CPSC 335 Fall 2019 MW 2:30 - 3:45 pm Project 2 Report

Carla Jacobsen cjacobsen2016@csu.fullerton.edu

End-to-Beginning Algorithm:

```
def       longest_nondecreasing_end_to_beginning(A)
{//function

        n = A.size()

        create a vector H filled with the value 0

        //calculate the values for H
        for i = n – 2 to 0 do    //block A
                {//outer

                for j = i + 1 to n do     //block B
                        {//inner

                        //criteria for H[i] to be updated
                        if (A[i] <= A[j]) && (H[i] <= H[j]) do          //block C
                                {//if
                                //update H[i]
                                H[i] = 1 + H[j]

                                }//if     //block C
                        }//inner          //block B
                }//outer          //block A


        //get the length of the longest subsequence
        max = <maximum value in H> + 1

        //R is used for a subsequence
        create a vector R

        index = max – 1

        j = 0
```

```
        //find the longest subsequence
        for i = 0 to n do          //block D
                {//for

                if H[i] == index do     //block E
                        {//if

                        R[j] = A[i]

                        --index

                        ++j

                        }//if     //block E
                }//for   //block D

        return R

}//function
```

Proof for End-to-Beginning Algorithm:

sc before block A:

<sc before block A> =
<n = A.size()> + <vector H> = 2 + 1 =
3

sc after block A and before block D:

<sc after block A and before block D> =
<max = maximum value in H + 1> + <vector R> + <index = max – 1> + <j = 0> =
2 + 1 + 2 + 1 = 4 + 2 =
6

sc after block D:

<sc after block D> =
<return R> =
1

sc outside of any block:

<sc outside of any block> =
<before A sc> + <after A before D sc> + <after D sc> =
3 + 6 + 1 =
10

sc for block A:

<A sc> = <duration A> * <inside A>

<duration A> = n – 2 - 0 + 1 = n – 1

<inside A> = <B sc>

<B sc> = <duration B> * <inside B>

<duration B> = i + 1 – n + 1 = i – n + 2

<inside B> = <C sc>

<C sc> = <condition C> + max(<inside C>, <else>)

<else> = 0

<inside C> = 2

<condition C> = 2

<C sc> = 2 + max(2,0) = 2 + 2 = 4 = <inside B>

<B sc> =

n
$\Sigma\ (4) \quad = \quad 4(n - i) = 4n - 4i$
j = i + 1

<A sc> =

0	0	0
Σ<B sc> =	$\Sigma 4n$ -	$\Sigma 4i$ =
i = n – 2	i = n – 2	i = n – 2

$4n(0 - n + 3) - 4(\ \frac{0*1}{2}\ ) = -4n^2 + 12n$ = <A sc>

sc for block D:

<D sc> = <duration D> * <inside D>

<duration D> = n – 0 + 1 = n + 1

<inside D> = <E sc>

<E sc> = <condition E> + max(<inside E>, <else>)

<else> = 0

<condition E> = 1

<inside E> = 3

<E sc> = 1 + max(3,0) = 1 + 3 = 4 = <inside D>

<D sc> = 4(n + 1) = 4n + 4

total sc for End-to-Beginning:

<total sc> =
<A sc> + <D sc> + <sc outside blocks> =
$- 4n^2 + 12n + 4n + 4 + 10 =$
$-4n^2 + 16n + 14 \rightarrow$
$O(n^2)$

The efficiency class for End-to-Beginning is $O(n^2)$.

Exhaustive Algorithm:

```
def longest_nondecreasing_powerset(A)
{//function

        n = A.size()

        create a sequence called best

        create a stack called stack

         k = 0

        //find the best subsequence
        while (2^n times)          //block A
                {//while

                if stack[k] < n            //block B
                        {//if

                        stack[k+1] = stack[k] + 1

                        ++k

                        }//if               //block B

                else                       //block C
                        {//else

                        stack[k-1]++

                        k--;

                        }//else             //block C

                if k == 0                  //block D
                        {//if

                        break

                        }//if               //block D

                create a sequence called candidate
```

```
            for i = 1 to k do          //block E
                {//for

                candidate.push_back(current)

                }//for   //block E

            //check for a new best
            if (candidate is non-increasing) && (size of candidate > size of best)        //block F
                {//if

                //found a new best
                 best = candidate

                }//if     //block F
            }//while          //block A

    return best

}//function
```

Proof for Exhaustive Algorithm:

sc outside A:

<sc outside A> =
<n = A.size()> + <sequence best> + <stack> + <k = 0> + <return best> =
2 + 1 + 1 + 1 + 1 = 2 + 4 =
6

sc for block A:

<A sc> = <duration A> * <inside A>

<duration A> = 2^{n}

<inside A> = <B sc> + <D sc> + <sequence candidate> + <E sc> + <F sc>

<B sc> = <condition B> + max(<inside B>,<C sc>)

<condition B> = 1

<C condition> = 1

<C sc> = 1 + 2 = 3

<inside B> = 3

<B sc> = 1 + max(3,3) = 1 + 3 = 4

<D sc> = <condition D> + max(<inside D>,<else>)

<else> = 0

<condition D> = 1

<inside D> = 1

<D sc> = 1 + max(1,0) = 1 + 1 = 2

<inside E> = 1

<E sc> =

k	k
Σ<inside E> = Σ1 =	
i = 1	i = 1

1k = k = <E sc>

<F sc> = <condition F> + max(<inside F>,<else>)

<else> = 0

<condition F> = 2

<inside F> = 1

<F sc> = 2 + max(1,0) = 2 + 1 = 3

<inside A> =
4 + 2 + 1 + k + 3 = k + 4 + 3 + 3 = k + 10

<A sc> =

$(k + 10)*2^n$

k + 10 <= n + 10

$(k + 10)*2^n \rightarrow (n + 10)*2^n$

Total sc:

<total sc> =
<sc A> + <sc outside A> =
$<(n + 10)*2^n> + 6 \rightarrow$
$O((n)*2^n)$

The efficiency class for the Exhaustive Algorithm is $O((n)*2^n)$.

