DSA 20407 2013b

Robert Barnes

Maman 13

1) For this question I implemented the Lomuto based quicksort and select, putting print statements in the appropriate places.

Quicksort

```
p = 1   r = 14
55 3 8 4 7 1 10 9 2 10 10 10 6 11
Call Partition. Done Partitioning: the pivot is 11
3 8 4 7 1 10 9 2 10 10 10 6 11 55
Recurse on p = 1    r = 12
3 8 4 7 1 10 9 2 10 10 10 6
Call Partition. Done Partitioning: the pivot is 6
3 4 1 2 6 10 9 7 10 10 10 8
Recurse on p = 1    r = 4
3 4 1 2
Call Partition. Done Partitioning: the pivot is 2
1 2 3 4
Recurse on p = 1    r = 1
Nothing to do
Recurse on p = 3    r = 4
3 4
Call Partition. Done Partitioning: the pivot is 4
3 4
Recurse on p = 3    r = 3
Nothing to do
Recurse on p = 5    r = 4
Nothing to do
The range p = 3 r = 4 is sorted
3 4
The range p = 1 r = 4 is sorted
1 2 3 4
Recurse on p = 6    r = 12
10 9 7 10 10 10 8
Call Partition. Done Partitioning: the pivot is 8
7 8 10 10 10 10 9
Recurse on p = 6    r = 6
Nothing to do
Recurse on p = 8    r = 12
10 10 10 10 9
Call Partition. Done Partitioning: the pivot is 9
```

```
9 10 10 10 10
Recurse on p = 8    r = 7
Nothing to do
Recurse on p = 9    r = 12
10 10 10 10
Call Partition. Done Partitioning: the pivot is 10
10 10 10 10
Recurse on p = 9    r = 11
10 10 10
Call Partition. Done Partitioning: the pivot is 10
10 10 10
Recurse on p = 9    r = 10
10 10
Call Partition. Done Partitioning: the pivot is 10
10 10
Recurse on p = 9    r = 9
Nothing to do
Recurse on p = 11   r = 10
Nothing to do
The range p = 9 r = 10 is sorted
10 10
Recurse on p = 12   r = 11
Nothing to do
The range p = 9 r = 11 is sorted
10 10 10
Recurse on p = 13   r = 12
Nothing to do
The range p = 9 r = 12 is sorted
10 10 10 10
The range p = 8 r = 12 is sorted
9 10 10 10 10
The range p = 6 r = 12 is sorted
7 8 9 10 10 10 10
The range p = 1 r = 12 is sorted
1 2 3 4 6 7 8 9 10 10 10 10
Recurse on p = 14   r = 14
Nothing to do
The range p = 1 r = 14 is sorted
1 2 3 4 6 7 8 9 10 10 10 10 11 55
```

Select

```
p = 1   r = 14  i = 10
55 3 8 4 7 1 10 9 2 10 10 10 6 11
Call Partition
```

```
Done Partitioning: the pivot is 11
3 8 4 7 1 10 9 2 10 10 10 6 11 55
k == 13
Recurse on p = 1    r = 12  i = 10
3 8 4 7 1 10 9 2 10 10 10 6
Call Partition
Done Partitioning: the pivot is 6
3 4 1 2 6 10 9 7 10 10 10 8
k == 5
Recurse on p = 6    r = 12  i = 5
10 9 7 10 10 10 8
Call Partition
Done Partitioning: the pivot is 8
7 8 10 10 10 10 9
k == 2
Recurse on p = 8    r = 12  i = 3
10 10 10 10 9
Call Partition
Done Partitioning: the pivot is 9
9 10 10 10 10
k == 1
Recurse on p = 9    r = 12  i = 2
10 10 10 10
Call Partition
Done Partitioning: the pivot is 10
10 10 10 10
k == 4
Recurse on p = 9    r = 11  i = 2
10 10 10
Call Partition
Done Partitioning: the pivot is 10
10 10 10
k == 3
Recurse on p = 9    r = 10  i = 2
10 10
Call Partition
Done Partitioning: the pivot is 10
10 10
k == 2
The 10th order statistic is 10
```

2) The running time of quicksort can be improved in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. When quicksort is called on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array

to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n\lg(\frac{n}{k}))$ expected time. How should k be picked, both in theory and in practice?

