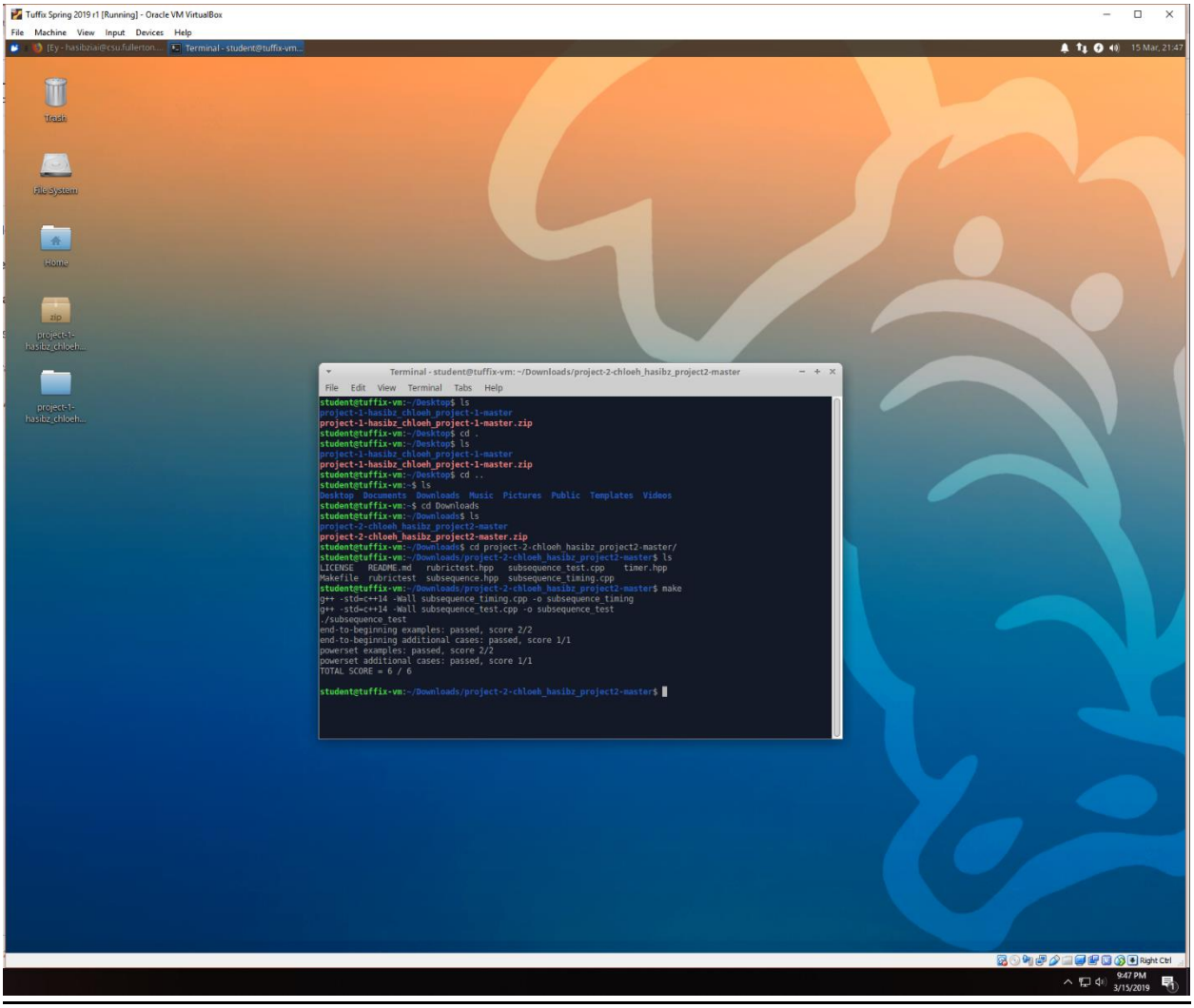
Project 2 Report

Chloe Ho **tho96@csu.fullerton.edu**

Hasib Ziai **hasibziai@csu.fullerton.edu**

Rubric Unit Test:



