Homework 4, Solution set

1. The partition will yield a subarray of $n-1$ items less than or equal to the pivot in value and an empty subarray of items larger than the pivot. Recall that the partition performs $n-1$ comparisons. RSelect recurses on the size $n-1$ subarray. The number of comparisons over the tree of recursive calls will be $(n-1)+(n-2)+\ldots+1 = \frac{1}{2}(n-1)n$.

On the other hand, if seeking the largest item, RSelect will stop after the first recursive call, as it will identify the pivot as the largest item. In this case, RSelect performs $n-1$ comparisons.

2. We observe that every item in $A[1:(r-1)m+m]$ is smaller than $A_{r+1}[1] = A[(r+1)m+1]$. Therefore the only items in $A[1:rm]$ that might be smaller than some or all of the items in $A_{r+1}[1:m]$ are the items in $A_r[1:m]$; clearly, there are at most m such items.

This leads to the following algorithm. In turn, we perform merges of A_r and A_{r+1}, for $r = 0, 1, \ldots, n/k-2$. Each merge takes $O(k)$ time, and consequently the algorithm runs in time $O(k \cdot (n/k-1)) = O(n)$.

Our invariant is that following the merge of A_{r-1} and A_r the items in $A[1:rm+m]$ are in sorted order. (In the base case we interpret this as being before any merges happen.)

The base case, $r=0$, holds trivially (for it is after 0 merges).

Suppose the claim holds for $r \le s$. We now show it for $r = s+1$. Because at most m of these items can be larger than items in $A_{s+1}[1:m]$, these potentially larger items can only be found in the rightmost m locations in $A[1:s+m+m]$, i.e., in $A_s[1:m]$.

Thus the merge of A_s and A_{s+1} will leave $A[1:(s+1)m+m]$ fully sorted.

When the algorithm completes, the inductive hypothesis holds for $r = n/m-2$, meaning that all the items in $A[1:n/m \cdot (m-1)+m] = A[1:n]$ are in sorted order.

3. When $m=1$, we want to perform a binary search for $A[i]$ in $B[1:n]$. This takes $O(\log n) = O(\log n/m)$ time in this case.

When $m=n$, we simply merge the two sets; when item $A[i]$ is added to the merged set, the current index into B is one larger than the number of items in B that are smaller than $A[i]$. This takes time $O(m+n) = O(m)$ in this case.

Note that in time $O(\log n/m)$ we can perform a binary search on a set of size n/m. This leads to the following algorithm.

Take every n/m-th item in B and merge these items with the items in array A. This takes $O(m)$ time.

Now, for each item $A[i]$ in A, we can identify two items in B that straddle it; i.e. there are items $B[k_i \cdot n/m] < A[i] < B[(k_i+1)\cdot n/m]$ (with corner cases when $k_i = 0$ or $k_i = m$ not spelled out). By means of a binary search on $B[k_i \cdot n/m, B[(k_i+1)\cdot n/m - 1]$ we can identify the predecessor of $A[i]$ and consequently how many items in B precede $A[i]$. Each binary search will take $O(\log n/m)$ time, for a total of $O(m \log n/m)$ time.

Thus the overall running time is $O(m + m \log n/m)$.

4. If B is preinitialized to 0, we can proceed as follows. For each item $A[i]$, write a 1 in location $B[A[i]]$ first checking whether there is a 1 or a 0 in that location. A 1 indicates that $A[i]$ is a duplicate.

```
for $i = 1$ to $n$ do
    if $B[A[i]] = 1$ then Print($A[i], i$) is a duplicate”
    else$B[A[i]] = 1$
    end if
end for
```

However,if B is not preinitialized it could have some values set to 1 at the beginning and therefore the above algorithm might detect some false duplicates. We can initialize B to 0, but this would take $O(m)$ time. Instead, with an initial scan over the array A, we will initialize exactly those $B[j]$ which are in the range of A.

```
for $i = 1$ to $n$ do
    $B[A[i]] = 0$
end for
for $i = 1$ to $n$ do
    if $B[A[i]] = 1$ then Print($A[i], i$) is a duplicate”
    else$B[A[i]] = 1$
    end if
end for
```

The algorithm has two for loops which each iterate n times. Each iteration takes $O(1)$ time, giving an overall $O(n)$ runtime.