2 A) Assume that quicksort evenly divides the array into $\frac{n}{k}$ groups of k elements each. Then the recursion stops after the subarrays are of size k, and we have that the hight of the recursion tree is $\lg n - \lg k = \lg\frac{n}{k}$. $O(n)$ work is done at each level of the tree, so that the cost of the quicksort step is $O(n\lg\frac{n}{k})$. After executing the quicksort step, the array has the property that for each of the n/k groups, the elements of that group are all less than or equal to the elements of any subsequent groups in the array. Since insertion sort searches backwards for elements greater than the current element, it will never search back more than k steps, meaning that it will run in $O(k^2)$ time on each of the $\frac{n}{k}$ groups. This gives a $O(nk)$ running time for the insertion sort step resulting in a total running time of $O(nk + n\lg\frac{n}{k})$. We want the running time of the hybrid function to be less than quicksort by itself, so we look for k such that $O(nk + n\lg\frac{n}{k}) < O(n\lg n)$ The O notation hides constants so what we're really interested in is that:

$$c_1 \cdot nk + c_2 \cdot (n\lg\frac{n}{k}) < c_2(n\lg n)$$
$$\Leftrightarrow c_1 \cdot nk + c_2 \cdot (n\lg n) - c_2 \cdot n\lg k < c_2(n\lg n)$$
$$\Leftrightarrow c_1 \cdot k - c_2 \cdot \lg k < 0$$

More specifically we want to maximize the value of the function $c_2 \cdot \lg k - c_1 \cdot k$ These average case constants are $1/4$ for insertion sort and 1.386 for quicksort, giving a theoretical value of $k = 8$. In practice, this could be used as a starting point for empirical testing to determine best value of k.

2 B) The first part of the analysis is the same, except that selection sort always scans the entire array for the next smallest value, leading to an $O(n^2)$ running time, asymptotically the same as running selection sort alone. In this case there would be no optimal value for k.

3 A) Write a stooge-sort implementation which makes recursive calls on 3/4 of an array and show why it is correct and what it's worst case running time is.

```
stooge-sort(A, i, j)
if ( A[i] > A[j] )
    swap( A[i], A[j] )
if ( i + 1 >= j )
    return
// calculate 1/4 of the array size
k = floor( (j - i + 1) / 4 )
stooge-sort(A, i, j - k) // sort first 3/4ths of the array
stooge-sort(A, i + k, j ) // sort last 3/4ths of the array
stooge-sort(A, i, j - k) // sort first 3/4ths of the array
```

The recurssion always terminates, because each call either increases `i` or decrease `j` such that after a finite number of steps the condition `i+1>=j` will be true and the function will return.

We see that for `n=1` the array is trivially sorted. For `n=2`, if the array is not sorted, the two elements will be swapped and the function returns resulting in a sorted array. Assume for all `k<n` stooge-sort sorts correctly. Look at the array as 3 regions `A`, `B` and `C`, where the first region `A` is 1/4th of the array, the second region `B` is 2/4ths of the array, and the third region `C` is 1/4th of the array. Since by assumption stooge-sort is correct for `k<n`, we can treat each region as a unit, and the problem is equivelant to sorting 3 elements using only comparisons of adjacent elements, in the order `AB`, `BC`, `AB`. After the first comparison we have `A<B`. After the second comparison we have `A<C` and `B<C`. After the final comparison we have `A<B<C` and the array is sorted.

The recurrence relation for this algorithm is $T(n) = 3T(\frac{3}{4}n) + \Theta(1)$. By case 1 of the master method we have that $T(n) = \Theta(n^{log_{4/3}3}) = \Theta(n^{3.81})$.

