2. Dado un conjunto X con $|X| = n$ y un entero $k \leq n$ queremos encontrar el máximo valor que pueden sumar los elementos de un subconjunto S de X de tamaño k. Más formalmente, queremos calcular $\max_{S \subseteq X, |S|=k} \sum_{s \in S} s$.

---

a) Proponer un algoritmo *greedy* que resuelva el problema, demostrando su correctitud. Extender el algoritmo para que también devuelva uno de los subconjuntos S que maximiza la suma.

Algoritmo

```
SumaSelectiva(X):
  X' = sort(X)

  res = 0
  n = |X'|

  for i: n-k...n:
    res+= X'[i]

  return res
```

Invariante: En la i-ésima iteración, res es la suma de los $i - (n - k)$ elementos más grandes de X'

Demostración de optimalidad

Tenemos que X' es nuestro conjunto X ordenado de forma creciente tal que $\forall i < j :: X'_i < X'_j$, o sea, los k elementos más grandes son los últimos k, tal que nuestro algoritmo greedy G computa:

$G = \sum_{i=n-k}^{n} X'_i$

Y sea O una solución óptima, $O \leq G$ dado que G suma los k elementos más grandes, pero si O es óptima también lo hace, entonces $O = G$

---

b) Dar una implementación del algoritmo del inciso *a*) con complejidad temporal $O(n \log n)$.

```java
public static int ssnlogn(List<Integer> X, int k) {
    int n = X.size();
    X.sort(null); //Es TimSort, O(nlogn)

    int res = 0;
    for (int i = n-k ;i<n ;i++) { //O(n)
        res+= X.get(i);
    }
    return res;
}
```

---

c) Dar una implementación del algoritmo del inciso *a*) con complejidad temporal $O(n \log k)$.

```java
public static int ssnlogk(List<Integer> X, int k) {
  int n = X.size();
  PriorityQueue<Integer> heap = new PriorityQueue<>();

  for (int x : X) { //O(n)
```

```java
        heap.offer(x); //O(logk)
        if (heap.size() > k) heap.poll(); //O(logk)
    }

    int res = 0;
    while(!heap.isEmpty()) res+= heap.poll(); //O(logk)

    return res;
}
```

Acá no dejamos que el heap sea más grande que k, por eso las operaciones quedan $O(\log k)$