Sangyun Lee

CS325 HW1

Question 1

1. Base: when N is 2, then T(2) = 2 lg 2
2. Inductive Hopothesis:
    a. Assume that T(n) = n lg n is true if n=2^k, for k>1
    b. Then it will be true when 2^(k+1)
3. Inductive case:
    a. $\mathrm{T}\left(2^{\mathrm{k}+1}\right) = 2T\left(\frac{2^{\mathrm{k}+1}}{2}\right) + 2^{\mathrm{k}+1}$
    b. $= 2T(2k) + 2^{\mathrm{k}+1}$
    c. $= 2 * 2^k \lg 2^k + 2 * 2^{\mathrm{k}}$
    d. $= 2 * 2^k (\lg 2^k + 1)$
    e. $= 2^{k+1} \lg 2^k + lg 2$
    f. $= 2^{k+1} \lg 2^{k+1}$

Question 2

a. $\mathrm{f(n)} = \mathrm{n}^{0.25}$ and $\mathrm{g(n)} = \mathrm{n}^{0.5}$
    - when n equals to 2 or is larger than 2
    - then $\mathrm{f(n)} = \mathrm{O(g(n))}$
    - Because when n = 2, f(2) = 1.18920…., and g(2) =1.4142….
b. f(n) = b and $\mathrm{g(n)} = \log^2 \mathrm{n}$
    - when n is getting to infinity
    - Then, $\mathrm{n}/\log^2 n = (\sqrt{n}/\log N)^2$
    - Log N is growing slower than $\sqrt{n}$
    - So, f(n) is growing faster so, It is goes to infinity
    - <u>Therefore, $\mathrm{f(n)} = \Omega(\mathrm{g(n)})$</u>
c. f(n) = log n and g(n) = ln n
    - Based on the wolframalpha calculation log n = ln n
    - So, when n goes to infinity = log n / log n
    - F(n) is $\theta(\mathrm{g(n)})$ Or
    - If log's base is 10, and ln base is e
    - Then I could say, $\log_{10}\mathrm{n} / \log_e \mathrm{n}$
    - $= \frac{\mathrm{logn}}{\frac{1}{\frac{logn}{loge}}}$ = (log n*log e) / log n = log e = 0.4342…. which is constant
    - <u>Therefore $\mathrm{f(n)} = \theta(\mathrm{g(n)})$</u>

d. $F(n) = 1000n^2$ and $g(n) = 0.000n^2 – 1000n$

- Since low order terms are not important
- And also the constant value is not important as well
- So, I can say f(n) = n^2 and g(n)^2
- <u>Therefore f(n) = θ(g(n))</u>

e. $F(n) = n \log n$ and $g(n) = n\sqrt{n}$

- $n \log n/n*n^{1/2}$
- if n goes to infinity, both sides go to infinity
- Here we need to apply hospital rule, so n approach some number c
- Now we can write it as f'(n) and g'(n)
- Then derivate both side
- $= 1/n$ / $½*n^{-1/2}$
- $= 2/n^{1.5}$
- When n goes to infinity, it is equal to 0
- <u>Therefore f(n) = O(g(n))</u>

f. $F(n) = e^n$ and $g(n) = 3^n$

- $e^n/3^n = (e/3)^n$
- So, when n goes to infinity, it goes to infinity
- Which mean f(x) is grow faster than g(x)
- <u>Therefore f(n) = Ω(g(n))</u>

g. $F(n) = 2^n$ and $g(n)=2^{n+1}$

- 2^n/2*2^n = ½ which is constant
- <u>Therefore f(n) = θ(g(n))</u>

h. F(n) = 2^n and g(n) = $2^\wedge 2^n$

- 2^n / $2^\wedge 2^n$ = $2^\wedge(n-2^n)$
- When we need to figure out where n is growing faster or 2^n is growing faster
- N / 2^n both goes to infinity, so apply Lhopital rule
- $= 1/2^n*\ln 2$
- So, it goes to 0
- Then $2^\wedge(n-2^n) = 0$
- <u>So, f(n) = O(g(n))</u>

i. $F(n) = 2^n$ and $g(n) = n!$
   - When n is larger than 3
   - N! goes faster than 2^n
   - So, $f(n) = O(g(n))$

j. $F(n) = \lg n$ and $g(n) = \sqrt{n}$
   - $\mathrm{Log}_{10}\, n / \sqrt{n}$
   - Then denominator will be infinity
   - And the numerator will be a constant
   - Constant devided by infinity is equal to 0
   - Then $f(n) = O(g(n))$

Question 3

By definition of θ, we need to show

- if n goes to infinity, (n+a)^b / n^b
- By rule of the exponent, (n+a/n)^b = (1 + a/n)^b
- From the question we know b > 0, so let say b = 1
- Then, it is 1 + a/n.
- N goes close to 0, a/n become 0. Consequently, 1 + 0 = 1;
- $\lim_{n\to\infty}\left(1+\frac{a}{n}\right)^b = 1$ which is constant
- So, we can say f(n) = θ(g(n))

