

# Project 2 Report

Chloe Ho

tho96@csu.fullerton.edu

Hasib Ziai

hasibziai@csu.fullerton.edu

## Rubric Unit Test:

```
student@tuffix-vm:~/Downloads/project-2-chloeh_hasibz_project2-master$ ls
project-1-hasibz_chloeh_project1-master
project-1-hasibz_chloeh_project1-master.zip
student@tuffix-vm:~/Downloads/project-2-chloeh_hasibz_project2-master$ cd ..
student@tuffix-vm:~/Downloads/project-2-chloeh_hasibz_project2-master$ ls
project-1-hasibz_chloeh_project1-master
project-1-hasibz_chloeh_project1-master.zip
student@tuffix-vm:~/Downloads/project-2-chloeh_hasibz_project2-master$ make
g++ -std=c++14 -Wall -subsequence.timing.cpp -o subsequence.timing
./subsequence.timing
TOTAL SCORE = 6 / 6
```

## 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) {	<u>Cost</u>	<u>Time</u>
69		$C_1$	1
70	const size_t n = A.size();		
71		$C_2$	1
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	$C_3$	$n-2$
78	for (signed int i = n-2; i >= 0; i--) {	$C_4$	$n-2(n-1) = n^2-3n+2$
79	for (size_t j = i+1; j < n; j++) {	$C_5$	$n^2-3n+2(2)$
80	if (A[i] > A[j] && H[i] <= H[j]){	$C_6$	$n^2-3n+2(1)$
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	$C_7$	1
88	auto max = *std::max_element(H.begin(), H.end()) + 1;		
89			
90	// allocate space for the subsequence R	$C_8$	1
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	$C_9$	1
96			$n$
97	size_t index = max-1, j = 0;	$C_{10}$	
98	for (size_t i = 0; i < n; ++i) {	$C_{11}$	$n(n) = n^2$
99	if (H[i] == index) {	$C_{12}$	$n^2(1) = n^2$
100	R[j] = A[i];	$C_{13}$	$n^2(1) = n^2$
101	--index;	$C_{14}$	$n^2(1) = n^2$
102	++j;		
103	}		
104	}		
105			
106	return sequence(R.begin(), R.begin() + max);		
107	}		

