Andrew Jordan

Problem Set #2 Working with Big-Oh

Homework Problem #2

CS 472

By definition of Big O, $f(n) \leq c*g(n)$ for $n > n_0$

Claim: $n^2 + 3n^3 \in O(n^3)$

This implies that: $n^2+3n^3 \leq c*g(n^3)$ for $n > n_0$.

If $n^2+3n^3 \leq c*g(n^3)$ for $n > n_0$ then $\frac{1}{n} + 3 \leq c$. Therefore, the Big-O condition holds for $n \geq n_0 = 1$ and $c = 4$.

By definition of Big Ω, $f(n) \geq c*g(n)$ for $n > n_0$

Claim: $n^2 + 3n^3 \in \Omega(n^3)$

This implies that: $n^2+3n^3 \geq c*g(n^3)$ for $n > n_0$.

If $n^2+3n^3 \geq c*g(n^3)$ for $n > n_0$ then $\frac{1}{n} + 3 \geq c$. Therefore, the Big-O condition holds for $n \geq n_0 = 1$ and $c \leq 4$.

Therefore: $n^2 + 3n^3 \in \Theta(n^3)$.

Output of 100 sorts of each type of data

5

0.0070004

10

0.0070004

50

0.0070004

100

0.00837048

500

0.00806046

1000

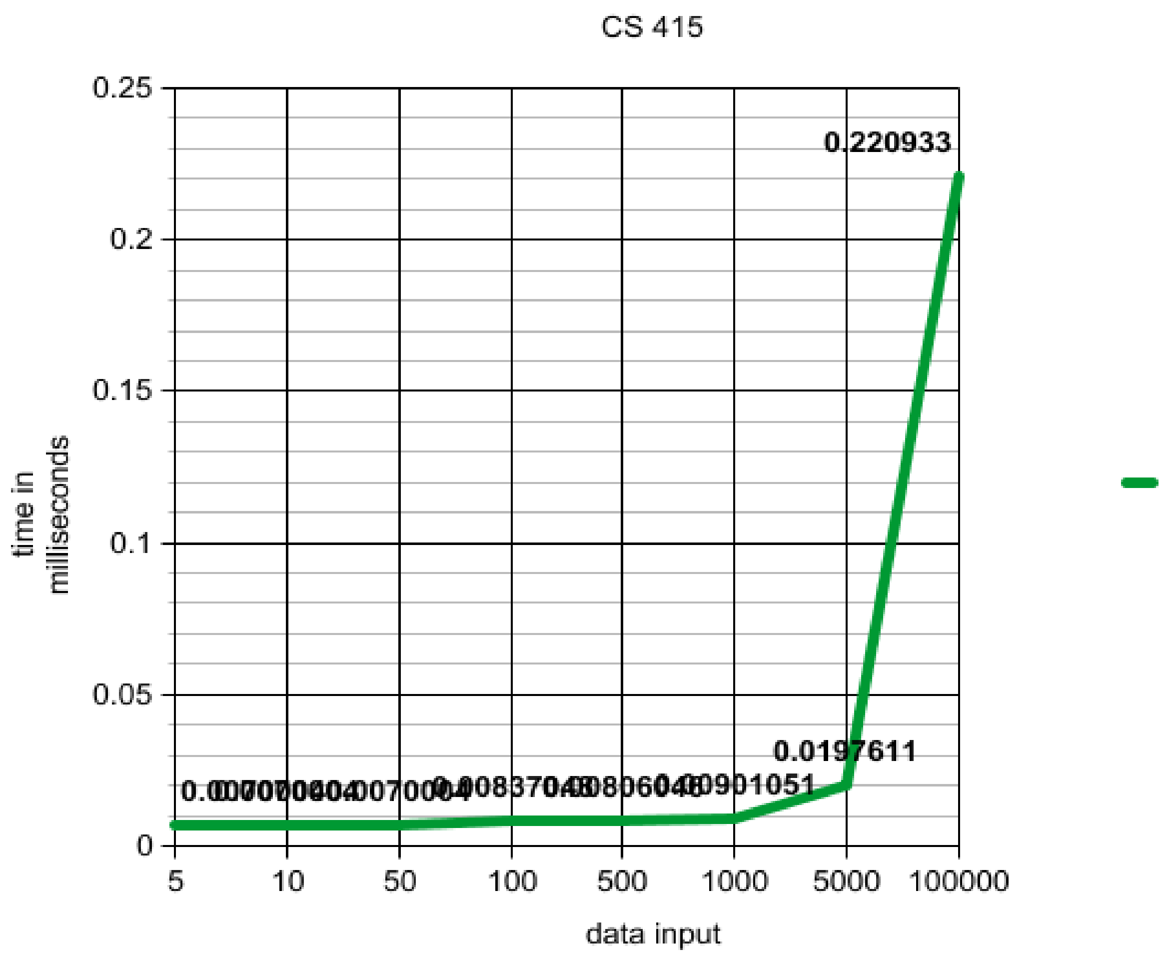
0.00901051

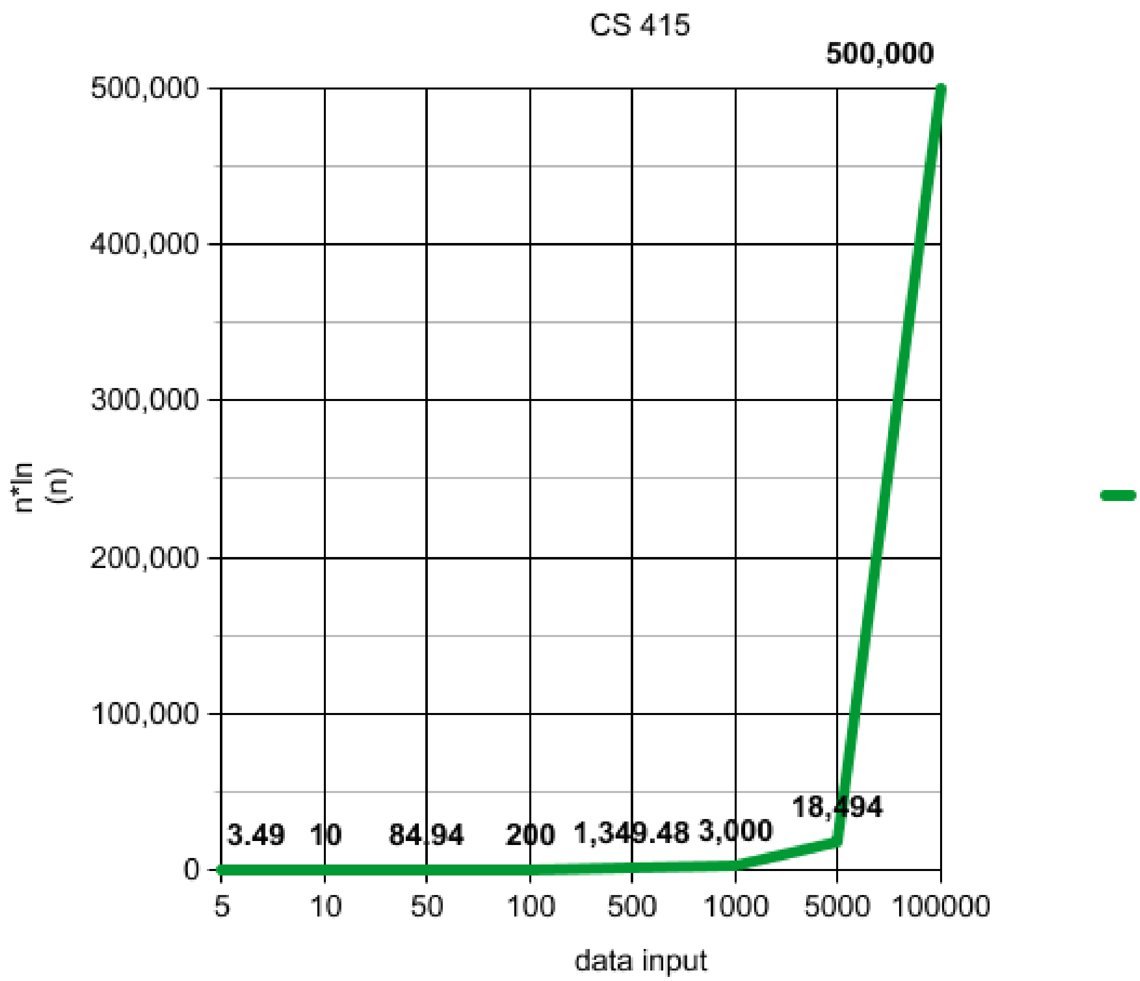
5000

0.0197611

100000

0.220933






```cpp
#include <iostream>
#include <array>
#include <algorithm>
#include <chrono>
#include <thread>
#include<fstream>
using namespace std;

class Timer{
public:
	Timer() : beg_(clock_::now()) {}
	void reset() { beg_ = clock_::now(); }
	double elapsed() const {
		return std::chrono::duration_cast<second_>
			(clock_::now() - beg_).count();
	}
private:
	typedef std::chrono::high_resolution_clock clock_;