Pseudocode

```
Longest_decreasing_end_to_beginning(Sequence A passed in){
        Set n equal to length of sequence;
        Create an empty vector that keeps track of sequence lengths;
        for (i = sequence length – 2; as long as I is positive; decrement i each iter.){
                for (j = i + 1; as long as j < sequence length n; increment j each iter.){
                        if(index i of sequence > index j of sequence AND
                            index i of size_vector  <= index j of size vector){
                                    index i of size vector = index j of size vector + 1;
                        }
                }
        }

        Max_length of longest subsequence = max_element_function(from
        size_vector beginning, to size_vector end) + 1;

        Create int_vector of max_length to store longest subsequence;

        Set indices of 'index' = max-1, set j = 0;
        for (i = 0; as long as i < n, increment i){
                if (size_vector[index i] is equal to 'index'{
                        int_vector[index j] equals sequence_vector[index i];
                        decrement 'index';
                        increment j;
                }
        }
        return int_vector with longest subsequence;
}
```

```
longest_decreasing_powerset(sequence A passed in){
        set n = size of sequence A;
        create a sequence 'best' which will store longest subsequence;
        create a size_vector of stacks which are size n+1 and 0;
        create k = 0 to keep track of subsequence sizes;

        infinite while loop until break{
                if (stack of index k < sequence length n){
                        stack of index [k + 1] = stack of index [k] + 1;
                        increment k;
                } else {
                        Increment stack of index [k-1];
                        Decrement k;
                }

                if (size counter k == 0) {
                        break out of while loop;
                }

                create a second sequence 'candidate' to compare to sequence 'best';
                for ( i = 1; as long as i <= k; increment i each loop iteration){
                        add sequence A of index [i] – 1 to back of candidate vector;
                }

                if (check that candidate vector is decreasing AND that candidate
                sequence's size > best sequence's size ){
                        then best sequence = candidate;
                }
                }
        return best subsequence;
        }
```

Mathematical Analysis of Algorithms

```
68  sequence longest_decreasing_end_to_beginning(const sequence& A) {
69
70    const size_t n = A.size();
71
72    // populate the array H with 0 values
73    std::vector<size_t> H(n, 0);
74
75    // calculate the values of array H
76    // note that i has to be declared signed, to avoid an infinite
77    // the loop condition is i >= 0
78    for (signed int i = n-2;  i>= 0; i--) {
79      for (size_t j = i+1; j < n ; j++) {
80        if(A[i] > A[j] && H[i] <= H[j]){
81          H[i] = H[j] + 1;
82        }
83      }
84    }
85
86    // calculate in max the length of the longest subsequence
87      // by adding 1 to the maximum value in H
88    auto max = *std::max_element(H.begin(), H.end()) + 1;
89
90    // allocate space for the subsequence R
91    std::vector<int> R(max);
92
93    // add elements to R by whose H's values are in decreasing order,
94      // starting with max-1
95    // store in index the H values sought
96
97      size_t index = max-1, j = 0;
98      for (size_t i = 0; i < n; ++i) {
99        if (H[i] == index) {
100         R[j] = A[i];
101         --index;
102         ++j;
103       }
104     }
105
106   return sequence(R.begin(), R.begin() + max);
107 }
```

Cost	Time
C_1	1
C_2	1
C_3	$n-2$
C_4	$n-2(n-1) = n^2-3n+2$
C_5	$n^2-3n+2(2)$
C_6	$n^2-3n+2(1)$
C_7	1
C_8	1
C_9	1
C_{10}	n
C_{11}	$n(n) = n^2$
C_{12}	$n^2(1) = n^2$
C_{13}	$n^2(1) = n^2$
C_{14}	$n^2(1) = n^2$

Total Cost: $C_1 + C_2 + C_3(n-2) + C_4(n^2-3n+2) + C_5(2n^2-6n+4) + C_6(n^2-3n+2) + C_7 + C_8 + C_9 + C_{10}(n) + C_{11}(n^2) + C_{12}(n^2) + C_{13}(n^2) + C_{14}(n^2)$

