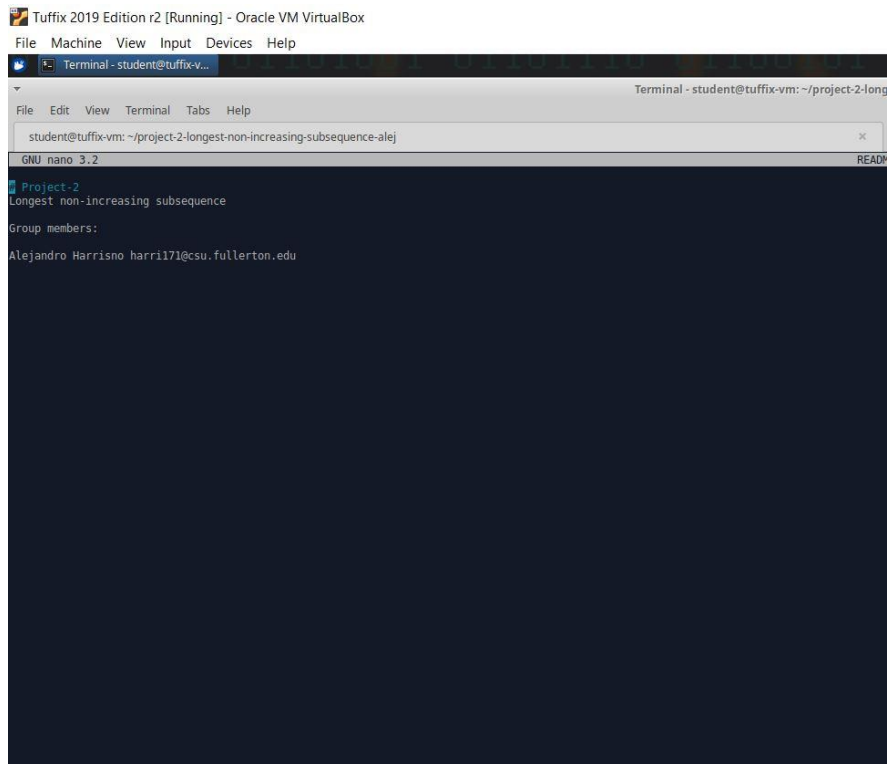


## PROJECT 2

Alejandro Harrison harri171@csu.fullerton.edu



```
Tuffix 2019 Edition r2 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal - student@tuffix-v-...
Terminal - student@tuffix-vm: ~/project-2-long
student@tuffix-vm: ~/project-2-longest-non-increasing-subsequence-alej
GNU nano 3.2
Project-2
Longest non-increasing subsequence
Group members:
Alejandro Harrison harri171@csu.fullerton.edu
```

*longest sorted (non-increasing) subsequence*

**input:** a vector  $V$  of  $n$  comparable elements

**output:** a vector  $R$  containing the longest reversely sorted subsequence of  $V$

### End To Beginning:

end\_to\_beginning(vector A):

vector R;

for i in range(n-2...n-1....0):

for j in range (i+1.....n):

if(A[j] <= A[i] && H[j] >= H[i])

H[i] = H[j]+1;

endif

endfor

endfor

```

int max = max_element(H.begin(), H.end())+1;
vector R(max);
int index = max-1;
int j = 0;
for i in range (0.....n):
    if(H[i] == index):
        R.insert(R.begin()+j, A[i];
        index = index +1;
        j = j+1;
    endif
endfor
return sequence(R.begin(), R.begin()+max);

```

### Longest Nonincreasing:

```

longest_nonIncreasing(vector A):
    vector best;
    int k = 0;
    while(true):
        if(stack[k] < n):
            stack[k+1] = stack[k] +1;
            k = k+1;
        else:
            stack[k-1] = stack[k-1]+1;
            k = k-1;
        endelse
    endif
    if (k == 0)
        break;

```

```

endif
vector candidate;
for i in range (1.....k):
    candidate.push_back(A[stack[i]-1]);
endfor
//call bool is_nonincreasing(candidate):
for i in range (0.....candidate.size()):
    if(candidate[i] < candidate[i+1]):
        return false;
    endif
endfor
return true
if( is_nonincreasing(candidate) == true && candidate.size() > best.size()):
    best = candidate;
endif

return best;