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

$$E O(n^2)$$

## Mathematical Analysis of Algorithms (continued)

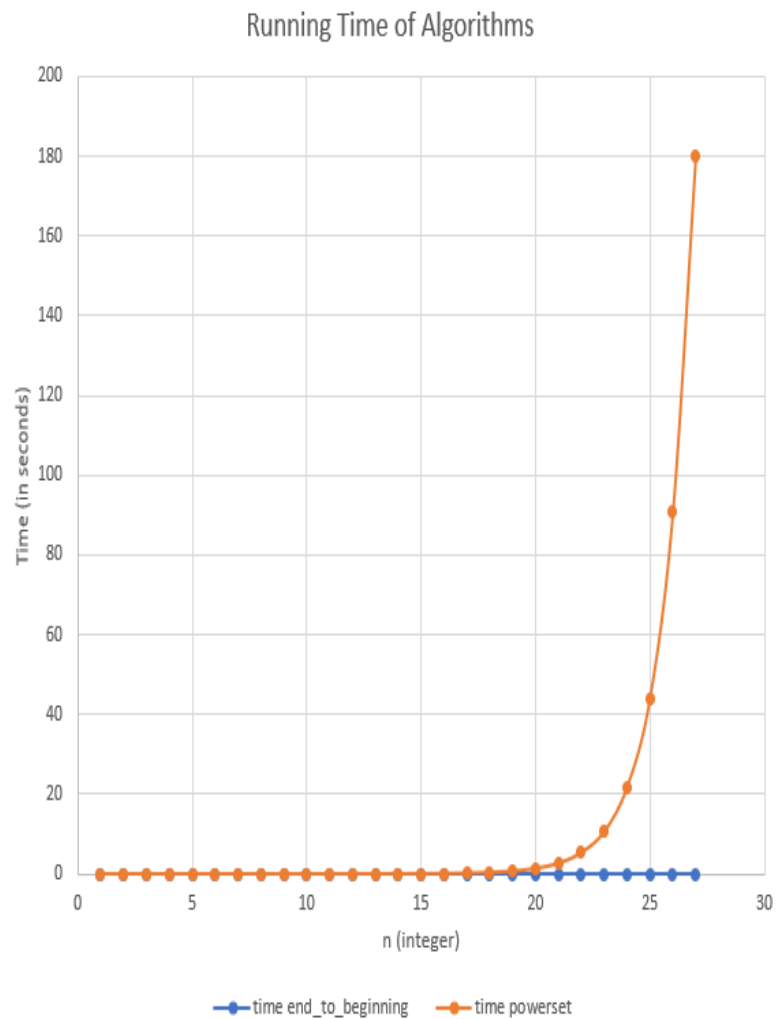
	<u>Cost</u>	<u>Time</u>
109 sequence longest_decreasing_powerset(const sequence& A) {	$C_1$	1
110 const size_t n = A.size();	$C_2$	1
111 sequence best;	$C_3$	1
112 std::vector<size_t> stack(n+1, 0);	$C_4$	1
113 size_t k = 0;	$C_5$	$n$
114 while (true) {		
115		
116 if (stack[k] < n) {	$C_6$	$n(n) = n^2$
117 stack[k+1] = stack[k] + 1;	$C_7$	$n(1) = n$
118 ++k;	$C_8$	$n(1) = n$
119 } else {		
120 stack[k-1]++;	$C_9$	$n(1) = n$
121 k--;	$C_{10}$	$n(1) = n$
122 }		
123		
124 if (k == 0) {	$C_{11}$	$n(1) = n$
125 break;		
126 }		
127	$C_{12}$	$n(1) = n$
128 sequence candidate;	$C_{13}$	$n(\sum_{k=0}^n \binom{n}{k}) = n(2^n)$
129 for (size_t i = 1; i <= k; ++i) {	$C_{14}$	$n(1) = n$
130 candidate.push_back(A[stack[i]-1]);		
131 }		
132		
133 if(is_decreasing(candidate) && candidate.size() > best.size()){	$C_{15}$	$n(n) = n^2$
134 best = candidate;	$C_{16}$	$n(1) = n$
135 }		
136 }		
137		
138 return best;		
139 }		

