $O(n^2)$.

**Addendum** A suo tempo, avevo notato che l'output è comunque $2^{n/2-1}$, quindi si potrebbe restituire semplicemente questo valore (in tempo costante $O(1)$ secondo il criterio di costo uniforme, oppure $O(n)$ secondo il criterio di costo logaritmico). Non avevo avuto tempo di trovare una dimostrazione. Alessio Blascovich nel 2023 mi ha dato il suggerimento giusto. Secondo la formula, che funziona solo per valori pari di $n$, sappiamo che:

$$\begin{aligned} DP[i] &= DP[i-2] + DP[i-4] + DP[i-6] \ldots + DP[2] + DP[0] \\ DP[i-2] &= DP[i-4] + DP[i-6] + \ldots + DP[2] + DP[0] \end{aligned}$$

Poichè le due parti in rosso sono uguali, possiamo riscrivere la formula per $DP[i]$ nel modo seguente:

$$\begin{aligned} DP[i] &= DP[i-2] + DP[i-2] \\ &= 2DP[i-2] \end{aligned}$$

Poichè $DP[2] = 1$ ed ad ogni incremento di 2 raddoppiamo il valore, otteniamo $DP[n] = 2^{n/2-1}$.