	typedef std::chrono::duration<double, std::ratio<1>> second_;
	std::chrono::time_point<clock_> beg_;
};

void mySleep(unsigned long timeInSeconds)
```

```cpp
{
	std::chrono::milliseconds timeInMS(timeInSeconds);
	std::this_thread::sleep_for(timeInMS);
}
void doSomething() { mySleep(3); };
void doSomeMoreWork() { mySleep(4); };

template<std::size_t SIZE>
void fill(std::array<int, SIZE> &arry,int max)
{
	arry.empty();
	for (int i = 0; i < max; i++)
		arry[i] = rand() % 1000 + 1;

}
template<std::size_t SIZE>
double test(std::array<int, SIZE> &arry, int max)
{
	double sumtime = 0;
	Timer tmr;
	for (int i = 0; i < 100; i++)
	{
		fill(arry, max);
		// Start time
		tmr.reset();
		doSomething();
		double t = tmr.elapsed();
		//std::outfile << t << std::endl;
		// sort array
		std::sort(arry.begin(), arry.end());
		// find endtime
		doSomeMoreWork();
		t = tmr.elapsed();
		sumtime += t;
		//std::outfile << t << std::endl;
	}
	return sumtime / 100;
}

int main()
{
	ofstream outfile;
	outfile.open("results.txt");

	std::array<int, 5> five;
	std::array<int, 10> ten;
	std::array<int, 50>  fifty;
	std::array<int, 1000>  onehundred;
	std::array<int, 500>  fivehundred;
	std::array<int, 1000>  onek;
	std::array<int, 5000>  fivek;
	std::array<int, 100000>  onehundredk;


	outfile << "5" << endl;
	outfile << test(five, 5) << endl;
	outfile << "10" << endl;
	outfile << test(ten, 10) << endl;