Therefore, The time complexity of the algorithm

$\in \Theta(n^2)$

Mathematical Analysis of Algorithms (continued)

```
109 sequence longest_decreasing_powerset(const sequence& A) {
110   const size_t n = A.size();
111   sequence best;
112   std::vector<size_t> stack(n+1, 0);
113   size_t k = 0;
114   while (true) {
115
116     if (stack[k] < n) {
117       stack[k+1] = stack[k] + 1;
118       ++k;
119     } else {
120       stack[k-1]++;
121       k--;
122     }
123
124     if (k == 0) {
125       break;
126     }
127
128     sequence candidate;
129     for (size_t i = 1; i <= k; ++i) {
130       candidate.push_back(A[stack[i]-1]);
131     }
132
133     if(is_decreasing(candidate) && candidate.size() > best.size()){
134       best = candidate;
135     }
136     }
137
138   return best;
139 }
```

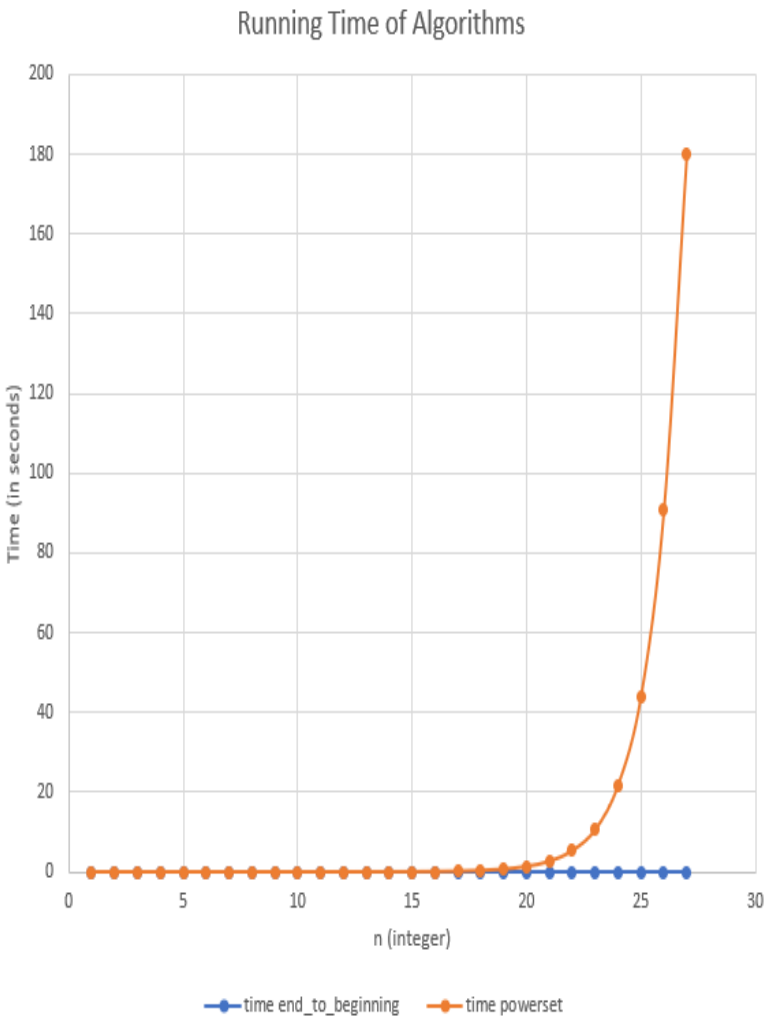
	Cost	Time
109	C_1	1
110	C_2	1
111	C_3	1
112	C_4	1
113	C_5	n
114	C_6	$n(n) = n^2$
116	C_7	$n(1) = n$
117	C_8	$n(1) = n$
118	C_9	$n(1) = n$
119	C_{10}	$n(1) = n$
120	C_{11}	$n(1) = n$
121	C_{12}	$n(1) = n$
122	C_{13}	$n\left(\sum_{k=0}^{n}\binom{n}{k}\right) = n(2^n)$
123	C_{14}	$n(1) = n$
124	C_{15}	$n(n) = n^2$
125	C_{16}	$n(1) = n$

Total Cost: $C_1 + C_2 + C_3 + C_4 + C_5(n) + C_6(n^2) + C_7(n) + C_8(n) + C_9(n) + C_{10}(n) + C_{11}(n) + C_{12}(n) + C_{13}(n2^n) + C_{14}(n) + C_{15}(n^2) + C_{16}(n)$

