Thomas Haddy
Section 9 @ 11 Th
TA: Trent Muhr
2/4/18

COM S 311 HW 2

1a).

```
1     r=0
n  [  for i in the range [1, n]
n-i   [ for j in the range [i, n]
j-i      [ for k in the range [1, j-i]
1           r++
1     print (r)
```

Runtime: $1 + n[(n-i+1)(j-i+1)]$.

r=0 runs 1 time

(for i in the range [1,n]) runs n times

print(r) runs 1 time * n times b/c in for loop block

for j in the range [i, n] runs i-n times * n

for k in the range [1, j-i] runs j-i times * i-n * n

r++ runs 1 time * j-i * j-n * n

Runtime as function of n:

$$1+\sum_{i=0}^{n}\left[1+\sum_{j=i}^{n}\left(1+\sum_{k=1}^{j-i}1\right)\right] = 1+\sum_{i=0}^{n}\left[1+\sum_{j=i}^{n}\left(1+j-i\right)\right]$$

$$=1+\sum_{i=0}^{n}\left[1+\left(\frac{(i-n-2)(i-n-1)}{2}\right)\right] = \frac{1}{6}(n+1)(n^2+5n+12)$$

$$= O(n^3)$$

1b)

$\frac{n}{2}$ [for (i=n ; i >= 1 ; i/2)

i [for (j in the range [1, i]

1 Constant Number of operations

Runtime : $\frac{n}{2}[i+1] = \log_2 n [i+1]$

Runtime as a function of n : $\sum_{i=1}^{n/2}\left[\sum_{j=1}^{i} 1 + 1\right]$

$$= \sum_{i=1}^{n/2}[i+1] = \frac{n}{8}(n+6) = \log_8(n)(n+6) = O(n \log n)$$

2. a) Input: Array a of size n, consisting of ints

Worst = ○ Best = ☁

```
n-1   for i in range [1, n-1] {
n-1     1   j = i;
        j   while (j>0 AND a[j-1] > a[j]) {
        1     3   swap(a[j], a[j-1]);
              1   j = j-1;
            }
      }
```

Worst case runtime: for the while loop, we assume the second condition is always true because the worst case is every element in array a is in reverse order. So the reverse sorted array a leads to the worst-case complexity.

$$(n-1)[(1+j)(3+1)] = (n-1)[(1+j)(4)] = (n-1)[4j+4]$$

$$= \sum_{i=1}^{n-1}\left[\sum_{j=1}^{n-1} 4 + 4\right] = \sum_{i=1}^{n-1} 4(n-1)+4 = 4n(n-1) = O(n^2)$$

Best case runtime: for the while loop, we assume the second condition is always false because the best case is every element in array a is in sorted order. So the sorted array a leads to the best-case complexity.

$(n-1)[1]$ // Never goes inside while loop so ignore its contents.

$$= \sum_{i=1}^{n-1} 1 = n-1 = O(n)$$

2b)

```
Runtimes are expressed in milliseconds, or one thousandth of a second.

Sorted Array Runtime---------------------------------------------------

Selection sort runtime for an arr of 3000 size: 9
Selection sort runtime for an arr of 30000 size: 108
Selection sort runtime for an arr of 300000 size: 11500

Insertion sort runtime for an arr of 3000 size: 0
Insertion sort runtime for an arr of 30000 size: 1
Insertion sort runtime for an arr of 300000 size: 4

Bubble sort runtime for an arr of 3000 size: 0
Bubble sort runtime for an arr of 30000 size: 1
Bubble sort runtime for an arr of 300000 size: 2

Reversed Array Runtime-------------------------------------------------

Selection sort runtime for an arr of 3000 size: 11
Selection sort runtime for an arr of 30000 size: 292
Selection sort runtime for an arr of 300000 size: 130585

Insertion sort runtime for an arr of 3000 size: 23
Insertion sort runtime for an arr of 30000 size: 976
Insertion sort runtime for an arr of 300000 size: 97718

Bubble sort runtime for an arr of 3000 size: 24
Bubble sort runtime for an arr of 30000 size: 900
Bubble sort runtime for an arr of 300000 size: 96595

Random Array Runtime---------------------------------------------------

Selection sort runtime for an arr of 3000 size: 14
Selection sort runtime for an arr of 30000 size: 1321
Selection sort runtime for an arr of 300000 size: 133097

Insertion sort runtime for an arr of 3000 size: 6
Insertion sort runtime for an arr of 30000 size: 489
Insertion sort runtime for an arr of 300000 size: 49002

Bubble sort runtime for an arr of 3000 size: 8
Bubble sort runtime for an arr of 30000 size: 1478
Bubble sort runtime for an arr of 300000 size: 157011
```

Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)

So, from this information we see that the best, worst, and average cases, respectively, for selection, insertion, and bubble sort were as expected.

