Brendan Jang
CS 325
Homework 2

1. A) $T(n) = T(n - 1) + 2n$
$= [T(n - 2) + 2(n - 1)] + 2n$
$= T(n - 2) + 2n + 2n – 2$
$= [T(n - 3) + 2(n - 2)] + 2n + 2n – 2$
$= T(n - 2) + 2n + 2n + 2n – 6$
$= [T(n – 4) + 2(n – 3)] + 2n + 2n + 2n – 6$
$= T(n – 4) + 2n + 2n + 2n + 2n – 12$

$T(n – i) + i(2n) + i(i – 1)$
$T(1) = 1$ so find $n – i = 1$
$i = n – 1$
$T(1) + (n – 1)(2n) + (n – 1)(n – 1 – 1)$
$= T(1) + (2n^2 – 2n + n^2 – n)$
$= 1 + 3n^2 – 3n$
$= 1 + 3(n^2 – n) = \Theta(n^2)$

B) $T(n) = T(n – 2) + 3$
$= [T(n – 4) + 3] + 3$
$= T(n – 4) + 3(2)$
$= [T(n – 6) + 3] + 3(2)$
$= T(n – 6) + 3(3)$
$= [T(n – 8) + 3] + 3(3)$
$= T(n – 8) + 3(4)$

$T(n – 2i) + 3(i)$
$T(1) = 1$ so find $n – 2i = 1$
$i = (n – 1)/2$
$T(1) + 3((n -1) / 2)$
$= 1 + (3n – 3) / 2$
$= 3(n-1) / 2 + 1$
$= 3/2 * (n – 1) + 1 = \Theta(n)$

C) $T(n) = 4T(n/2) + 3n^2$
$= 4[4T(n/2^2) + 3(n/2)^2] + 3n^2$
$= 4^2 T(n/2^2) + 6n^2$
$= 4^2 [T(n/2^2) + 3(n/2^2)^2] + 6n^2$
$= 4^3 T(n/2^3) + 9n^2$

$4^i T(n/2^i) + 3i(n^2)$
$T(1) = 1$ so find $n/2^i = 1$

n = 2^i
log_2(n) = i
4^log_2(n) T(1) + (3log_2(n)) * (n^2)
= n^log_2(4) + (3log_2(n)) * (n^2)
= n^2 + (3n^2)(log_2(n)) = Θ(n^2 log n)

D) T(n) = 2T(n/4) + n^2
This is in the form of T(n) aT(n/b) + f(n) so we can use the master theorem.
We compare n^log_b(a) with f(n).
a = 2, b = 4
n^log_b(a) = b^log_4(2) = n^.5 = sqrt(n)
n^log_b(a) = b^log_4(2) = n^.5 = sqrt(n) < f(n) = n^2
This comes under case 3 so the complexity is Θ(n^2).

2. A) Set a base case for when the index range is only 1 element. Get the midpoint and use it to find a lower midpoint index and upper midpoint index

```
// base case
quaternarySearch(arr[], start, end, val)
    if (start == end)
        return false

// get mid points
mid = (start + end) / 2
lowMid = (start + mid / 2
upMid = (mid + end) / 2

// check to see if mid index are equal to target value
if (arr[mid] or arr[lowMid] or arr[upMid] == val)
    return true

// check the first quarter
else if (val < arr[lowMid])
    return quaternarySearch(arr, lowMid + 1, mid, val)

// check the second quarter
else if (val < arr[mid] && val > arr[lowMid])
    return quaternarySearch(arr, start, lowMid, val)

// check the third quarter
else if (value > arr[mid] && value < arr[upMid])
    return quanternarySeach(arr, mid + 1, upperMid, value)
```

```
// check last quarter
else
      return quaternarySearch(arr, upMid + 1, end, val)
```

B) The recurrence for the quaternary search algorithm is:

$T(n) = T(n/4) + c$

C) Using the master method we get:

a = 1, b = 4, f(n) = c
n^log_b (a) = n^log_4 (1) = n^0
N^0 = 1 = Θ(1) = Θ(c)
Therefore, case 2 would apply
T(n) = Θ(n^log_b (a) * log_4 (n)) = Θ(1 * log_4(n))
Therefore, the run time for quaternary search is T(n) = Θ(log_4 (n))

The worst case run time for the quaternary search is technically equivalent to that of the binary search. The worst case for binary search is T(n) Θ(log_2(n)). The run times of logarithms with different bases are equivalent in asymptotic analysis. Therefore, Θ(log_2(n)) = Θ(log_4(n))

3. A) The recurrence for the number of comparisons executed by STOOGESORT would be as follows.

   T(n) = a * T(n/b) + f(n)

B) Using the master method:
a = 3, b = 3/2, f(n) = c
n^log_b (a) = n^log_3/2 (3)
n^log_3/2 (3) = (log_10 (3) / log_10 (3/2)) = 2.71
n^log_3/2 (3) = n^2.61 = Ω(f(n)) because f(n) = c where c is a constant
f(n) = O(n^(2.71 - c))
Case 1 would apply so the run time for the recurrence would be as follows T(n) = Θ(n^2.71).