```

## Time Complexity:

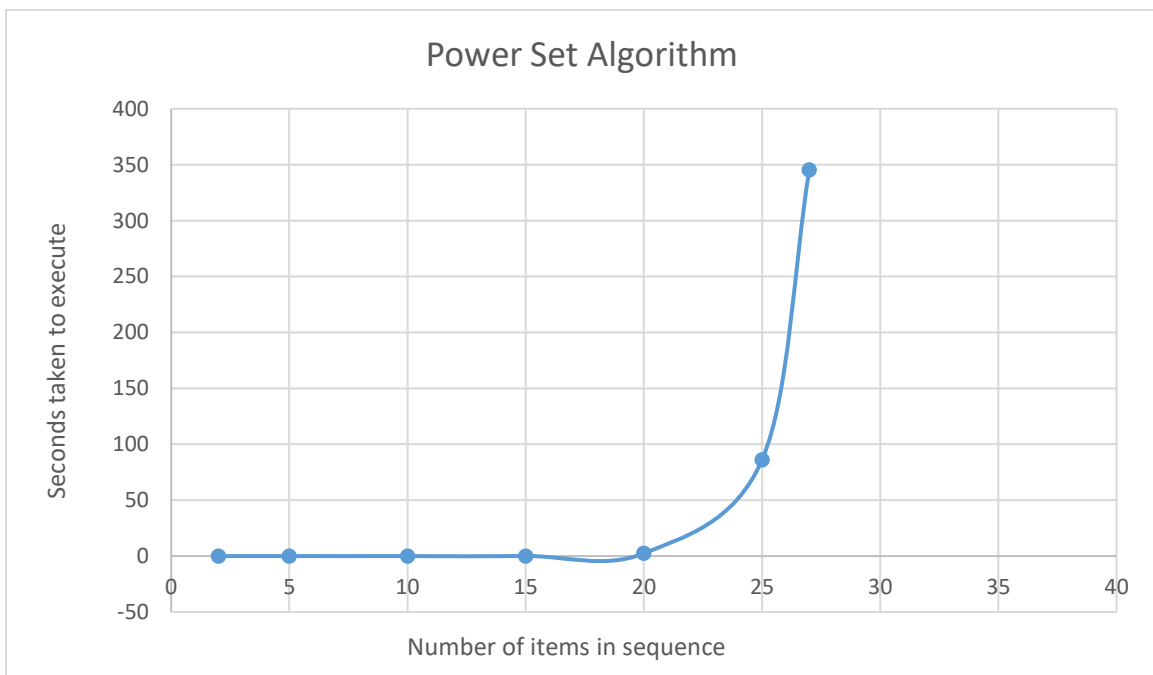
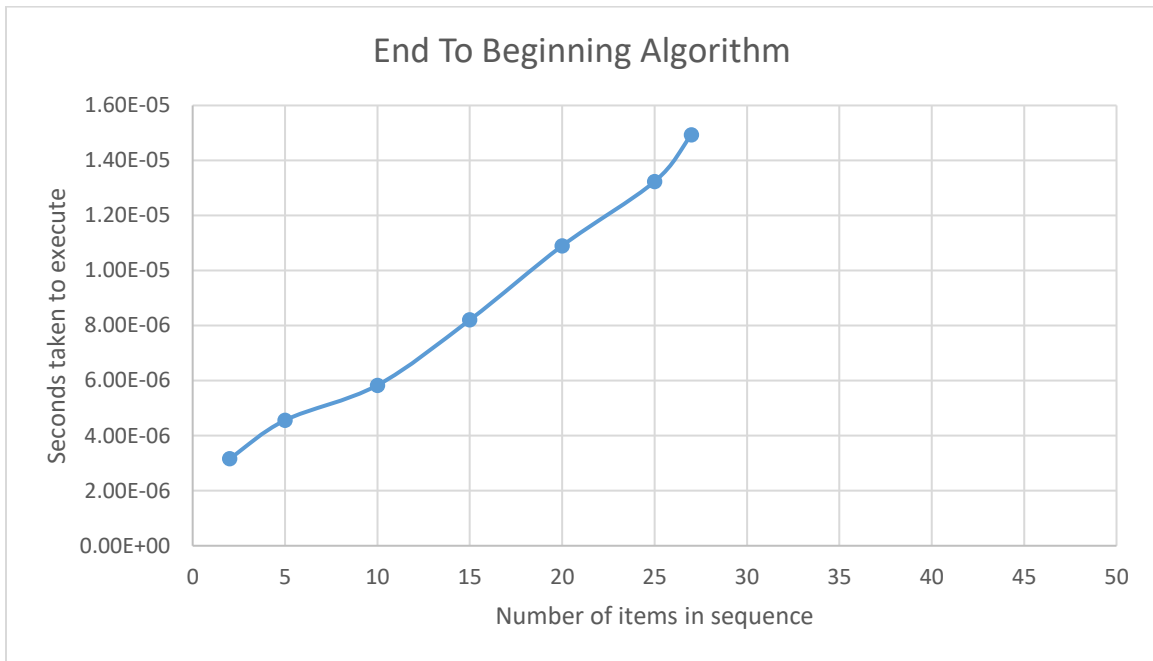
For the end to beginning algorithm, the efficiency class will be  $O(n^2)$ . The steps are demonstrated below.