Question 4: Upload the code on TEACH site

Question 5: Modify code and work on the questions

a. Modified code

-------------------------mergeTime.cpp -------------------------

```
std::srand(time(NULL));

std::vector<std::vector<int> >myNumber;

int size = 7;

int n = 600;
```

```
	for (int i = 0; i < size; i++)
	{
		std::vector<int> temp;
		for (int j = 0; j < n; j++) {
			int num = rand() % n;
			temp.push_back(num);
		}
		myNumber.push_back(temp);
		n = n * 2;
	}

	for (int i = 0; i < myNumber.size(); i++)
	{
		auto start = high_resolution_clock::now();
		myNumber[i] = mergeSort(myNumber[i]);
		auto stop = high_resolution_clock::now();
		auto duration = duration_cast<microseconds>(stop - start);
		std::cout << "Size of vector array: " << myNumber[i].size() << std::endl;
		std::cout << "Execution time: " << duration.count() << " ms" << std::endl <<
std::endl;
	}
		------------------------------insertTime.cpp-----------------------------------
	std::srand(time(NULL));
	std::vector<std::vector<int> >myNumber;
	int size = 7;
	int n = 600;

	for (int i = 0; i < size; i++)
	{
```

```
        std::vector<int> temp;
        for (int j = 0; j < n; j++) {
            int num = rand() % n;
            temp.push_back(num);
        }
        myNumber.push_back(temp);
        n = n * 2;
    }

    for (int i = 0; i < myNumber.size(); i++)
    {
        auto start = high_resolution_clock::now();
        myNumber[i] = insertionSort(myNumber[i]);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop - start);
        std::cout << "Size of vector array: " << myNumber[i].size() << std::endl;
        std::cout << "Execution time: " << duration.count() << " ms" << std::endl <<
std::endl;
    }
```

b. Collect running times

i. Mergesort execution time

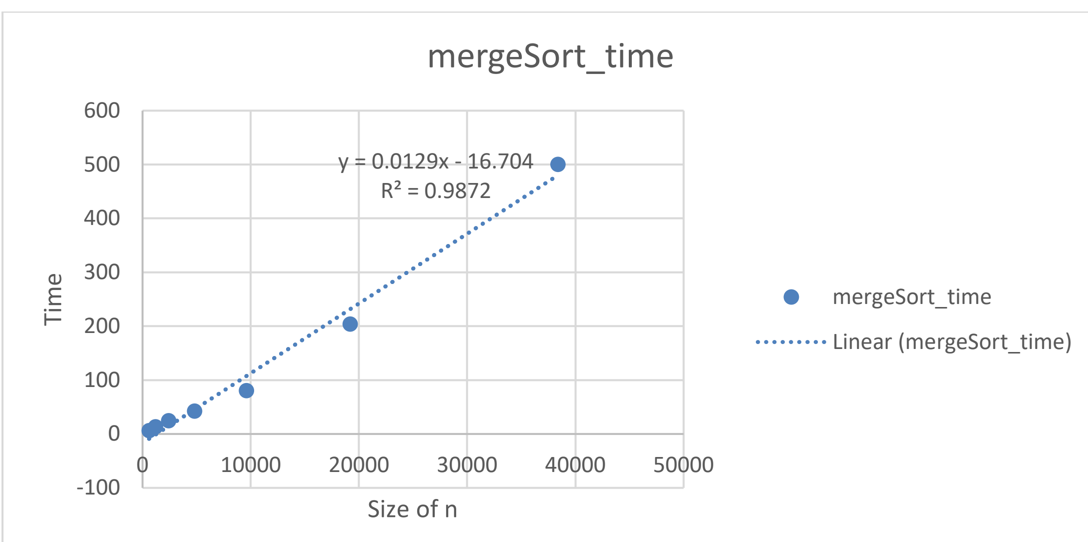
size	m_try1	m_try2	m_try3	avg	ms to sec
600	4810	8439	4704	5984.3333	5.9843333
1200	10102	18342	10132	12858.667	12.858667
2400	16471	35334	21209	24338	24.338
4800	32411	47938	45559	41969.333	41.969333
9600	70046	69429	100856	80110.333	80.110333
19200	199213	176660	235168	203680.33	203.68033
38400	497807	469488	533650	500315	500.315

