**Algoritmi e Strutture Dati**
**30 gennaio 2023**

*Note*

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea e motivarne la correttezza.
3. L'efficienza e l'aderenza alla traccia sono criteri di valutazione delle soluzioni proposte.
4. Si consegnano tutti i fogli, con nome, cognome, matricola e l'indicazione *bella copia* o *brutta copia*.

# Domande

**Domanda A** (7 punti) Dare la definizione della classe $\Theta(f(n))$. Mostrare che la ricorrenza

$$T(n) = \frac{3}{4}T(n/3) + T(2n/3) + 2n$$

ha soluzione in $\Theta(n)$.

**Soluzione:** Per provare che $T(n) = O(n)$ dobbiamo dimostrare che $T(n) \leq cn$, per un'opportuna costante $c > 0$. Procediamo per induzione:

$$\begin{aligned} T(n) &= 3/4\,T(n/3) + T(2n/3) + 2n \\ &\leq 3/4\,c(n/3) + c(2n/3) + 2n \qquad \text{[per ipotesi induttiva]} \\ &= 11/12\,cn + 2n \\ &\leq cn \end{aligned}$$

dove, per la validità dell'ultima disuguaglianza $11/12\,cn + 2n \leq cn$, occorre che

$$1/12\,cn \geq cn$$

ovvero, $c \geq 24$, con $n$ qualunque.

Per provare $T(n) = \Omega(n)$ dobbiamo dimostrare che $T(n) \geq dn$, per un'opportuna costante $d > 0$. La dimostrazione è più semplice della precedente

$$\begin{aligned} T(n) &= 3/4\,T(n/3) + T(2n/3) + 2n \\ &\geq 2n \geq dn \end{aligned}$$

Quindi è sufficiente scegliere $0 < d \leq 2$, e la relazione vale per $n$ qualunque.

**Domanda B** (6 punti) Calcolare la lunghezza della longest common subsequence (LCS) tra le stringhe *store* e *shoes*, calcolando tutta la tabella $L[i, j]$ delle lunghezze delle LCS sui prefissi usando l'algoritmo visto in classe.

**Soluzione:** Si ottiene

| | | *s* | *h* | *o* | *e* | *s* |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 |
| *s* | 0 | 1 | 1 | 1 | 1 | 1 |
| *t* | 0 | 1 | 1 | 1 | 1 | 1 |
| *o* | 0 | 1 | 1 | 2 | 2 | 2 |
| *r* | 0 | 1 | 1 | 2 | 2 | 2 |
| *e* | 0 | 1 | 1 | 2 | 3 | 3 |

La lunghezza della longest common subsequence tra *store* e *shoes* è quindi 3.

# Esercizi

**Esercizio 1** (10 punti) Un array di interi `A[1..n]` si dice *3-ordinato* se per ogni coppia di indici $i, j \in [1, n]$ con `i` $\leq$ `j` vale `A[i]%3` $\leq$ `A[j]%3`, dove `k%3` indica il resto della divisione intera di $k$ per 3. Realizzare una funzione `3Order(A)` che dato un array `A[1..n]`, lo rende 3-ordinato. Valutarne la complessità in tempo e in spazio, e indicare se l'algoritmo è stabile. Anche qualora si usasse un algoritmo di ordinamento noto, lo pseudo-codice va comunque scritto esplicitamente.

**Soluzione:** Un'osservazione semplice ma fondamentale è che di fatto si vuole ordinare l'array ottenuto sostituendo ogni valore `A[i]` con `A[i]%3`.

Si può dunque ricorrere un algoritmo di ordinamento classico, sostituendo, nel confronto, i valori `A[i]` con `A[i]%3`, ovvero con il loro resto modulo `3`.

Ad esempio, se si usa il `Mergesort` si avrà complessità di tempo $O(n \log n)$ e spazio $O(n)$, e l'algoritmo è stabile:

```
3Order (A, p, r)
    if p < r
       q = (p+r)/2
       3Order (A, p, q)
       3Order (A, q+1, r)
       Merge (A, p, q, r)


Merge (A, p, q, r)
    n1 = q-p+1
    n2 = r-q
    for i = 1 to n1
        L[i] = A[i]
    for j = 1 to n2
        R[j] = A[p+j]

    L[n1+1]=R[n2+1] = infinity

    i=p
    j=q+1
    for k = p to r
        if (A[i]%3 <= B[j]%3):  // si assume infinity%3 = infinity
            A[k] = L[i]
            i++
        else
            A[k]=B[j]
            j++
}
```

Tuttavia si può osservare che `A[i]%3` ha valori in $\{0, 1, 2\}$ così che si può facilmente utilizzare un counting sort, con complessità di tempo $O(n)$ e spazio $O(n)$, stabile.

```
3Order (A, B, n)
    allocate C[0..2]

    for i=0 to 2
        C[i]=0

    for j=1 to n
        C[A[n]]++

    for i=1 to 2
        C[i]=C[i-1]+C[i]

    for j=n downto 1
        B[C[A[j]]] = A[j]
        C[A[j]]--
```

La soluzione migliore è probabilmente operare una tripartizione, come visto per il QuickSort, sulla base del valore di `A[i]%3`. Concretamente si scorre l'array con un indice $k$ mantenendo tre partizioni, corrispondenti ai valori 0, 1 e 2 per `A[i]%3`: valore 0 in `A[1..p-1]`, valore 1 in `A[p..k-1]`, valore 2 in `A[j..n]`. La complessità di tempo è ancora $O(n)$ ma l'algoritmo è in place, quindi spazio $O(1)$. L'algoritmo non è stabile.

```
3Order (A,n)
  i = 0
  k = 1
  j = n+1
  while k < j

    if A[k]%3 == 0
       i++
       A[k] <-> A[i]
       k++

    else if A[k]%3 == 1
       k++

    else // A[k]%3 == 2
       j--
       A[k] <-> A[i]
```

**Esercizio 2** (9 punti) Supponiamo di avere un numero illimitato di monete di ciascuno dei seguenti valori: 50, 20, 1. Dato un numero intero positivo $n$, l'obiettivo è selezionare il più piccolo numero di monete tale che il loro valore totale sia $n$. Consideriamo l'algoritmo greedy che consiste nel selezionare ripetutamente la moneta di valore più grande possibile.

(a) Fornire un valore di $n$ per cui l'algoritmo greedy *non* restituisce una soluzione ottima.

(b) Supponiamo ora che i valori delle monete siano 10, 5, 1. In questo caso l'algoritmo greedy restituisce sempre una soluzione ottima: dimostrare che ogni insieme ottimo $M^\star$ di monete di valore totale $n$ contiene la scelta greedy.

**Soluzione:**

(a) Per esempio $n = 60$, perché la soluzione ottima è 3 monete da 20, mentre l'algoritmo greedy restituisce 11 monete (una da 50 e 10 da 1).

(b) Sia $M^\star$ una soluzione ottima. Sia $x$ il valore maggiore tra 10, 5, e 1 che sia non superiore a $n$. Se $M^\star$ contiene una moneta di valore $x$, la proprietà è dimostrata. Altrimenti, sia $M \subseteq M^\star$ un insieme di (2 o più) monete di valore totale $x$ (si osservi che tale insieme esiste sempre quando i valori delle monete sono 10, 5, 1); consideriamo $M' = M^\star \setminus M \cup X$, dove $X$ è l'insieme contenente una moneta di valore $x$. $M'$ è un insieme di monete di valore totale $n$ e di cardinalità inferiore a quella di $M^\star$: assurdo, quindi questo secondo caso non può verificarsi, e quindi $M^\star$ contiene necessariamente una moneta di valore $x$.