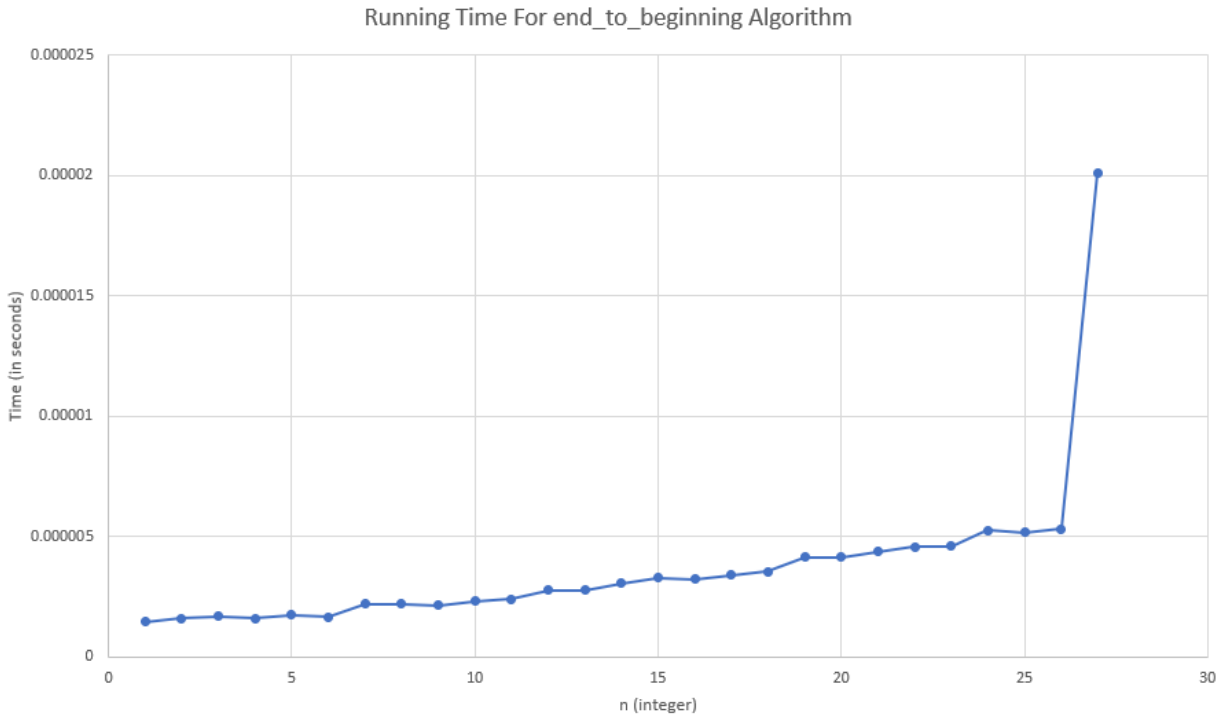
Therefore, the time complexity is $\in \Theta(n2^n)$

