Vlad Predovic
CS325
Due 4/10/2016

Homework 2

1.

a) $T(n)=T(n-2)+n$: T(n-2) = T(n-4) + n - 2

T(n) = T(n-4) +n - 2 + n base case k = n-1

$$T(n) = T(n-k) + \frac{k}{2} n - k$$
$$= n-n+1 + \frac{n-1}{2} * n - n + 1$$
$$= \frac{n2 - 3n + 4}{2} \rightarrow \mathbf{T(n) = O(n^2)}$$

Also, using Muster's theorem:

For d=1, a=1, F(n)=n since F(n) = O(n)

$T(n) = O(n^{d+1}) = O(n^{1+1}) = \mathbf{O(n^2)}$

b) $T(n)=T(n-1)+3$: **Roll pattern out:**

T(n-1) = T(n-2) + 3
T(n-2) = T(n-3) + 3
T(n) = T(n-3) + 3 + 3 + 3 = T(n-3) + 9
To generalize: T(n) = T(n-k) + 3k

Substitute in:

Guess that k = n
T(n) = n-n + 3n
T(n) = 3n $\rightarrow$ O(n)

c) $T(n)=2T(\frac{n}{4})+n$: $n = \mathbf{\Omega}(n^{\log_4 2-E}) = \mathbf{\Omega}(n^{1/2})$ and $2(\frac{n}{4}) \leq cn$ so case 3
$T(n)=\Theta(n)$ of the master method.

d) $T(n)=4T(\frac{n}{2})+n^2\sqrt{n}$: $n^2\sqrt{n} = \mathbf{\Omega}(n^{\log_2 4-E}) = \mathbf{\Omega}(n^2)$ and $4(\frac{n}{2})^{2.5} \leq c\, n^2\sqrt{n}$
so $T(n) = \Theta(n^{2.5})$ by the master method

2.

a. $T(n) = 5T(\frac{n}{2}) + n$	master case 1 /w a=5 b=2 f(n)=n: $n = O(n^{\log_2 5}) = n^2$ **$T(n) = O(n^{2.33})$**
b. $T(n) = 2T(n-1) + c$ define $T(0)$ = constant	First expand using iteration method: $T(n) = 2T(n-1) + c$ $= 2(2T(n-2) + c) + c$ $= 2(2(2T(n-3) + c) + c) + c$ $T(n) = 2^kT(n-k) + c(2^k-1)$ Guess that $T(n) = 2^n$ so then $T(n) \leq d2^n-b$ where d and b are const. Base Case: $T(1) = 1 \leq d2^1 - b$ Which is true when $\frac{1+b}{2} \leq d$ $T(n) = 2T(n-1) + c$ $= 2(d2^{n-1} - b) + c$ $= d2^n - 2b + c \leq d2^n-b$ Which is true as long as $b > c$ **Therefore $T(n) = O(2^n)$**
c. $T(n) = 9T(\frac{n}{3}) + n^2$	Master case 2 /w a=9, b=3 f(n)=n² $n^2 = \Theta(n^{\log_3 9 + E})$ **$T(n) = O(n^2 lg(n))$**

Option c is the best as it has the lowest order exponent: **$O(n^2lg(n))$**

3.

Each call calls foo four times on half the previous input n and then prints n times.

$T(n) = 4T(\frac{n}{2}) + n \rightarrow$ a=4 b=2 f(n)=n; $f(n) = O(n^{\log_2 4 - E}) = O(n^2)$

So due to master theorem case 1: $T(n) = O(n^2)$

4.

Binary: $T(n) \leq T(\frac{n}{2}) + c \rightarrow T(n)$ is $O(\log n)$

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

$T(n) = T(\frac{n}{27}) + c + c + c$

$T(n) = T(\frac{n}{3\hat{}k}) + kc$ stop when $n / 3^k = 1 \rightarrow \log_3 n = k\log_3 3 \rightarrow k = \log_3 n$

$T(n) = T(\frac{n}{3\hat{}\log 3n}) + (\log_3 n)c = T(1) + (\log_3 n)c$

For my Ternary Search T(n) is also O(logn) as the algorithm will take at most logn time. It can bottom out early as well. However in each recursive call there are six comparisons as opposed to three for the binary algorithm. This is double the amount and would prove very costly in the long run.

```
 TernarySearch(array A, query x) {
     If length(A) = 0
          Return false
     Else
          If x = middle element of A
               Return true
          else
               I = length of A / 3
               If x < A[I]
                    B = first third of A
               Elseif x < A[I+I]
                    B = second third of A
               Else
                    B = last third of A
               Return TernarySearch(array B,  x)
  }
```

5.

