Chad Hirsch

CS4310 Assignment 1: Algorithm Complexity

1. For each of the code snippets below,
   1. Derive the time complexity using summation notation,
   2. Find the close form
   3. Express it using tight asymptotic (big-Oh, big-Theta, big-Omega) notation.

   Assume constant overhead per iteration of a loop.
   Keep details of your calculation until the last step of expressing your solution using tight asymptotic notation.

```
a. sum = 0;
   for(i = 1; i ≤ n; i++)
       for(j = (n/2); j ≥ 1; j--)
           sum++;
```

$\sum_{i=1}^{n}(\sum_{j=\frac{n}{2}}^{1} 1) = \sum_{i=1}^{n}(1 + 1 + \cdots (n/2\ times) \ldots + 1) = \sum_{i=1}^{n} \frac{n}{2}$

$\sum_{i=1}^{n} \frac{n}{2} = \sum_{i=1}^{n}\left(\frac{n}{2} + \frac{n}{2} + \cdots (n\ times) \ldots + n/2\right) = n\left(\frac{n}{2}\right) = \frac{n^2}{2}$

$T(n) = \frac{n^2}{2}$

$O(n^2),\ Omega(n^2)\ , Theta(n^2)$

```
b. for (i = 1; i < n; i++) {
       y = y + 1;
       for (j = 0; j ≤ 2n; j++)
           x++;
   }
```

$\sum_{i=1}^{n}(\sum_{j=0}^{2n} 1) = \sum_{i=1}^{n}(1 + 1 + \cdots (2n\ times) \ldots + 1) = \sum_{i=1}^{n} 2n$

$\sum_{i=1}^{n} 2n = \sum_{i=1}^{n}(2n + 2n + \cdots (n\ times) \ldots + 2n) = n\,(2n) = 2n^2$

$T(n) = 2n^2$

$O(n^2),\ Omega(n^2)\ , Theta(n^2)$

```
c. for (i = 1; i ≤ n; i++)
       for (j = 1; j ≤ i; j++)
           for (k = j; k ≤ n; k++)
               x = x + 2;
```

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

$\sum_{i=1}^{n}\left(\sum_{j=1}^{i}(n-j+1)\right)=\sum_{i=1}^{n}(n+(n-1)+(n-2)+\cdots+2+1)$

$\sum_{i=1}^{n}(n+(n-1)+(n-2)+\cdots+2+1)=\sum_{i=1}^{n}\frac{n(n+1)}{2}=n(\frac{n(n+1)}{2})$

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

$O(n^3),\ Omega(n^3)\ ,Theta(n^3)$

```
d. for (i = 1; i ≤ n; i += 2);
```

$\sum_{i=1}^{n/2}1=1+1+\cdots\left(\frac{n}{2}\ times\right)\ldots+1=\frac{n}{2}$

$T(n)=\frac{n}{2}$

$O(n),\ Omega(n),\ Theta(n)$

```
e. for (i = 1; i ≤ n; i *= 2)
       x = y+2;
```

$\sum_{i=1}^{log_2 n}1=1+1+\cdots(log_2 n\ times)\ldots+1=\ log_2 n$

$T(n)=\ log_2 n$

$O(\log n),\ Omega(\log n)\ ,Theta(\log n)$

2. This question concerns the asymptotic relations between functions; you can assume that all logarithmic functions are in base 2. Sort the following functions in an asymptotically **non-decreasing** order of growth using big-oh and big-theta notations. Again, **you must justify your answers**, otherwise no credit.