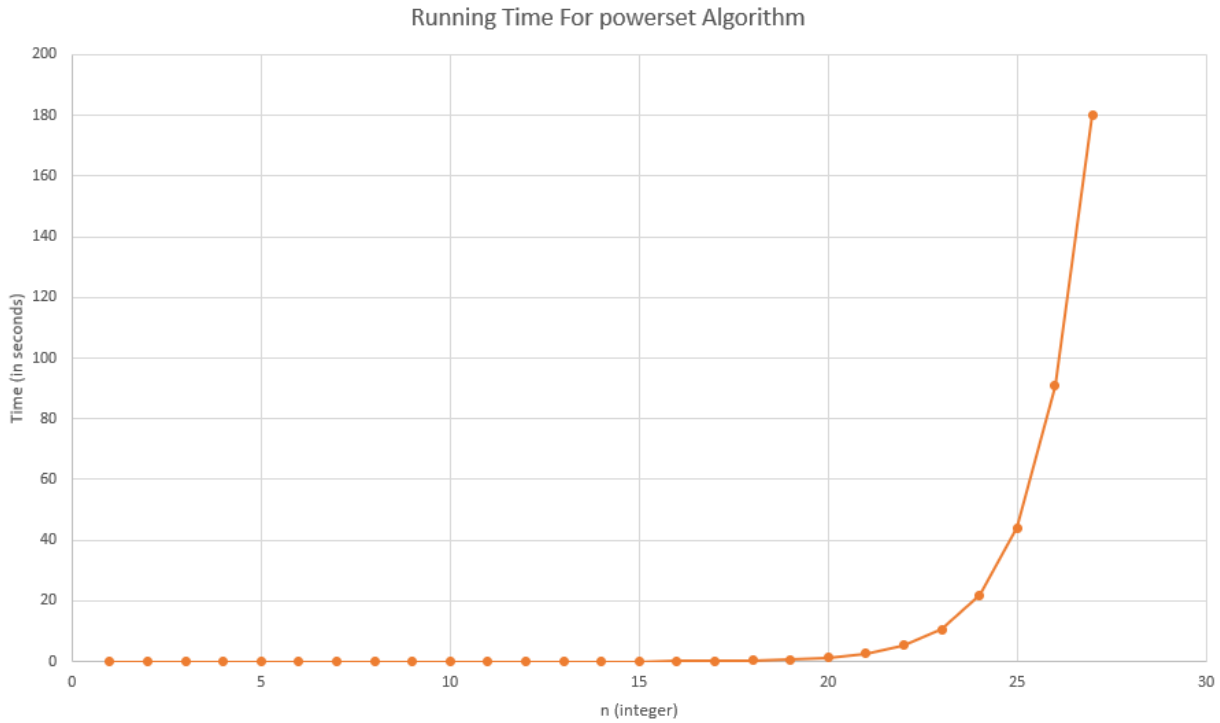
Scatterplots for running time of the two algorithms

n	time end_to_beginning	time powerset
1	0.000001456	0.000001288
2	0.000001598	0.000002318
3	0.000001669	0.000004355
4	0.000001581	0.000008879
5	0.000001726	0.00001937
6	0.000001659	0.000040851
7	0.000002191	0.000141633
8	0.000002195	0.000188723
9	0.000002135	0.000477234
10	0.000002305	0.0010937
11	0.000002395	0.00194063
12	0.000002765	0.00403402
13	0.000002752	0.00893702
14	0.000003054	0.0188038
15	0.000003281	0.0377464
16	0.000003231	0.0760978
17	0.000003381	0.16263
18	0.00000354	0.342611
19	0.000004141	0.66662
20	0.000004136	1.36028
21	0.000004358	2.64218
22	0.000004555	5.34903
23	0.000004599	10.6137
24	0.000005254	21.8315
25	0.000005172	44.1362
26	0.000005309	90.9077
27	0.000020091	180.2