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

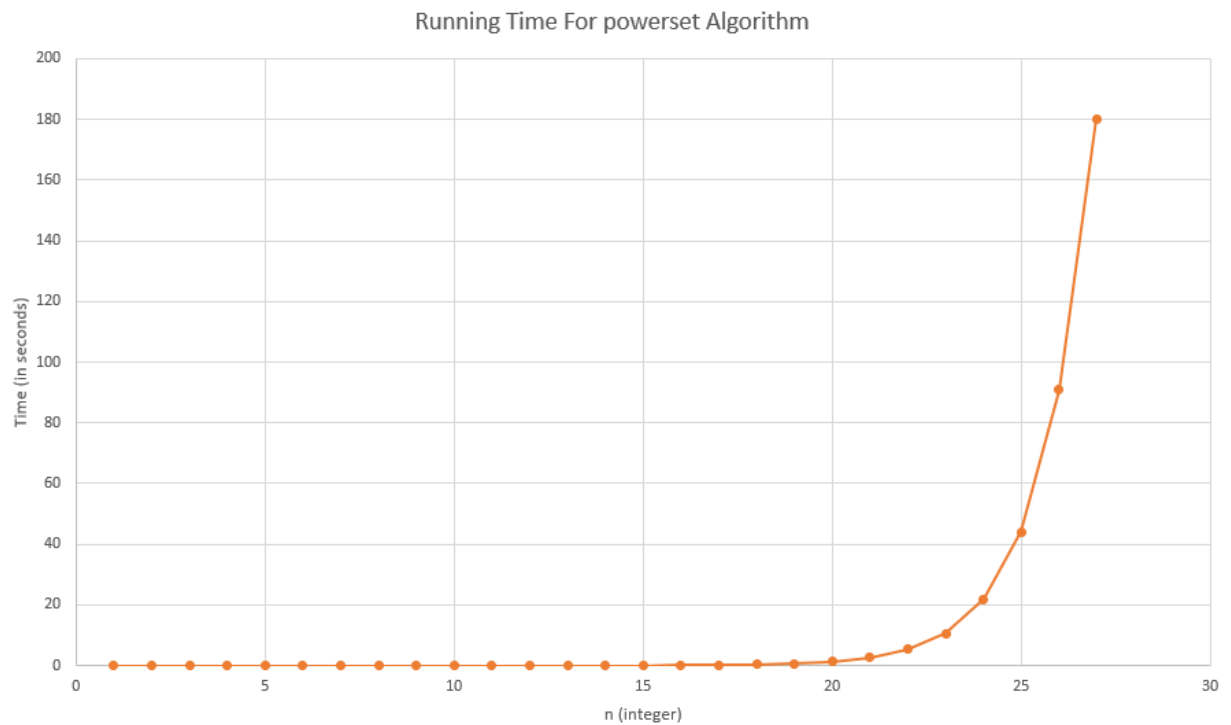
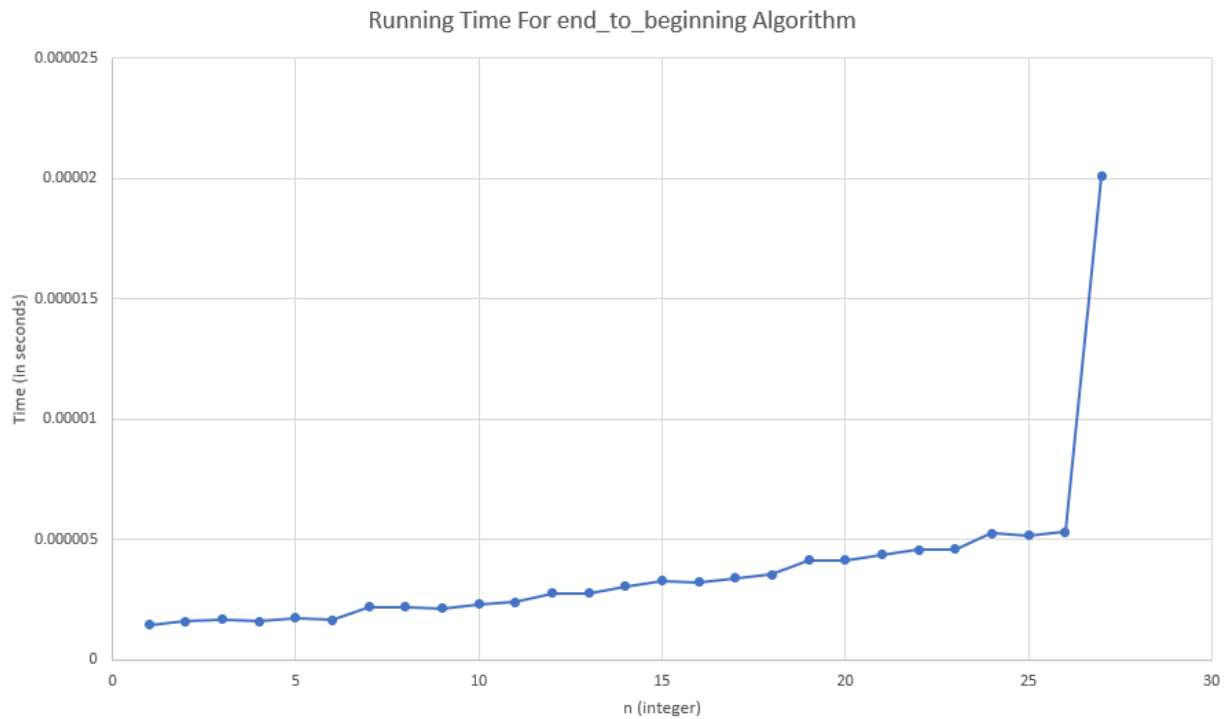
Therefore, the time complexity is  
 $\in \Theta(n2^n)$

## 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):

The screenshot displays a virtual machine interface with nine terminal windows arranged in a 3x3 grid. Each terminal window shows the output of a program for a specific value of  $n$  (from 1 to 9). The output includes the input array, the time taken to generate the array, the time taken to process the array, and the time taken to output the result. The data is as follows:

$n$	Input Array	end to beginning (seconds)	power set (seconds)	output (seconds)
1	[549]	1.456e-06	1.289e-06	-
2	[549, 593]	1.598e-06	2.318e-06	-
3	[549, 593, 715]	1.669e-06	4.355e-06	-
4	[549, 593, 715, 845]	1.581e-06	8.879e-06	-
5	[549, 593, 715, 845, 603]	1.726e-06	1.937e-05	-
6	[549, 593, 715, 845, 603, 858]	1.659e-06	4.0851e-05	-
7	[549, 593, 715, 845, 603, 858, 545]	2.191e-06	0.000141633	-
8	[549, 593, 715, 845, 603, 858, 545, 848]	2.195e-06	0.000188723	-
9	[549, 593, 715, 845, 603, 858, 545, 848, 424]	2.135e-06	0.000477234	-



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

The screenshot shows a VirtualBox VM titled "Tuffin Spring 2019 v1 (Running) - Oracle VM VirtualBox". The VM is running a program that outputs performance data for various values of  $n$  (from 10 to 18). Each terminal window displays the following information:

- $n$  value
- Array of numbers: [549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646]
- end to beginning
- output = [715, 603, 545, 424]
- of length = 4
- elapsed time = 2.305e-06 seconds
- power set
- output = [715, 603, 545, 424]
- of length = 4
- elapsed time = 0.0010937 seconds
- (program exited with code: 0)
- Press return to continue

The data for  $n=10$  to  $n=18$  is as follows:

n	Array	end to beginning	output	of length	elapsed time (s)	power set	output	of length	elapsed time (s)
10	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646]	end to beginning	[715, 603, 545, 424]	4	2.305e-06	power set	[715, 603, 545, 424]	4	0.0010937
11	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646]	end to beginning	[715, 603, 545, 424]	4	2.395e-06	power set	[715, 603, 545, 424]	4	0.00194083
12	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384]	end to beginning	[715, 603, 545, 424, 384]	5	2.765e-06	power set	[715, 603, 545, 424, 384]	5	0.00403402
13	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438]	end to beginning	[715, 603, 545, 424, 384]	5	2.752e-06	power set	[715, 603, 545, 424, 384]	5	0.00893702
14	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297]	end to beginning	[715, 603, 545, 424, 384, 297]	6	3.854e-06	power set	[715, 603, 545, 424, 384, 297]	6	0.0188038
15	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892]	end to beginning	[715, 603, 545, 424, 384, 297]	6	3.281e-06	power set	[715, 603, 545, 424, 384, 297]	6	0.0377464
16	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56]	end to beginning	[715, 603, 545, 424, 384, 297, 56]	7	3.231e-06	power set	[715, 603, 545, 424, 384, 297, 56]	7	0.0760978
17	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964]	end to beginning	[715, 603, 545, 424, 384, 297, 56]	7	3.381e-06	power set	[715, 603, 545, 424, 384, 297, 56]	7	0.16263
18	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272]	end to beginning	[715, 603, 545, 424, 384, 297, 56]	7	3.54e-06	power set	[715, 603, 545, 424, 384, 297, 56]	7	0.342611

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

The screenshot displays a virtual machine interface titled "Tuffin Spring 2019 r1 [Running] - Oracle VM VirtualBox". It contains nine terminal windows arranged in a 3x3 grid, each showing the output of a program for different values of  $n$  (19 to 27). The output for each terminal includes a list of numbers, timing information, and a confirmation message.

$n$	Output List	elapsed time
19	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383]	4.141e-06 seconds
20	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478]	4.136e-06 seconds
21	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478, 792]	4.358e-06 seconds
22	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478, 792, 812]	4.555e-06 seconds
23	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478, 792, 812, 529]	4.599e-06 seconds
24	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478, 792, 812, 529, 480]	5.254e-06 seconds
25	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478, 792, 812, 529, 480, 568]	5.172e-06 seconds
26	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478, 792, 812, 529, 480, 568, 393]	5.309e-06 seconds
27	[549, 593, 715, 845, 603, 858, 545, 848, 424, 624, 646, 384, 438, 297, 892, 56, 964, 272, 383, 478, 792, 812, 529, 480, 568, 393, 926]	9.0977 seconds

Each terminal window also displays the following text:

```
end to beginning
output = [715, 603, 545, 424, 384, 297, 56]
of length = 7
elapsed time=... seconds

powerset
output = [715, 603, 545, 424, 384, 297, 56]
of length = 7
elapsed time=... seconds

(program exited with code: 0)
Press return to continue
```

## 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.**