```

```
	outfile << "50" << endl;
	outfile << test(fifty, 50) << endl;
	outfile << "100" << endl;
	outfile << test(onehundred, 100) << endl;
	outfile << "500" << endl;
	outfile << test(fivehundred, 500) << endl;
	outfile << "1000" << endl;
	outfile << test(onek, 1000) << endl;
	outfile << "5000" << endl;
	outfile << test(fivek, 5000) << endl;
	outfile << "100000" << endl;
	outfile << test(onehundredk, 100000) << endl;

	cout << "finished" << endl;

	return 0;
}
```

Conclusion:

For this project, one-hundred sets of randomly generated data for each data group ranging from five to one-hundred thousand were sorted by the function sort() provided by the C++ software library. A graph was generated from the C++ data based on the average amount of time it took to sort each set. The graphs Y-axis represents the time it took to sort the data and the X-axis represents the amount of data inputted. After the graph was generated, another graph was generated showing n*log(n). The n*log(n) graph displays the amount of data on the X-axis and the equation n*log(n) on the Y-axis. The library sort() claims to be O(n*log(n)) and after comparisons of the two graphs, it appears that the sort() function in C++ has a BigO of n*log(n), that is for small data sets, the sort time is roughly the same but as the data size continues to increase, the time to sort that data increases dramatically.