Selection problem : Find the k largest elements (best)

Input: Array $A[n]$, k $\quad 1 \le k \le n$

Output: k largest elements of A (need not be sorted)

Naive algorithm 1: Sort A, return $A[n-k \,..\, n-1]$

$RT = O(n \log n)$.

Naive algorithm 2:

Create a max heap q $\qquad$ buildHeap(A) $T\,O(n)$

$O(n\log n)$ ~~for $i \leftarrow 0$ to $n-1$ do $q.add(A[i])$~~

? list $\leftarrow$ empty

$O(k\log n) \leftarrow$ for $i \leftarrow 1$ to k do list.add(q.remove())

// Output is sorted in decreasing order.

$O(n\log n)$ $\qquad O(n + k\log n)$

---

1. $O(n)$ algorithm — divide and conquer
   similar to Quicksort

2. Heap : $RT = O(n \log k)$.

$O(n)$ algorithm for Select :

```
Select (A, k) :
    if k ≤ 0 then return empty list
    if k ≥ n then return A
    select (A, 0, n, k)
    Return A[n-k .. n-1].

Select (A, p, n, k) :    // k^th largest of A[p .. p+n-1]
    r ← p+n-1
    if n < T then
        insertionSort (A, p, r)
        Return A[p+n-k]
    else
        q ← Partition (A, p, r)
        left ← q-p
        right ← r-q
        if right ≥ k then
            return Select (A, q+1, right, k)
        else if right+1 = k then
            return A[q]
        else return select (A, p, left, k-(1+right))
```

If all elements are distinct, then $RT = O(n)$ expected time

Extend version of Select

n is too big to fit in memory. K is small.
or A is a stream.

A → [Remember the k largest elements seen so far.] → Output k largest elements

Priority queue of max size k.
Use a min heap.

```
Select (A, k):   // Extend version
    it ← A.iterator()
    q ← Priority queue of size k   // Min heap
    for i ← 1 to k do
        if (it.hasNext()) { q.add(it.next()) }
        else return q.toArray();

    while (it.hasNext()):
        x ← it.next()
        if x > q.peek() then
            q.remove()
            q.add(x)
        } if we implement our own version of heaps, q.replace(x)
```

Space: O(k)
RT: O(nlogk)

Finding the k largest elements of an unsorted array A of size n:
Naive algorithm 1: sort A and take A[n − k ⋯ n − 1]. RT = O(nlog(n)).
Naive algorithm 2: insert A[0 ⋯ n − 1] into a max heap (priority queue). Repeat k times: Delete max.

The following algorithm runs in expected O(n) time:

```
Select ( A, k ):  // Find the k largest elements of unsorted array A
  n ← A.length
  if k ≤ 0 then return empty list
  if k > n then return A
  Select(A, 0, n, k)
  return A[ n − k ⋯ n − 1 ]   // Output is not in sorted order

Select( A, p, n, k ):   // Find kth largest element of A[ p ⋯ p + n − 1 ].  Precondition k ≤ n.
  r ← p + n − 1
  if n < T then
    insertionSort(A, p, r)
    return A[ p + n − k ]
  else
    q ← partition( p, r )
    left ← q − p
    right ← r − q
    if right ≥ k then     // kth largest element of A[ p ⋯ r ] is also kth largest of A[ q + 1 ⋯ r ]
      return Select( A, q + 1, right, k )
    else if  right + 1 = k then
      return A[ q ]       // Pivot element happens to be kth largest element
    else                  // kth largest in A[ p ⋯ r ] is [k − (right + 1)]th largest in A[ p ⋯ q − 1 ]
      return Select( A, p, left, k − (right + 1) )
```

The above algorithm is not suitable when A is a stream or an array stored on disk that is too big to be stored in memory. This version of the problem can be solved in O(nlog(k)) time by using a priority queue:

```
Select( A, k ):   // Find the k largest elements of a stream A
  it ← A.iterator()
  q ← new Priority Queue (min heap)   // for storing the k largest elements seen
  for i ← 1 to k do
    if it.hasNext() then
      q.add( it.next() )              初始化PQ，先放k个
    else
      return q
  while it.hasNext() do               有更大的，就删掉原来最小的，更新大的进去
    x ← it.next()
    if q.peek() < x then   // This step is more efficiently done with our own implementation of heaps
      q.remove()
      q.add( x )
  return q
```