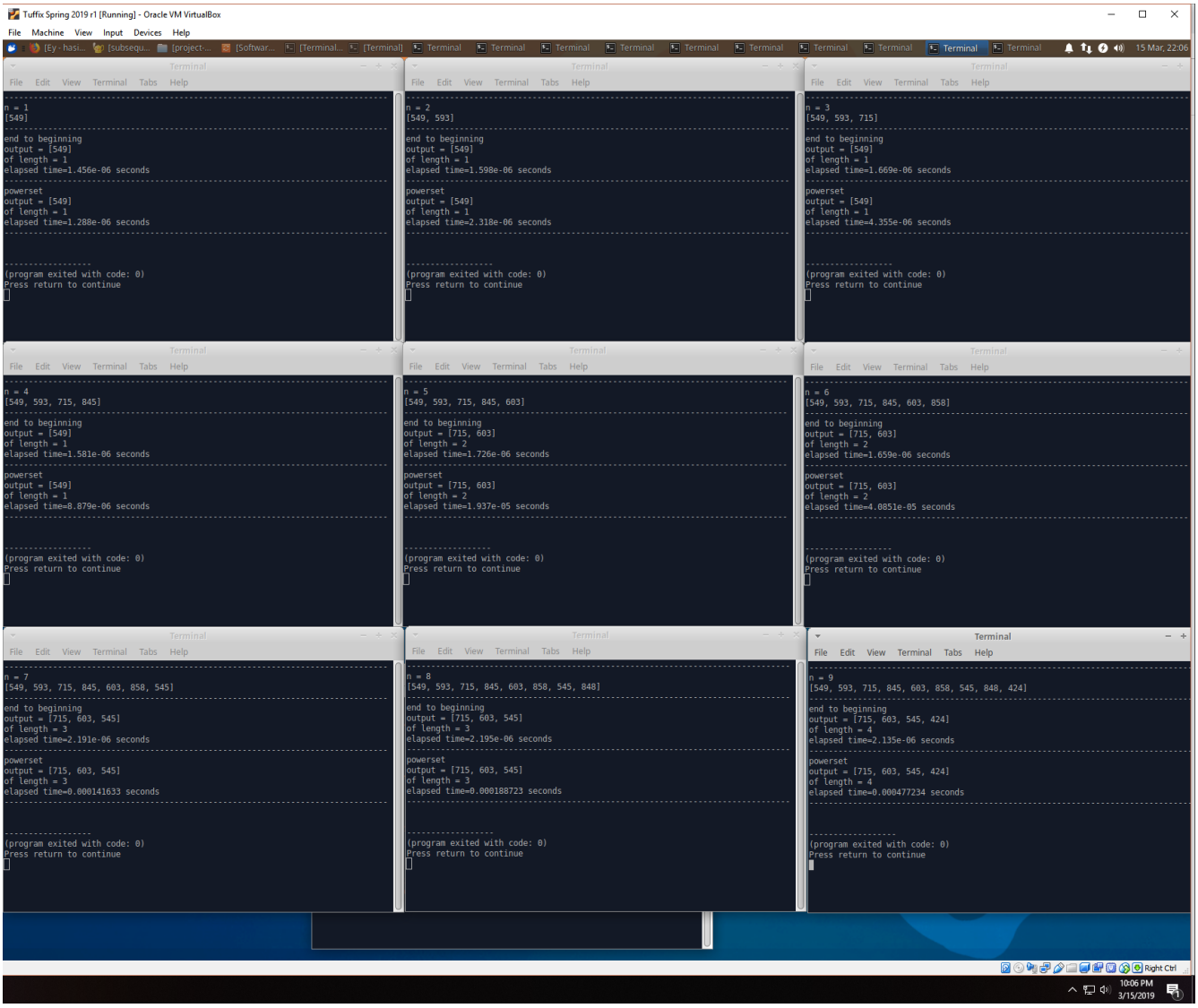


Scatterplots for running time of the two algorithms (continued)