a) $T(n_2) = n + n$ (merge 1 and 2) $T(n_3) = 2n + n$ (merge 3)
$T(n_k) = (k-1)n + n$ $\rightarrow$ $T(n) = 2n + \ldots + (k-1)n + n = n(2+3+4+\ldots(k-1))$
$T(n) = (k^2+K)/2 - 1$ (POWER SERIES YAH)= **$O(k^2n)$**

b) Take k sorted arrays of size n and merge the kth array with each (k+1) array until you are left with k/2 arrays. Repeat the process until you are left with 1 final array.
In order to merge each array begin at the lowest index of both arrays and create the new array by adding on the lowest value in either array until you run out of elements.
Each comparison to create a merged array takes O(1) time iterating through n elements. Therefore each merge takes O(n) time. Additionally each time a round of merges is completed the amount of merges left is halved.
So ultimately it will take O(knlog(k))

6.

a) With k elements insertion sort will use $\Theta(k^2)$ time. However since there are n/k sub-lists, there ultimately have to be n/k sub-lists of k elements sorted: $\Theta(\frac{n}{k}k^2)$
$\Theta(\frac{n}{\cancel{k}}k^{\cancel{2}}) = \Theta(nk)$

b) Mergesort usually takes $\Theta(n\lg(n))$ to sort n elements because merge sort keeps halving the arrays until you are left with one element in each. However, since the sublists are already sorted, Mergesort does not have to iterate further. Hence n/k 'elements' are considered instead. Substituting n/k elements for n elements results in the following recurrence: where d = n/k
$T(n) = 2T(d/2) + dk$
Applying the Master Theorem case 2 with a = 2, b = 2, $f(n) = dk = \frac{n}{\cancel{k}}\cancel{k} = n$

$f(n) = dk = \Theta(n^{\log_2 2 - E}) = \Theta(d)$ and therefore
$T(n) = \Theta(n^{\log_2 2-E}\lg d) = \Theta(dk\lg d) = \Theta(\frac{n}{\cancel{k}}\cancel{k}\lg d) =$ **$\Theta(n\lg(\frac{n}{k}))$**

c) Since T(nk+nlg($\frac{n}{k}$)) = Θ(nlg(n)) → we know ~~n~~k ≤ ~~n~~lg(n) and ~~n~~lg($\frac{n}{k}$) ≤ ~~n~~lg(n) → k ≤ lg(n) and ~~lgn~~ – lgk ≤ ~~lgn~~ → 1 ≤ k → **1 ≤ k ≤ lg(n)**
If we substitute k for the max value lg(n):
nlg(n)+nlg($\frac{n}{lgn}$) → nlg(n) + nlg(n) - nlg(lgn) → 2nlg(n) – nlg(lg(n))
nlg(lg(n)) is extremely small compared to nlg(n). So therefore
2nlg(n) – nlg(lg(n)) = Θ(nlg(n))

d) By comparing the equations with constants dependent on the implementations chosen. Find at what point Insertion sort is faster than Mergesort. You could write a function that loops, comparing the times of both until Mergesort becomes faster than Insertion sort.

7)
Note: Cannot return two variables at once. That is up to the users implementations (for example in c you could use a struct). The first variable returned is considered the min and the second the max.

```
minmaxCursive(array A) {
     If length(A) = 1
          Return A[0], A[0]
     If length(A) = 2
          Return A[0], A[1]
     Else
          Aleft = A[0…length/2-1]
          Aright = A[length/2…length-1]
          maxr, minr = minmaxCursive(Aright)
          maxl, minl = minmaxCursive(Aleft)
          If maxl < maxr
               Max = maxr
          Else
               Max = maxl
          If minl < minr
               Min = minl
          Else
               Min = minr

          Return(min, max)
  }
```

$T(n) = T(n/2) + T(n/2) + c = 2T(n/2) + c$
As opposed to direct Mergesort where the recurrence is $T(n) = 2T(n/2) + n$,
The items do not have to be merged together.

Using the master theorem case 1:

$a=2, b=2, f(n) = c \rightarrow \quad c = O(n^{\log_2 2 - E}) = O(n) \rightarrow$ **$T(n) = O(n)$**

Iterative Implementation:

```
minmaxSearch(array A) {
     If length(A) = 0
          Return false
     Else
          Array MinMax[2]
          MinMax[0] = A[0]
          MinMax[1] = A[0]
     For value in A:
          If value > MinMax[0]:   MinMax[0]  = value
          Else if value < MinMax[1]:  MinMax[1]  = value

     Return MinMax
  }
```

The time complexity of this algorithm is $T(n) = O(n)$ since it loops once over all items in the array comparing it to the current max/min.

Both the recursive and iterative versions of the algorithm show to have the same worst case time complexity of $O(n)$. However, the iterative version has two comparisions that occur n times(size of the array). The recursive implementation has 4 comparisons that occur making it the slower choice