BucketSort e RadixSort

Come abbiamo già visto possiamo usare l'IntegerSort per ordinare n elementi piccoli.
Esiste una variante chiamata **BucketSort** che ci permette di ordinare n record con chiavi intere da $[1, k]$.

> ⓘ **Record:**
>
> Insieme di informazioni identificate da una chiave primaria.
> EX: Il record matricola potrebbe essere composto da:
>
> 1. Numero di matricola
> 2. Nome
> 3. Cognome
>
> Con il numero della matricola chiave.

Quindi abbiamo n record mantenuti nell'array e ognuno di questi è composto da:

- Campo chiave (Rispetto al quale ordinare).
- Altri campi associati alla chiave (Informazioni satellite).

Per utilizzare il BucketSort basta mantenere un array di liste, anziché di contatori, ed operare come per **IntegerSort**.
La lista $Y[i]$ conterrà gli elementi con chiave uguale ad i, poi concateniamo le liste.

Il tempo d'esecuzione è $O(n + k)$ come per IntegerSort.
Codice per **BucketSort**:

```
def bucket_sort(X, k):

    # 1. Sia Y un array di dimensione k
    Y = [[] for _ in range(k)]

    # 2. for i=1 to k do Y[i]=lista vuota
    # Già fatto nella riga precedente


    # 3. for i=1 to n do
    for i in range(len(X)):
```

```
        # 4. appendi il record X[i] alla lista Y[chiave(X[i])]
        index = int(X[i] * k)  # chiave(X[i]) calcolata in base al valore di
X[i]
        Y[index].append(X[i])

    # 5. for i=1 to k do
    sorted_array = []
    for i in range(k):
        # 6. copia ordinatamente in X gli elementi della lista Y[i]
        sorted_array.extend(sorted(Y[i]))

    return sorted_array

# Esempio di utilizzo
X = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]
k = len(X)  # Numero di bucket
sorted_X = bucket_sort(X, k)
print(sorted_X)`
```

> **Algoritmi stabili:**
>
> Un algoritmo è stabile se preserva l'ordine iniziale tra elementi con la stessa chiave, il BucketSort è stabile se si appendono gli elementi di X in coda alla opportuna lista $Y[i]$

RadixSort

Molto simile al BucketSort può essere usato anche per interi grandi, ordina n interi con valori tra $[1, k]$, il funzionamento è semplice:

1. Rappresentiamo gli elementi in una base b;
2. Eseguiamo una serie di BucketSort;

Partendo dalla cifra meno significativa a quella più significativa:

- Ordiniamo per l'$i-esima$ cifra con il BucketSort;
- $i-esima$ cifra è la chiave, il numero invece l'informazione satellite;
- $i-esima$ cifra è un intero in $[0, b-1]$.

> **Correttezza:**
>
> Se x e y hanno una diversa $t-esima$ cifra, la $t-esima$ passata di BucketSort li ordina,
> Se x e y hanno la stessa t-esima cifra, la **proprietà di stabilità** del BucketSort li mantiene

ordinati correttamente
Dopo la t-esima passata di BucketSort, i **numeri sono correttamente ordinati** rispetto alle t cifre meno significative.

Tempo d'esecuzione:

- $O(log_b(k))$ è il numero di passate del BucketSort.
- Ciascuna passata richiede tempo $O(n+b)$.

Il tempo richiesto è $O((n+b)log_b(k))$.
La complessità del nostro algoritmo dipende principalmente da b, infatti:
Se $b = \Theta(n)$, si ha $O(nlog_n(k)) = O(n\frac{log(k)}{log(n)})$
Che porta ad avere tempo lineare se $k = O(n^c)$, con c costante.

Esercizio dell'oracolo

Dato un vettore X di n interi in $[1,k]$, costruire in tempo $O(n+k)$ una struttura dati (oracolo) che sappia rispondere a domande (query) in tempo $O(1)$ del tipo: "quanti interi in X cadono nell'intervallo $[a,b]$?", per ogni a e b.

Idea: Costruire in tempo $O(n+k)$ un array Y di dimensione k dove $Y[i]$ è il numero di elementi di X che sono minori o uguali ad i. Possiamo fare ciò con IntegerSort, ma invece di salvare il numero di elementi per ogni indice, salviamo il numero di elementi di ogni indice più il numero degli elementi con gli indici precedenti.

```
def CostruisciOracolo(X, k):

    # 1. Sia Y un array di dimensione k
    Y = [0] * k
    # 2. for i=1 to k do Y[i]=0
    # fatto nella riga di prima
    # 3. for i=1 to n do incrementa Y[X[i]]
    for i in range(len(X)):

        Y[X[i]] += 1
    # 4. for i=2 to k do Y[i]=Y[i]+Y[i-1]
    for i in range(1, k):
        Y[i] += Y[i - 1]
    # 5. return Y
    return Y
```

```python
def InterrogaOracolo(Y, k, a, b):

    # 1. if b > k then b=k
    if b > k:
        b = k

    # 2. if a <= 1 then return Y[b]
    if a <= 1:
        return Y[b]
    else:
        # else return (Y[b]-Y[a-1])
        return Y[b] - Y[a - 1]

# Esempio di utilizzo
X = [3, 4, 2, 5, 1]
k = 6  # Valore massimo in X + 1

# Costruisci l'oracolo
Y = CostruisciOracolo(X, k)
print("Oracolo Y:", Y)


# Interroga l'oracolo
a = 2
b = 4
risultato = InterrogaOracolo(Y, k, a, b)
print(f"InterrogaOracolo(Y, {k}, {a}, {b}) = {risultato}")

# Un altro esempio di interrogazione
a = 1
b = 5
risultato = InterrogaOracolo(Y, k, a, b)
print(f"InterrogaOracolo(Y, {k}, {a}, {b}) = {risultato}")`
```