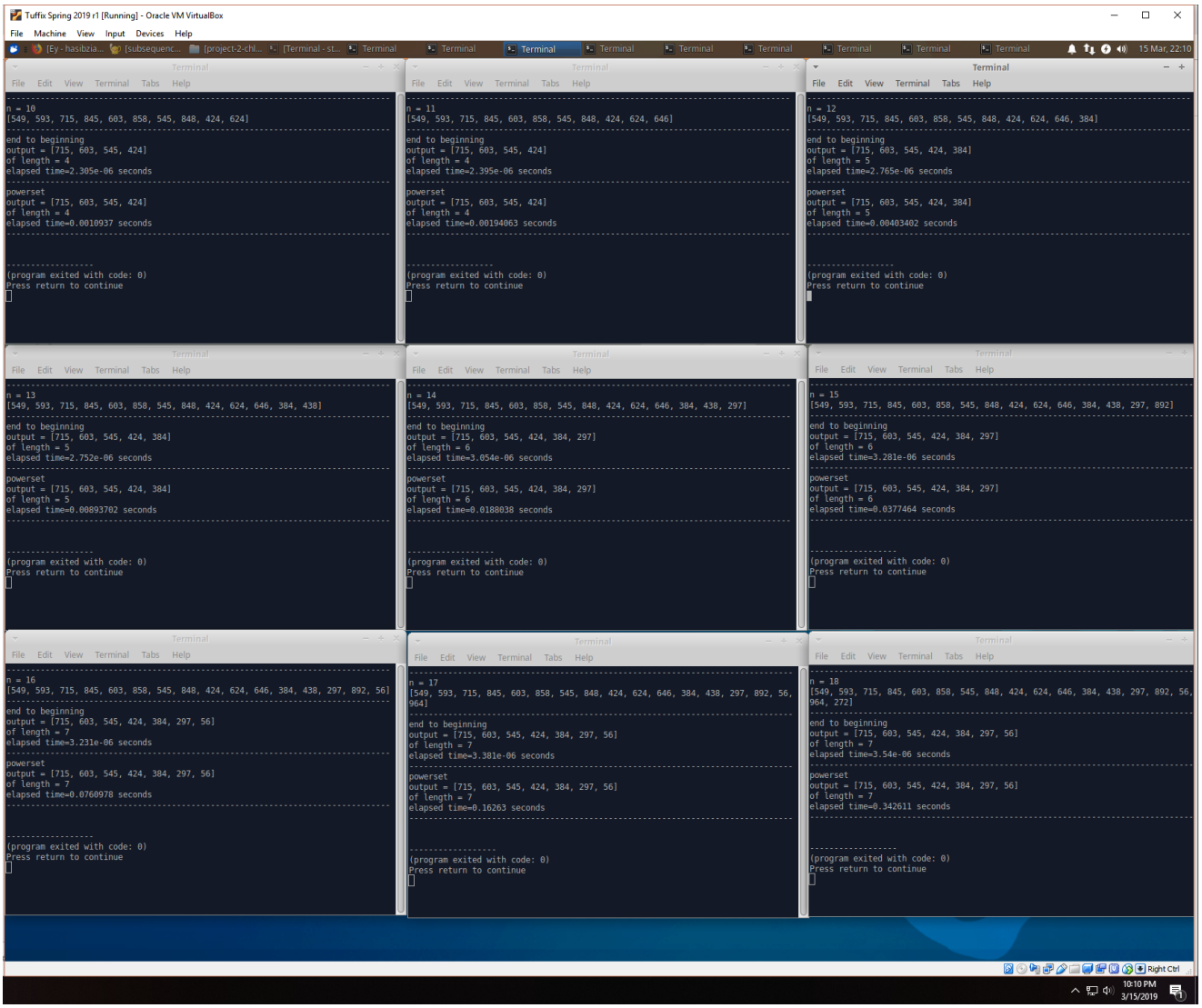




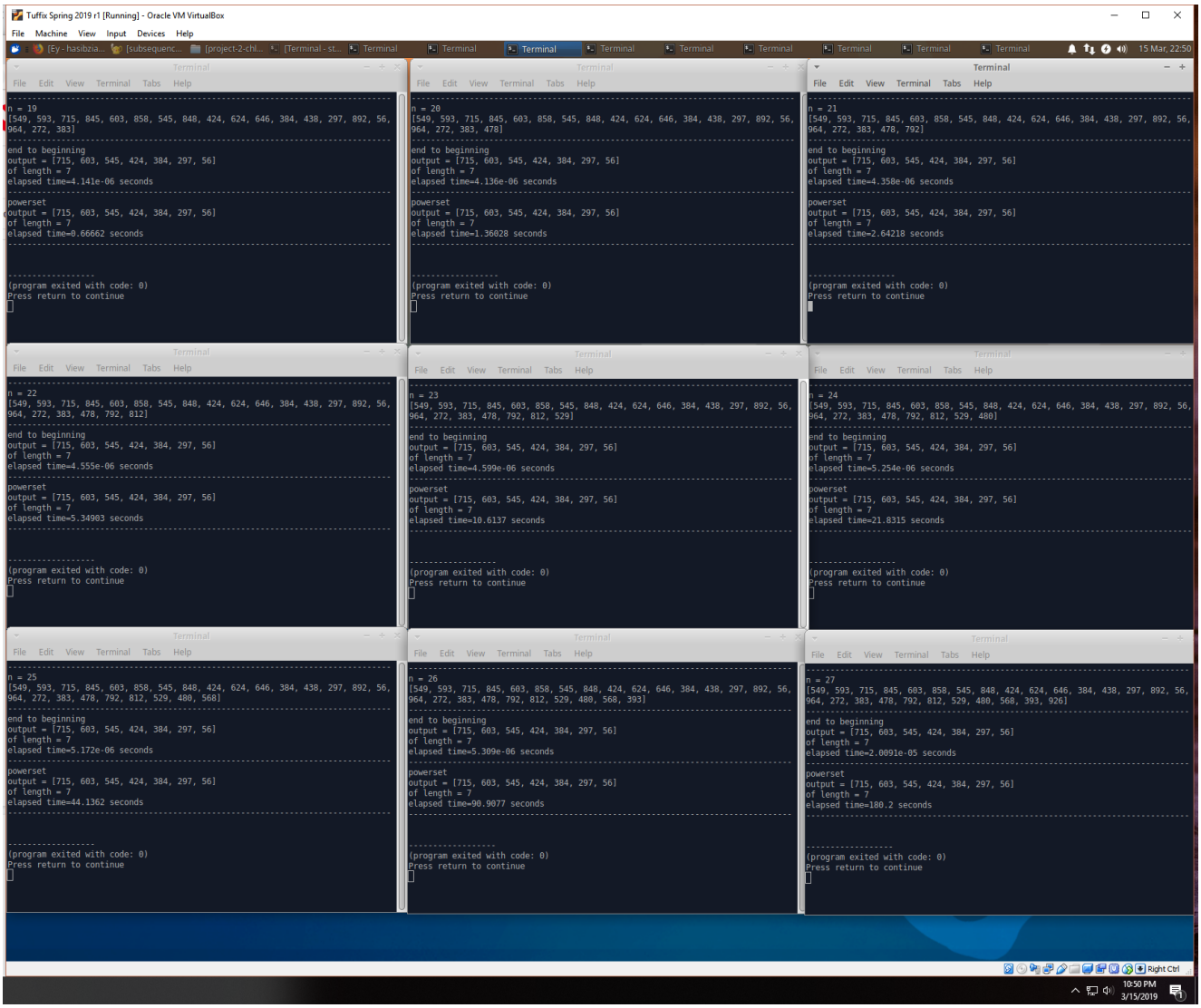
Raw Data for time analysis (Part 1 of 3):