4. A) File upload

B)

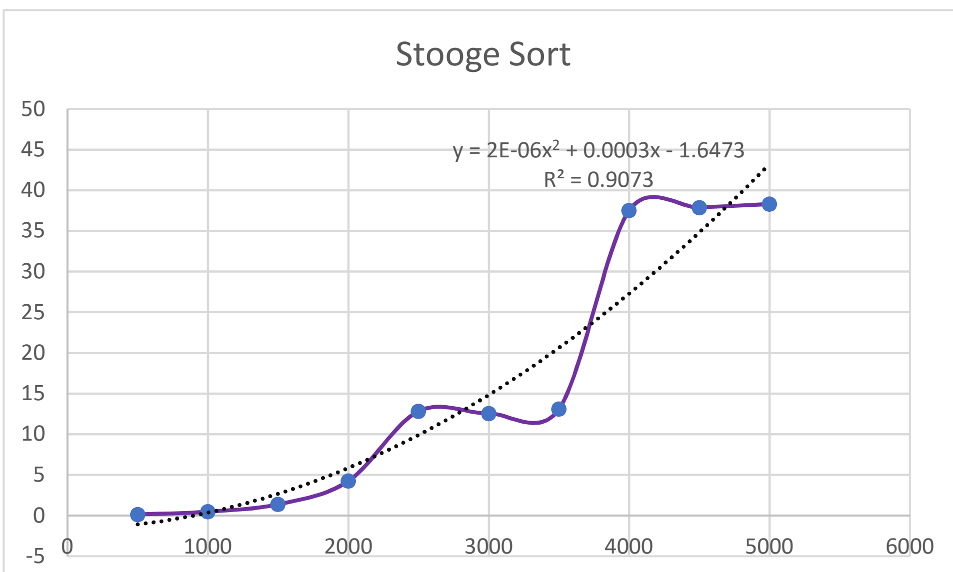
```
#include <iostream>
#include <string>
#include <fstream>
#include <ctime>

using namespace std;

// modified stooge sort method
void stoogeSort(int arr[], int first, int last) {
	int size = last - first;

	// base case
	if (size == 2) {
		// check to see if we need to swap
		if (arr[first] > arr[last]) {
			int temp = arr[first];
			arr[first] = arr[last];
			arr[last] = temp;
		}
	}
	else if (size > 2) {
		// get the index of 2/3rd of our array
		int val = size / 3;

		// recursively sort the first 2/3rd
		stoogeSort(arr, first, last - val);

		// revursively sort the second 2/3rd
		stoogeSort(arr, first + val, last);

		// recursive call on the first 2/3rd to verify
		// if there was a swap in the second call
		stoogeSort(arr, first, last - val);
	}
}

int main() {
	for (int i = 0; i < 10; i++) {
		srand(time(0));
		int size;
		int* list;
```

```
		cout << "Enter a number between 500 and 5000: " << endl;
		cin >> size;

		// allocate memory for our dynamic array
		list = new int[size];

		// get elements for  dynamic array
		for (int x = 0; x < size; x++) {
			int temp = rand() % 10001;
			list[x] = temp;
		}

		// Sort values and output the time
		clock_t start;
		start = clock();
		stoogeSort(list, 0, size-1);
		start = clock() - start;
		cout << endl << "It took " << double(start) / CLOCKS_PER_SEC << " seconds to 
sort " << size << " elements." << endl << endl;

		// deallocate
		delete[] list;
	}
	return 0;
}
```

C)

stooge sort	
n	time
500	0.15
1000	0.49
1500	1.4
2000	4.26
2500	12.84
3000	12.57
3500	13.11
4000	37.53
4500	37.89
5000	38.31

D)



E) To get the theoretical running time I solved the recurrence using the master method. The theoretical run time or stooge sort that I found above was Θ(n^2.71). My results show that the experimental run time is on par with the theoretical run time.

F)