CMPS101 HW 3

Nicole Kyriakopoulos
Jacob Katzeff
Due 4/27/16

Q1.
a. T(n) $\leq 7T(\frac{n}{3}) + n^2$
Let a=7, b=3, f(n)= n^2
Using a case from the Master Theorem:
If f(n) = $\Omega(n^{log_b a + \varepsilon}), \varepsilon > 0$
and if a$f(\frac{n}{b}) \leq cf(n)$ where $c < 1$ then T(n) = $\Theta(f(n))$
$n^2 = \Omega(n^{log_3 7 + \varepsilon})$ is true when $\varepsilon \geq 2 - log_3 7$ so the first requirement for the case is fulfilled.
$7(\frac{n}{3})^2 \leq cn^2$
$\frac{7}{9}n^2 \leq cn^2$ –second requirement is fulfilled when $\frac{7}{9} \leq c < 1$

b. T(n) $\leq 7T(\frac{n}{3})$ + n
a=7, b=3, f(n)= n
Use Case 1: f(n) = O($n^{log_b a - \varepsilon}$) where ε >0. If there is a value for that epsilon, then T(n)=$\Theta(n^{log_b a})$.
n = O($n^{log_3 7 - \varepsilon}$) – true if $0 < \varepsilon \leq log_3 7$ - 1
Therefore T(n) = $\Theta(n^{log_3 7})$

c. T(n) $\leq 7T(\frac{n}{3})$ + 1
Use Case 1:
1 = O($n^{log_3 7 - \varepsilon}$) – true if $0 < \varepsilon \leq log_3 7$
Therefore T(n) = $\Theta(n^{log_3 7})$

These prove Θ bounds but not specifically O time complexity; However, Θ implies both O and Ω so they can be used as that.

Q2.
Let T(n)$\leq n - \sqrt{n} - \sqrt{2n} + 2 \in O(n)$.
T(1)=1, so the base case is true. Assume true for k=1,...,n-1.
In particular, it's true for n/2.
So we get $T(n) \leq 2(\frac{n}{2} - \sqrt{\frac{n}{2}} - \sqrt{n} + 2) + \sqrt{n}$
$= n - \sqrt{2n} - 2\sqrt{n} + 4 + \sqrt{n}$
$= n - \sqrt{2n} - sqrtn + 4$
Thus, we get $T(n) \leq n - \sqrt{2n} - sqrtn + 4$, which is true. Thus it is true for all n, and $T(n) \in O(n)$
Q3.
Adding a counter to the recursions and returning a number of inversions along with the sorted array.

```
merge(array a, array b)
la = length a, lb = length b
sortedarr = []
```

```
initialize x, y, counter
while x < la AND y < lb
nextmin = minimum of a[x], b[y]
add to sortedarr (nextmin)
if b[y]==nextmin:
counter += la - x
y+=1
else:
x+=1

return (sortedarr, counter)


sort(A):
if length(A) == 1
return 0

L = A[0 to n/2]
R = A[n/2 + 1 to n]

(sortedL, counterL) = sort(L)
(sortedR, counterR) = sort(R)
(combinedA, crosscounter) = merge(sortedL, sortedR)
return (combinedA, counterL+counterR+crosscounter)
```

Another solution for Q3.

```
Inversions(A):
n=A.length
i=0
L=A[1...n/2]
R=A[n/2 + 1 ... n]
i+=Inversions(L)          T(n/2)
i+=Inversions(R)          T(n/2)
Mergesort(L)          //O(nlogn)
Mergesort(R)          //O(nlogn)
return i+ModifiedMerge(L,R)          //O(n)

ModifiedMerge(A,B):
n=A.length=B.length
A[n+1]=B[n+1]=∞
count=i=j=0
while A[i] < ∞ or B[j] < ∞:
    if A[i]≤B[j]:
        count+=1
        i+=1
    else:
        j+=1
return count
```