- $2^{10000}=O(1)$
    - $\lim_{n\to\infty}\frac{2^{10000}}{2\log(n)}=0$
- $\log(n^2)=2\log(n)=\ O(\log(n))$
    - $\lim_{n\to\infty}\frac{2\log(n)}{\log(n)^2}=\lim_{n\to\infty}\frac{2}{\log(n)}=0$
- $(\log n)^2=\ O(\log(n)^2)$
    - $\lim_{n\to\infty}\frac{\log(n)^2}{n}=\lim_{n\to\infty}\frac{\frac{2}{nln(2)}}{1}=\lim_{n\to\infty}\frac{2}{nln(2)}=0$
- $\sqrt{4}^{log\ n}=\ 2^{log\ n}=n=O(n)$
    - $\lim_{n\to\infty}\frac{n}{2nlog(n)}=\lim_{n\to\infty}\frac{1}{2log(n)}=0$
- $2n\log n=\frac{2\ n\log(n)}{\log(2)}=O\big(nlog(n)\big)$

- $\lim_{n\to\infty} \frac{2nlog(n)}{n^3} = \lim_{n\to\infty} \frac{2log(n)}{n^2} = \lim_{n\to\infty} \frac{\frac{2}{nln(2)}}{n} = \lim_{n\to\infty} \frac{2}{n^2\ln(2)} = 0$

- $8^{\log n} = 2^{3\log n} = n^3 = O(n^3)$
  - $\lim_{n\to\infty} \frac{n^3}{n^4} = \lim_{n\to\infty} \frac{1}{n} = 0$
- $n^4 = O(n^4)$
  - $\lim_{n\to\infty} \frac{n^4}{n2^n} = \lim_{n\to\infty} \frac{n^3}{2^n} = \lim_{n\to\infty} \frac{3n^2}{n2^n} = \ldots = \lim_{n\to\infty} \frac{6}{\ln^3(2)2^n} = 0$
- $n2^n = O(n2^n)$
  - $\lim_{n\to\infty} \frac{n2^n}{2^{6n}} = \lim_{n\to\infty} \frac{n}{32^n} = \lim_{n\to\infty} \frac{1}{5\ln(2)32^x} = 0$
- $2^{(6n)} = 64^n = O(64^n)$
  - For n =1,000,000
  - $64^{1,000,000} = 64 \times 64 \times 64 \times 64 \ldots (1{,}000{,}000\ times)$
  - (1,000,001)(1,000,000)! = $1{,}000{,}001 \times 1{,}000{,}000 \times 999{,}999 \times 999{,}998 \times 999{,}997 \times \ldots$
    - Much greater than $64^{1,000,000}$
  - $\lim_{n\to\infty} \frac{64^n}{(n+1)n!} = 0$
- $(n+1)n! = O((n+1)n!)$
  - Take log of both $(n+1)n!)\ and\ 2^{(4^n)}$
  - $\lim_{n\to\infty} \frac{\log((n+1)n!)\ )}{\log(2^{(4^n)})} = \lim_{n\to\infty} \frac{\log(n+1)+\log(n!)}{4^n} = 0$
- $2^{(4^n)} = O\left(2^{(4^n)}\right)$

3.

a. Design and analyze an algorithm to find the median, upper quartile and lower quartile from a list of n integers (unordered). Your time complexity should be no more than $O(n \log n)$.

i. We will use a helper method called find_median to find the middle element between two end points. If there are an even number of elements between the end points, then the median is the average of the two middle elements. This helper method will be used to calculate the median, upper quartile and lower quartile. In the find quartiles method, we will first use mergesort to sort our list. Then we will call find_median to find the median. When calculating the upper and lower quartiles we need to see if the median is included in our calculation. We can see if the median is in the sorted_list by using modulus. If the median was an average of two elements, it will not be in the sorted_list, so we should include the n /2 element when finding the upper bound. If the median is in the sorted_list, we should exclude the n/2 element when finding the upper bound.

Pseudocode:

```
find_median(array, left, right):
        int n = right – left +1
        if (n modulus 2 == 0)
                x = array[(right – left)/2 + left )
                y = array[(right – left)/2 + left +1)
                return (double) ((x+y) /2
        else
```

```
                return array[(right – left)/2 + left )

findQuartiles(unsorted_list):
        sorted_list = mergesort(unsorted_list)
        n = sorted_list.length
        median = find_median(array, 0, n -1)
        if(n % 2 != 0)  //median is an element in the list
                lower_quartile = find_median(sorted_list, 0, n/2 -1)
                upper_quartile = find_median(sorted_list, n/2 + 1, n-1)
        else
                lower_quartile = find_median(sorted_list, 0, n/2 -1)
                upper_quartile = find_median(sorted_list, n/2, n-1)
```

i. Time / Space complexity
   1. Merge sort has a time complexity of O(n log n). Finding the median, upper quartile, and lower quartile takes O(1) time complexity. Overall, this algorithm has a time complexity of O(n log n). Since no additional space is used, this algorithm has a space complexity of O(1).