Raw data for time analysis (Part 2 of 3):



Raw data for time analysis (Part 3 of 3):



Conclusion

Therefore, given the data above, it can be concluded that the empirically-observed time efficiency data is consistent with our mathematically-derived big-O efficiency class.

a. Provide pseudocode for your two algorithms.

Pseudocode is provided on pages 2-3.

b. What is the efficiency class of each of your algorithms, according to your own mathematical analysis? (You are not required to include all your math work, just state the classes you derived and proved.)

The efficiency class of the Longest_decreasing_end_to_beginning algorithm is $\theta(n^2)$.
The efficiency class of the Powerset algorithm is $\theta(n2^n)$.

c. Is there a noticeable difference in the running speed of the algorithms? Which is faster, and by how much? Does this surprise you?

There is a noticeable difference in the running speed of the algorithms. The Longest_decreasing_end_to_beginning algorithm is faster by a considerable amount compared to the Powerset algorithm. It can be seen from the empirical data that the amount of time the Powerset algorithm takes quickly diverges in comparison to the end_to_beginning algorithm, as early as n=3, and becomes more and more obvious as n>20. This does surprise me, as from what I read, n^2 should be larger than 2^n until around n>1000. This just shows that the variable at the front, $n*2^n$, makes all the difference.

d. Are the fit lines on your scatter plots consistent with these efficiency classes? Justify your answer.

Yes, the fit lines are consistent with the efficiency classes. This can be seen in the second part of the scatterplots (page 7). After n>20, the graph of the end_to_beginning algorithm begins to grow faster, much like how the graph of n^2 should look. The graph of the powerset algorithm is consistent with the graph of $n2^n$, and begins to grow quickly in comparison to n^2. After n>20, the graph of $n2^n$ begins to grow very quickly and closely resembles the graph of 2^n.

e. Is this evidence consistent or inconsistent with the hypothesis stated on the first page? Justify your answer.

The evidence shown is consistent with the hypothesis stated on the first page. Exhaustive search algorithms, such as the powerset function, are clearly feasible to implement and produce the correct outputs, given that n is small. As n grows larger, algorithms with exponential or factorial running times become impractical as the running time is much larger compared to an algorithm, with say, polynomial time. Our empirical data shows this clear difference in running time, and it can be concluded that the data is consistent with the hypothesis.