

CS375 Small Assignment 6 Solution

Due Oct 11th 10PM ET

Recall the following algorithms, which we wrote in class:

```
Partition(A, p, r):
    x = A[r]                // our pivot
    i = p                   // the end of the smalls
    for j in range(p, r):  // not inclusive of r
        if A[j] ≤ x:
            swap A[j] and A[i]
            i++
    swap A[r] and A[i]
    return i
```

```
QuickSort(A, p, r):
    q = Partition(A, p, r)  // Restructure the array into
                           //      bigs and smalls
    if q - p > 1:           // if there is more than 1 item
                           //      in the smalls
        QuickSort(A, p, q - 1) // recurse on the smalls
    if r - q > 1:           // if there is more than 1 item
                           //      in the bigs
        QuickSort(A, q + 1, r) // recurse on the bigs
```

1. First, we showed that the worst case runtime of QUICKSORT was $O(n^2)$: show that it is also $\Omega(n^2)$ (by identifying an input on which it will take $\Omega(n^2)$ time)

Solution: On a sorted or reverse sorted input, QuickSort will make parts of size $n - 2$ and 0. As a result, the runtime on such an input would satisfy $T(n) = T(n - 2) + \Theta(n)$, which is $\Theta(n^2)$. Hence the worst case runtime is $\Omega(n^2)$.

Next, we wrote down the following few attempts at an algorithm to solve the Selection problem: given an unsorted array A of length n and an integer $1 \leq i \leq n$, determine the i^{th} smallest item in A .

```

BadSelect(A, p, r, i):
    q = Partition(A, p, r)    // Restructure the array into
                               //      bigs and smalls
    k = q - p + 1             // The number of items in the smalls
    if i == k:                // The pivot was conveniently
        return A[q]          //      the answer
    else if i < k:            // We are looking for something in
                               //      the smalls, and so should
                               //      recurse in that side
        return BadSelect(A, p, q - 1, i)
    else:                    // We are looking for something in
                               //      the bigs, and so should
                               //      recurse in that side
        return BadSelect(A, q + 1, r, i - k)

```

2. We showed that the worst case runtime of BADSELECT was $O(n^2)$: show that it is also $\Omega(n^2)$ (by identifying an input on which it will take $\Omega(n^2)$ time)

Solution: Suppose we ran BADSELECT with an input of a sorted list and $i = 1$. Then, just as in part 1 above, we could only rule out 2 values per iteration, giving a runtime of $T(n) = T(n - 2) + \Theta(n)$, giving a runtime of $\Omega(n^2)$.

Next, we wrote the following algorithm to get a linear runtime algorithm for Selection:

```
Select(A, p, r, i):
    if (r - p < 5):                // if there are at most 5 items left
        QuickSort(A, p, r)        // Run any sorting algorithm
        return A[p + i - 1]       // Return the ith smallest item

    // The following pseudocode will make a list MedianList
    // comprised of medians taken from each sequential group of 5
    MedianList = empty list
    for j in range((r - p + 1) / 5):
        QuickSort(A, p + j * 5, p + j * 5 + 4)
        add A[p + j * 5 + 2] to MedianList

    // Recursively find the median of the MedianList
    PseudoMedian = Select(A, p, r, (r - p) / 2 + 1)
    // Find the index of PseudoMedian
    // Obviously I'm cheating by calling a function that we haven't
    // actually written but I promise it takes  $O(n)$  time
    index = FindIndex(A, p, r, PseudoMedian)
    // Next, we'll put PseudoMedian at the
    Swap A[r] and A[index]        // end to ensure it's the pivot
    q = Partition(A, p, r)        // Restructure the array into
    // bigs and smalls
    k = q - p + 1                 // The number of items in the smalls
    if i == k:                    // The pivot was conveniently
        return A[q]              // the answer
    else if i < k:                // We are looking for something in
    // the smalls, and so should
    // recurse in that side
        return Select(A, p, q - 1, i)
    else:                        // We are looking for something in
    // the bigs, and so should
    // recurse in that side
        return Select(A, q + 1, r, i - k)
```

3. Why don't we do the same trick for QUICKSORT? That is, if we first found the median of the medians (as in above) and used that as the pivot for QUICKSORT, what would be the resulting worst case runtime?

Solution: In SELECT, we were guaranteed to recurse on exactly **one side** of the resulting parts after running PARTITION. But in QuickSort we are required to recurse on both halves. As a result, the gains from recursing are immediately lost and the total sum of recursive calls sums up to over the input size.