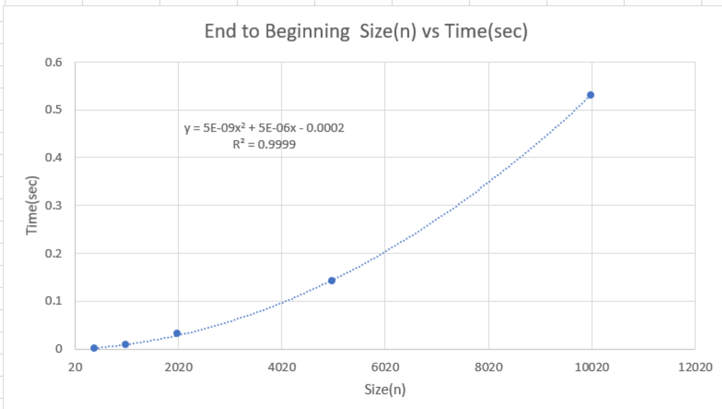
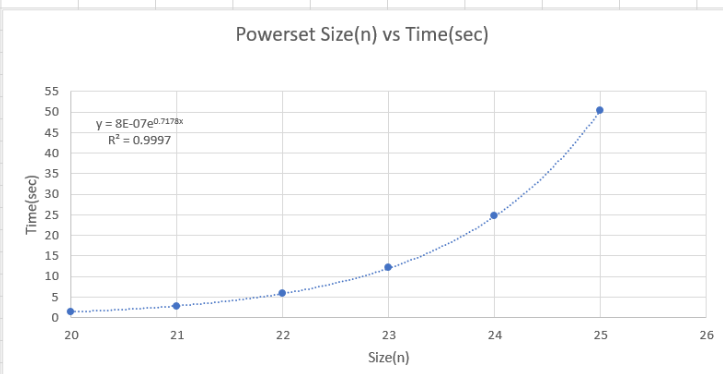
CPSC 335 Project 2 Submission
Shivam Shah shivshah98@csu.fullerton.edu
Andrew Nguyen andrewngn13@csu.fullerton.edu

End to Beginning	Size	Time	Powerset	Size	Time
	400	0.00135		20	1.43
	1000	0.00834		21	2.75
	2000	0.032368		22	5.91
	5000	0.14249		23	12.16
	10000	0.53059		24	24.78
				25	50.34





#PSEUDOCODE#

sequence longest_increasing_end_to_beginning(const sequence& A) {
 const size_t n = A.size(); // 1 TS

 // populate the array H with 0 values
 std::vector<size_t> H(n, 0); // 1 TS

```
 // calculate the values of array H
 // note that i has to be declared signed, to avoid an infinite loop, since
 // the loop condition is i >= 0
 for ( i = n-2;  to 0)  { // n-2+1 times
   for ( j = i+1;to n) {  // n times
     if(A[j] > A[i] && H[j] >= H[i])  // 3 +max (2,0)TS
     {
      Add 1 to the element// 2 TS
     }
   }
 }
// calculate in max the length of the longest subsequence
 // by auto max = *std::max_element(H.begin(), H.end()) + 1;

 // allocate space for the subsequence R
 std::vector<int> R(max);

 // add elements to R by whose H's values are in decreasing order,
   // starting with max-1
 // store in index the H values sought

   size_t index = max-1, j = 0; //2 TS
   for (i = 0; to n) { //n Times
     if (H[i] is index) { //1 + max(3,0)
         Set R[j] to be A[i] // 1 TS
        Subtract index;       // 1 TS
          Add j;         // 1 TS
= 2+(n-1)(n)(5)+2+n(4 + 1+ 1+ 1)
= 5n^2 - 5n + 4 + 7n
= 5n^2 + 3n + 4
  // write the statements to add A[i] to the sequence R by
         // storing it into R[j], decrement index and increment j
     }
   }

const sequence A                    //parameter provided
int k=0                      // 1 TS
for all sets X up to size n               // 2^n times
  if X[k] < n                     // 1 + max(3,2)
    X[k+1] = X[k]+1               // 3 TS
```

```
  else
    stack[k-1]++              // 1 TS
    k--                 // 1 TS
if k == 0                    // 1 TS + max(1,0)
  break;

sequence candidate
  for 1 to <= k                    // k times
   candidate.push_back(A[stack[i]-1]);       // 2 TS
  end

if is_increasing(candidate) && candidate.size() > best.size     //2+ max(1,0)
  best = candidate                         // 1
end
```

1 + (2^n)*((1+max(3,2) + (1 + max(1,0)) + (k *2) + (2 + max(1,0))
=2^n(4+2+2k+3)
= 2^n(2k+10)

B. The efficiency of longest increasing end to beginning algorithm is O(n^2) while the efficiency of the powerset algorithm is O(2^n). These efficiency classes are derived from the timestep solutions such that 5n^2 + 3n + 4 results in O(n^2) and 2^n(2k+10) results in O(2^n).

C. There is a noticeable difference in the running speed of the algorithms. The O(n^2) algorithm is significantly faster than O(2^n) by a large margin. We were not surprised by the large time difference between the two algorithms.

D. Yes. They are consistent, however for the first algorithm's set of N data, the range had to be increased to see a major change in time computation. For the second algorithm, the time difference was immediately noticeable even with small increments of size N.

E. Yes. The evidence is consistent from the hypothesis given to us on the first page. We can see through the best fit line and the equations that this is the case. The data and results produced indicate that algorithms with exponential or factorial running times are extremely slow, and in most cases, too impractical for common use.