- We start off with a nested for loop. The nested for loop has an innermost if statement as well.
- In the nested for loop, the inner for loop starts at  $j = i+1$  and goes to  $n$ . This means that the  $j$  innermost for loop depends on the outer for loop to determine how many times to loop, which will differ each time. Therefore we're only concerned with the inner loop.
- Using what we learned in class, we can set up the for loop using the summation  $\sum_{j=i}^n 1$ .
- This can be broken up into  $\sum_1^n 1 - \sum_1^i 1$
- $\sum_1^n 1$  can be simplified into  $n$ .
- $\sum_1^i 1$  can be simplified into  $\frac{n(n+1)}{2}$
- Thus you have  $n - \frac{n(n+1)}{2}$

- Reducing you will get  $\frac{2n}{2} - \frac{n^2+n}{2}$  or  $\frac{-n^2-n}{2}$
- Pulling out a  $-\frac{1}{2}$  gives you  $-\frac{1}{2}(n^2+n)$  t.u.
- We then look at the if statement.
- Calculating the t.u. of an if statement uses the formula: (if-condition t.u.) + Max(statement\_block<sub>1</sub> t.u., statement\_block<sub>2</sub> t.u. ...)
- Using the formula, the first condition ( if((A[j] <= A[i]) && (H[j] >= H[i])) ) has 3 t.u, while the inner statement has 2 t.u. Therefore the t.u of the entire if statement would be 3 + max(2,0) which is 5.
- The first block of code (the nested for loop with the if statement) is now equivalent to the inner for loop statement t.u. multiplied by the if statement t.u. :  $-\frac{1}{2}(n^2+n) * 5$ .
- We move onto the last for loop in this function which also contains an if statement.
- The for loop which starts at i = 0 and ends at n, loops  $\frac{\text{Final value} - \text{initial value}}{\text{Number of steps incremented}} + 1$  times or in this case, n+1 times.
- The if statement then has 1 + max(3,0) or 4 t.u.
- Multiplying this by the for loop, you get 4 \* (n+1).
- Putting this all together we have :  $-\frac{5}{2}(n^2+n) + 4(n+1)$  or :  $-\frac{5}{2}(n^2+n) + 4n+4$ .
- By definition, we can eliminate trailing factors and so then have :  $-\frac{5}{2}(n^2+n)$ .
- Again by definition we can eliminate any constants, so we are left with  $n^2+n$ , which simplifies by definition to  $n^2$ .
- Therefore the efficiency class of this algorithm is  $O(n^2)$ .  
For the longest nonincreasing algorithm, the efficiency class will be  $O(n * 2^n)$ . The steps are demonstrated below.

- We start off with a while loop, which we were told executes  $2^n$  times.
- Then we look inside the while loop, where we have a few if statements, and a for loop.
- The first if statement ( if (stack[k] < n) ) t.u. using the formula mentioned before is 1+max(2,2) or 3 t.u.
- The next if statement ( if (k == 0) ) is 1 + max(1,0) = 1 t.u.
- Then we come to the for loop and following the formula, we get:  $\frac{n-1}{1}+1$  or n t.u.
- The last if statement ( if(is\_nonincreasing(candidate) && candidate.size() > best.size())) has a t.u. of 2 + max (1,0) or 3.
- However it also contains the function is\_nonincreasing(), which taking into account it's for loop and if loop has a t.u of 2n.
- Adding the max to the last if statement gives a t.u. of 2n+2
- Putting these all together including the while loop we have:  $2^n (3+1+n+2n+2)$
- Simplifying we get  $2^n(3n+6)$ .
- However by definition of O we can drop the constants and lesser terms and we get  $2^n(n)$ .
- Therefore the efficiency class of this algorithm is  $O(2^n*n)$ .

## Scatterplots



## Conclusion

Both of these scatterplot patterns are consistent with the pattern expected from such efficiency classes. Because the powerset algorithm was efficiency class  $O(n * 2^n)$  it showed exponential growth and therefore the more elements added, the longer it will take to finish. It did indeed take longer to finish the more elements were added, especially when compared to the end to beginning algorithm. This makes sense, because  $O(c^n)$  is the second worst efficiency class you can (right below  $O(n!)$ ), and therefore will take significantly longer with the slightest of changes. When I tried to change from 27 elements to 30, the time difference was significantly longer, to the point where I walked away from my computer and came back later, and it still hadn't finished. I was surprised by how much longer it took for the powerset algorithm as compared to the end to beginning algorithm, since  $O(n^c)$  is right below  $O(c^n)$  when it comes to worst cases. All in all, the evidence does support both hypotheses that exhaustive algorithms are feasible and produce correct outputs, but that if one was to use an exponential growth exhaustive algorithm, it would be too slow and of little practical use.