b. Can you improve your algorithm if the input list is sorted? If so, design and analyze such an algorithm. If not, justify why no improvement in time/space complexity can exist.
   i. If the list is already sorted, then there is no need to use merge sort as done previously. The rest of the code will be the same as above.

Pseudocode:

```
find_median(array, left, right):
        int n = right – left +1
        if (n modulus 2 == 0)
                x = array[(right – left)/2 + left )
                y = array[(right – left)/2 + left +1)
                return (double) ((x+y) /2
        else
                return array[(right – left)/2 + left )

findQuartiles(sorted_list):
        n = sorted_list.length
        median = find_median(array, 0, n -1)
        if(n % 2 != 0)  //median is an element in the list
                lower_quartile = find_median(sorted_list, 0, n/2 -1)
                upper_quartile = find_median(sorted_list, n/2 + 1, n-1)
        else
                lower_quartile = find_median(sorted_list, 0, n/2 -1)
                upper_quartile = find_median(sorted_list, n/2, n-1)
```

ii. Time / Space Complexity
   1. Since mergesort was not used we need to analyze the rest of the operations done in the algorithm. Find_median works in O(1) time, so finding the median, upper quartile and lower quartile takes O(1) time

each. Overall, the algorithm has O(1) time complexity. Since no additional space is used, this algorithm also has a space complexity of O(1).

4.

a. Algorithm A uses $(100n \log n)$ operations, while algorithm B uses n^2 operations. Determine the value n_0 such that A is better than B for $n \geq n_0$.
   i. $100n \log n < n^2$
   ii. $100 \log n < n$
   iii. $100 < n / (\log n)$
   iv. Assuming the logarithmic function is in base 2, the above statement is true when $n_0 = 997$. For $n = 997$, $n / (\log n) = 100.085 > 100$.

b. Are there values of n such that B is better than A? If yes, list the ranges. If not, justify why not.
   i. If $1 \leq n < 997$, then B is better than A.

c. Determine the value n_0 such that A is better than B for $n \geq n_0$, when B uses $n^{1.5}$ operations.
   i. $100n \log n < n^{1.5}$
   ii. $100 \log n < n^{0.5}$
   iii. $100 < n^{0.5} / (\log n)$
   iv. Assuming the logarithmic function is in base 2, A is better than B for $n \geq n_0$ when $n_0 = 4{,}945{,}094$. For $n = 4{,}945{,}094$, $n / (\log n) = 100.0000036 > 100$.

5. (20pts) Solve the following recurrence relation:

$$T(n) = \begin{cases} 2 * T(n/2) + 4n^2, & if\ n \geq 2 \\ 3, & otherwise \end{cases}$$

Find a closed form of $T(n)$ and then express your solution using tight asymptotic notation.

$T(2) = 2\,T(1) + 4(2)^2 = 2[\,T(1) + 2(2)^2] = 2[3 + 2^3]$
$T(4) = 2\,T(2) + 4(4)^2 = 2[\,T(2) + 2(2^2)^2] = 2[(2[3 + 2^3] + 2^5) = 4[\,3 + 2^3 + 2^4]$
$T(8) = 2T(4) + 4(8)^2 = 2[\,T(4) + 2(2^3)^2] = 2[(4[\,3 + 2^3 + 2^4]) + 2^7] = 8[3 + 2^3 + 2^4 + 2^5]$

$T(n) = n(3 + \sum_{i=3}^{log_2(n)+2} 2^i\,)$

$O\left(2^{log_2(n)+2}\right) = O\left(2^{log_2(n)} * 2^2\right) = O(n * 4) = O(n)$