3 B) Show the same except now make calls on 1/2 the array.

```
stooge-sort(A, i, j)
if ( A[i] > A[j] )
    swap( A[i], A[j] )
if ( i + 1 >= j )
    return
// calculate 1/4 of the array size
k = floor( (j - i + 1) / 4 )
stooge-sort(A, i, j - 2k) // sort first half AB
stooge-sort(A, i + 2k, j) // sort last half CD
stooge-sort(A, i + k, j -k  ) // sort middle half BC
stooge-sort(A, i, j - 2k) // sort first half AB
stooge-sort(A, i + 2k, j) // sort last half CD
stooge-sort(A, i + k, j -k  ) // sort middle half BC
```

The first portion of the proof is identical to the previous section, except now we are dealing with and array split up into quarter regions, `A`, `B`, `C`, and `D`, and the problem is equivelent to sorting 4 elements, using only comparisons of adjacent elements. After the first sort we have `A<B`.
After the second sort we have `A<B` and `C<D`. After the third sort we have `B<C`, `A<C`, `B<D`. After the fourth sort we have `A<B<C`. After the fifth sort we have `A<D`, `B<D`, `C<D`, `A<B`, and `A<C`. After the sixth sort we have `A<B<C<D` and the array is sorted. Another way of looking at the problem is that the sequence of compares `AB`, `BC`, `CD` move any element to final position `D`. `AB` or `CD`, `BC` moves any element to it's final position in `B`. `AB`, `BC` or `CD` moves any element to `C` and finally `CD`, `BC`, `AB` moves any element to `A`. All these sequences are subseuences of the calls above, and any element only ever moves in one direction towards it's final destination,

so calls which occur “inbetween” don’t interfere with the proper movement of an element towards it’s final position.

The recurrence for this algorithm is $T(n) = 6T(\frac{1}{2}n) + \Theta(1)$ which by case 1 of the master method we have $T(n) = \Theta(n^{log_2 6}) = \Theta(n^{2.58})$.

4) Find in `A`, values `x`,`y` such that `x<y` and `x` occurs more than `n/3` times, and `y` occurs more than `n/4` times. Do this in linear time.

We use `select` to partition the array into four segments and then do a linear search based on the values at the quarter, half and three quarters positions. If such `x` or `y` exist they must cross one of these boundaries since they both must occur more than `n/3` times in the array. There can be at most two `n/3` elements and one `n/4` element, or one `n/3` element and two `n/4` elements. Also, any `n/3` element is also an `n/4` element, so if `x<y` and both are `n/3` elements the condition holds.

```
findxy(A)
// partition into four regions which is O(n)
for( i = 1; i != 4; i = i + 1 )
    R[i] = select(A, floor( A.length * i / 4 ))

// count occurences of each element in the two adjacent regions
// which is O(n)
for( i = 1; i != 4; i = i + 1 )
    Count[i] = 0
    for ( j = n * ( i - 1 ) / 4; j != n * ( i + 1 ) / 4; j = j + 1 )
    if ( A[j] == R[i] )
        Count[i] = Count[i] + 1

// sort R and Count maintaining relative indexes and using Count as the sort
// key.  Result is Counts in descending order. This is O(1) since we're
// always sorting 3 elements.
issort(R, Count)

// R and Count are arranged in descending order of Counts so we only have to
// check these three cases.
if ( R[1] < R[2] && Count[1] > floor(n/3) && Count[2] > floor(n/4) )
    return true
if ( R[1] < R[3] && Count[1] > floor(n/3) && Count[3] > floor(n/4) )
    return true
if ( R[2] < R[3] && Count[2] > floor(n/3) && Count[3] > floor(n/4) )
    return true
else
    return false
```

5) Show that SELECT3 which operates the same as deterministic SELECT, but uses medians of 3 instead of medians of 5, is not $O(n)$.

The first step in the algorithm divides the array into $\lceil \frac{n}{3} \rceil$ groups and finds the median of each of these groups which takes $\Theta(n)$ time, since there are at most $n/3+1$ medians.