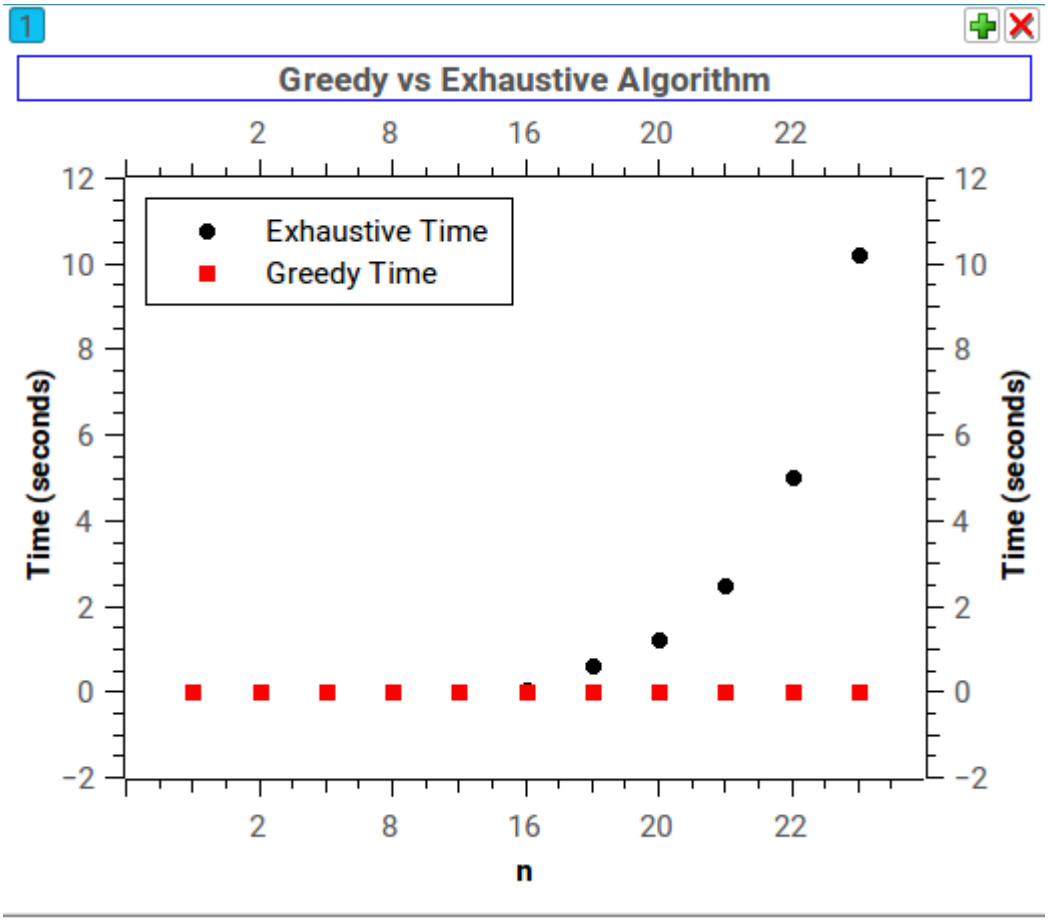
Screenshot of command line:

```
carla@scotchmallow:~/Desktop/cpsc 335 proj2$ make
g++ -std=c++14 -Wall subsequence_timing.cpp -o subsequence_timing
g++ -std=c++14 -Wall subsequence_test.cpp -o subsequence_test
./subsequence_test
end-to-beginning examples: passed, score 2/2
end-to-beginning additional cases: passed, score 1/1
powerset examples: passed, score 2/2
powerset additional cases: passed, score 1/1
TOTAL SCORE = 6 / 6
```

Data:

	Greedy	Exhaustive
n = 1	5.329e-06 = 0.000005329	2.548e-06 = 0.000002548
n = 2	3.696e-06 = 0.000003696	3.2788e-05 = 0.000032788
n = 4	4.679e-06 = 0.000004679	1.4613e-05 = 0.000014613
n = 8	6.147e-06 = 0.000006147	0.000235571
n = 12	5.132e-06 = 0.000005132	0.00426263
n = 16	9.155e-06 = 0.000009155	0.0748804
n = 19	7.939e-06 = 0.000007939	0.622448
n = 20	7.525e-06 = 0.000007525	1.23323
n = 21	8.153e-06 = 0.000008153	2.51395
n = 22	9.36e-06 = 0.00000936	5.03063
n = 23	9.613e-06 = 0.000009613	10.1992

Scatter Plot:



This graph shows that the greedy algorithm is significantly faster than the exhaustive algorithm. This is consistent with the efficiency classes because the greedy algorithm's complexity is $O(n^2)$, which is much faster than the $O(n*2^n)$ complexity of the exhaustive algorithm.

The greedy algorithm is much faster than the exhaustive algorithm, by a factor of $\frac{2^n}{n}$. It makes sense for the greedy algorithm to be faster than the exhaustive algorithm because an exhaustive algorithm will examine all of the data, while a greedy algorithm will examine data until it finds a candidate that satisfies the requirements for the algorithm.

The hypothesis that the exhaustive algorithm will produce correct outputs is supported by the program passing its tests.

The hypothesis that algorithms with factorial or exponential run times are very slow and not very efficient is supported by the very slow execution of the exhaustive algorithm. At n = 24 and higher values, the exhaustive algorithm could not run on my laptop.