So we get the recurrence T(n)$\leq$2T(n/2)+O(2nlogn + n)=2T(n/2)+O(nlogn+n) with T(1)$\leq$ c for some c
Proof by induction that T(n)=nlogn:
For the case n=1, it's clearly true by picking the correct c value.
Assume it's true from k=1 to k=n-1. In particular, it's true for $T(n/2)$.
So we get T(n)$\leq$ $2\frac{n}{2}log_2\frac{n}{2} + n + nlog_2n$
$= n(log_2n - 1) + n + nlog_2n = nlog_2n - n + n + nlog_2n = 2nlog_2n$
Thus we get $T(n) \leq 2nlog_2n$, which is what we were looking for.
Thus T(n) $\in O(nlog_2n)$
Q4. Part 1. Pseudocode for algorithm, using Partition function from Quicksort algorithm discussed in class:

```
findkth(array A, k)
    i = partition(A)
    if i==k:
        return A[k]
    else if i<k
        return findkth( A[i+1 to length(A), k-i+1 )
    else if i>k
        return findkth(A[0 to i], k)
```

Part 2. Worst Case:
= (n-1)+(n-2)+...+2
= $\sum_{i=1}^{n}(n-i)$
= ((n-1)($\frac{n}{2}$))
= ($\frac{n^2-n}{2}$) = $\Theta(n^2)$

Part 3. for each recursion, each element has a $\frac{1}{n}$ probability of being selected as the pivot where n is the size. We can guess that T(n) = O(nlogn) so T(n) $\leq$ cnlogn as n$\rightarrow\infty$. Base case is T(1) = 0, trivially sorted.
T(n) = (n-1) + $\sum_{i=1}^{n-1}\frac{1}{n}(T(oneside^i))$
Worst Case T(n) $\leq (n-1) + \sum_{i=1}^{n}\frac{1}{n}(T(i))$
= (n-1) + $\frac{1}{n}\sum_{i=1}^{n-1} cilogi$
= (n-1) + $\frac{1}{n}\int_{i=1}^{n} cilogi$
$\leq (n-1) + \frac{c}{n}(\frac{n^2logn}{2} - \frac{n^2}{4} + \frac{1}{4})$
xlogx where x$\geq$1 is concave down so we know this inequality holds, and we integrate by parts
$= (n-1) + \frac{c}{2}nlogn - \frac{cn}{4} + \frac{c}{4n}$
If we let c=3, the weaker terms are quickly approaching 0 or are negative. Therefore T(n) = O(nlogn)

Q5. Pseudocode:

```
findkth(A, k)
    (d, j) = partition(A)
```

```
(tuple of array and its first index)          if K∈[j, j+len(B)]:
          return A[j]
     if k<j:
          return findkth(A[1 ... j-1], k)
     if k> j+length(B):
          return findkth(A[j+length(B) + 1...n], k-j+len(B))
```

The partition function is different:

```
Partition(A):
n=A.length
piv = A[i]
i=1, j=n+1, k=1
while true:
     do i++ while A[i] < piv
     do j- - while A[j] > piv
     if i≥j: break
     if A[i] = piv:
          swap A[i] and A[k+1], k+=1
     if A[j] = piv:
          swap A[i] and A[j]
          i-=1 and j+=1
     if A[i] != piv and A[j] != piv:
          swap A[i] and A[j]
for t=1 to k:
     swap A[t] and A[j-t+1]
return(k, j)
```

Its necessary for all elements to be distinct because if a duplicate is picked as the pivot, there wouldn't be a single partition index, but multiple. You'd have to keep track of where to put the duplicate which screws things up. The algorithm above basically says that if there is duplicates of the pivots, it keeps track of how many there are, and takes that into account when recursively calling findkth. It takes a lot more work to do this, and the worst case is the same as in Q4. The only difference here is if there are duplicates, you have to do more work to keep track of that. It takes 2-3 times more operations for this, which is O(1) times more, so it still ends up being worst case $\emptyset(n^2)$, but average case is still $O(nlog(n))$.