Next, we call SELECT3 recursively to find the median of this set which takes $T(\lceil \frac{n}{3} \rceil)$ time.

The number of groups of 3 is $\lfloor \frac{n}{3} \rfloor \geq \frac{n-2}{3}$. The median of medians will be greater than at least 2 elements from half of these groups. This means it will be strictly less than at most $n - \frac{n-2}{3} = \frac{2n-2}{3} < 2n/3+1$ elements. Thus in the worst case we must recurse on $2n/3+1$ elements. From all this we have the recurence:

$$T(n) = T(\left\lceil \frac{n}{3} \right\rceil) + T(\frac{2n}{3}+1) + \Theta(n)$$

For sufficiently large n, any constant terms added in the recursive call are insignificant and we are simply left with the recurance:

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \Theta(n)$$

We guess that $T(n) = O(n \lg n)$ and check with the substitution method.

We assume that $T(n) \leq c \cdot n \lg n$ for all $k < n$, and some constant c.

$$\begin{aligned}
T(n) &\leq c_1 \cdot \frac{n}{3} \lg \frac{n}{3} + c_1 \cdot \frac{2n}{3} \lg \frac{2n}{3} + c_2 n \\
&= c_1 \cdot \frac{n}{3}(\lg n - \lg 3) + c_1 \cdot \frac{2n}{3}(\lg 2 + \lg n - \lg 3) + c_2 n \\
&= c_1 \cdot \frac{n}{3} \lg n - c_1 \cdot \frac{n}{3} \lg 3 + c_1 \cdot \frac{2n}{3} \lg 2 + c_1 \cdot \frac{2n}{3} \lg n - c_1 \cdot \frac{2n}{3} \lg 3 + c_2 n \\
&= c_1 n \lg n + c_1 \cdot \frac{2n}{3} - c_1 n \lg 3 + c_2 n \\
&\leq c_1 n \lg n + c_1 n \lg n + c_2 n \lg n \\
&= (2c_1 + c_2) n \lg n
\end{aligned}$$

And we have shown the exact form of the inductive hypothosis.

This inequality holds for any $n \geq 4$.

If we assume that $T(k) = 1$ for any $k \leq 1$ and take $T(4)$ as our base case then we have that $T(4) = 6 + 4 = 10 \leq 16c_1 + 8c_2$ for any $c_1, c_2 \geq 1/2$ and our base case holds.

We guess that $T(n) = \Omega(n \lg n)$ and check with the substitution method.

We assume that $T(n) \geq c \cdot n \lg n$ for all $k < n$, and some constant c.

$$\begin{gathered}
T(n) \geq c_1 \cdot \frac{n}{3} \lg \frac{n}{3} + c_1 \cdot \frac{2n}{3} \lg \frac{2n}{3} + c_2 n \\
= c_1 \cdot \frac{n}{3}(\lg n - \lg 3) + c_1 \cdot \frac{2n}{3}(\lg 2 + \lg n - \lg 3) + c_2 n \\
= c_1 \cdot \frac{n}{3} \lg n - c_1 \cdot \frac{n}{3} \lg 3 + c_1 \cdot \frac{2n}{3} \lg 2 + c_1 \cdot \frac{2n}{3} \lg n - c_1 \cdot \frac{2n}{3} \lg 3 + c_2 n \\
= c_1 n \lg n + c_1 \cdot \frac{2n}{3} - c_1 n \lg 3 + c_2 n \\
\geq c_1 n \lg n - c_1 n \lg 3 \\
\geq c_1 n \lg n - c_1 \cdot \frac{9}{10} \cdot n \lg n \\
= \frac{1}{10} \cdot c_1 \cdot n \lg n
\end{gathered}$$

This inequality holds for all $n \geq 4$ and we have shown the exact form of the inductive hypothesis. For the base case $T(4) = 10 \geq \frac{c_1 \cdot 8}{10}$ for any $c_1 \leq 10$.