ii. Insertsort execution time

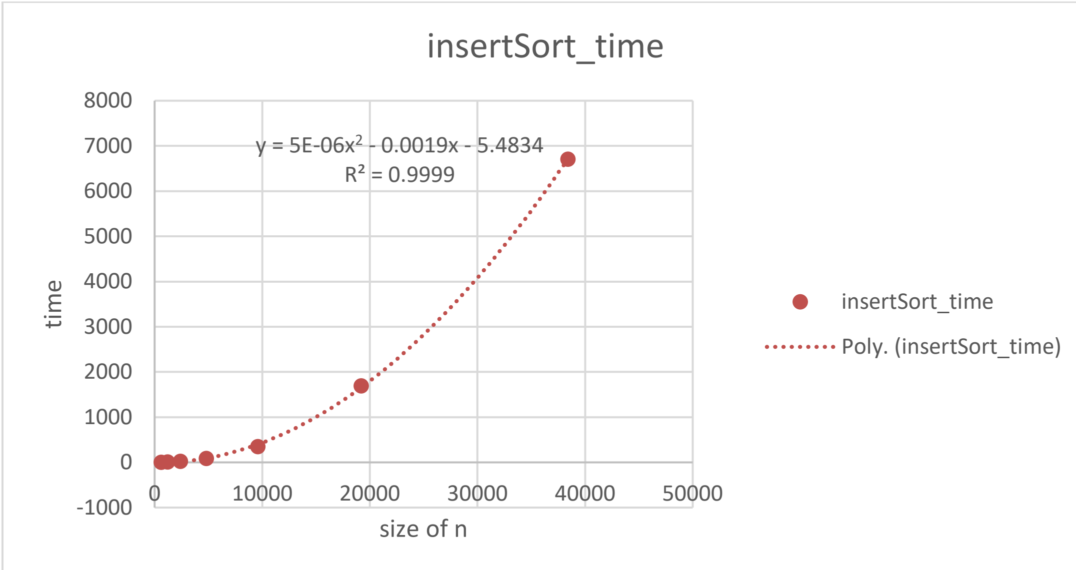
size	i_try1	i_try2	i_try3	avg	ms to sec
600	1729	1765	1900	1798	1.798
1200	7256	7106	7208	7190	7.19
2400	27038	21242	27851	25377	25.377
4800	88950	88593	89150	88897.667	88.897667
9600	422462	317470	302955	347629	347.629
19200	1889581	1653698	1535320	1692866.3	1692.8663
38400	7130488	6346010	6646496	6707664.7	6707.6647

c. Plot data and fit a curve

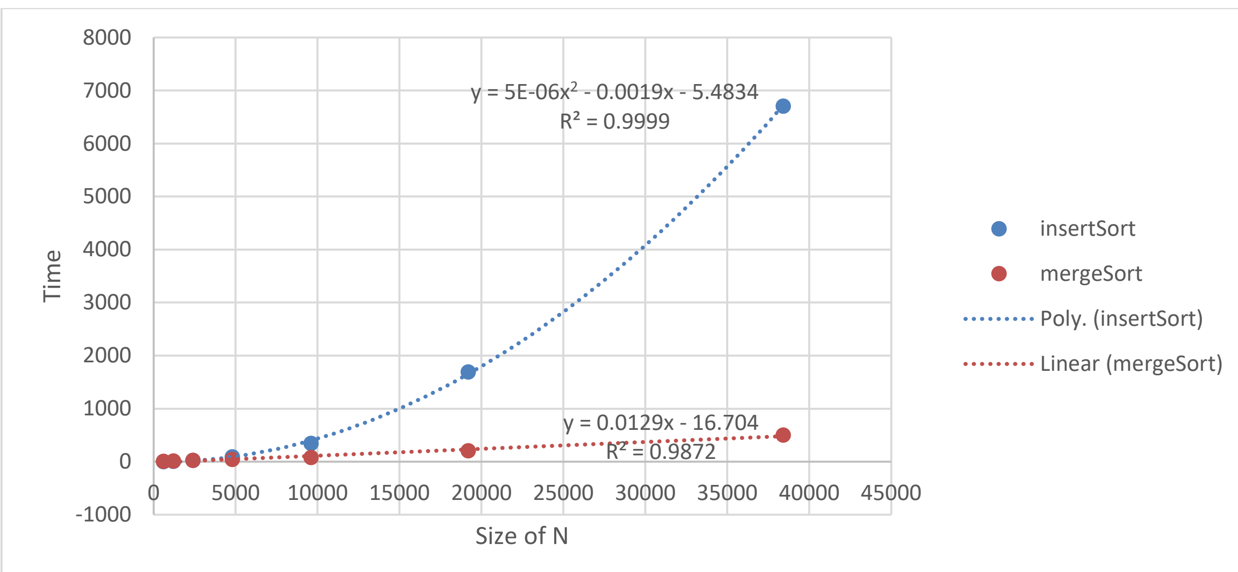
- Merge sort (n*log n) function: It fits well on Linear curve than logarithmic curve when I compared the R-square value. I believe it is because merge part of the merge-sort-algorithm affects this algorithm in term of time complexity when n is getting larger.



- Insert Sort (n^2) : it is fitted with polynomial as I can expected



d. Combined Plot



e. Comparison

Comparing two curves of mergesort algorithm and insertionsort algorithm, I found the mergesort algorithm is much after than insertion sort as we can expected from its f(n). Interest thing is the mergeosrt algorithm is close to linear when n is getting large.