3 a) Is $48n^4 - 46n^2 + 25n + 31 \in O(n^4)$

Proof: Let $n > k$ and prove $f(n) \leq C\,g(n)$

So $\exists C \, \exists k \, \forall n (n > k \rightarrow f(n) \leq C\,g(n))$

Choose $k=1$. Assuming $n > 1$, we will find a C such that

$\frac{f(n)}{g(n)} \leq \frac{C\,g(n)}{g(n)} = C$. Take out the negatives and assume $n > 1$

$$\frac{48n^4 - \cancel{46n^2} + 25n + 31}{n^4} < \frac{48n^4 + 25n^4 + 31n^4}{n^4} = 104$$

Choose $C = 104$. $25n < 25n^4$, $31 < 31n^4$

Therefore, $48n^4 - 46n^2 + 25n + 31 \in O(n^4)$ because $48n^4 - 46n^2 + 25n + 31 \leq 104n^4$ for $\forall n > 1$.

b) Is $n^{\log n} \in O(2^{\sqrt{n}})$

Proof: $\exists C > 1$ such that for $\forall n \geq 1$, $n^{\log n} < C\,2^{\sqrt{n}}$

Assume $n^k \in O(n^{k+a})$, let $k = \log n$ and $C > 1$

So, $n^{\log n} \leq C \cdot n^{\log n + \sqrt{n}} \rightarrow n^{\log n} \leq C \cdot n^{\log n} \cdot n^{\sqrt{n}}$

Clearly $1 \leq C \cdot n^{\sqrt{n}}$, $C \geq 1$.

Therefore, $n^{\log n} \in O(2^{\sqrt{n}})$ because $n^{\log n} < 1 \cdot 2^{\sqrt{n}}$ for $\forall n \geq 1$.

c) Is $2^{2^{n+1}} \in O(2^{2^n})$

Proof: Assume for contradiction, that $2^{2^{n+1}} \leq C \cdot 2^{2^n}$ for $C > 1$.

So, $2^{2^{n+1}} = 2^{2 \cdot 2^n} = 2^{2^n \cdot 2} \leq C \cdot 2^{2^n}$.

Let $x = 2^{2^n}$. So $x^2 \leq C \cdot x$. This is impossible because $x^2 \geq Cx$.

Therefore, $2^{2^{n+1}} \notin O(2^{2^n})$ for $\forall n \geq 1$.

d) Is $n^3(5+\sqrt{n}) \in O(n^3)$

Proof: Assume $\exists C \exists k \forall n (n > k \rightarrow f(n) \leq C g(n))$

Choose $k=1$. Assuming $n > 1$, we find a C such that

$\frac{f(n)}{g(n)} \leq \frac{C g(n)}{g(n)} = C$ $\quad * \; n^3(5+\sqrt{n}) = 5n^3 + n^{3.5} \; *$

$\frac{5n^3 + n^{3.5}}{n^3} \leq \frac{5n^{3.5} + n^{3.5}}{n^3} = C$. This is not possible because $f(n) > g(n) \rightarrow n^{3.5} > n^3$.

Therefore since no constant C can satisfy the above equation, $n^3(5+\sqrt{n}) \notin O(n^3)$ for $\forall n \geq 1$

4. a) Input: Array d of size n, int base k

```
(1)  int total = 0;
(n)  for i in the range [0, n-1]   //exclude n
       (n)   total = total + (d[n-i-1] * k^i);
     }
(1)  return total;
```

Runtime: $1 + n(n) + 1 = n^2 + 2 = O(n^2)$

4b) Input: int decimal m, int base k

```
(1)    int temp = m;
(1)    int length = 0;
(n/k)  while temp > 0
         (1) temp = temp / k;
         (1) length = length + 1;
       }
(1)    int total = 0;
(n)    while length > 0
         (n)  total = total + (m % k) * 10^(length-1)
         (1)  length = length - 1;
         (1)  m = m / k;
       }
(1)    return total;
```

Runtime: Note that n = size of m. So length = n. k is constant b/c $2 \le k \le 9$

$1 + 1 + \frac{n}{k}(1+1) + 1 + n(n+1+1) + 1 = n^2 + 2n + \log_k n + 4 = n^2 + 2n + \log_k n + 4$

$= O(n^2)$