Project 2 Documentation: Solving the Longest Increasing Subsequence

Group members:

Wayne Lin waylin@csu.fullerton.edu

Readme to be inserted

Pseudocode for Longest Increasing Subsequence (Powerset)

Input: a sequence of numbers A of size n

Output: the longest increasing subsequence in A

```
def longest_increasing_powerset(A):
   n = A.size()
   sequence best
   stack = a vector containing (n+1) elements
   // stack[0] is ununsed, so there are actually n relevant elements
   k = 0
   while (true) do // repeat loop until break is called
      if (stack[k] < n) then
         stack[k+1] = stack[k] + 1
         ++k
      else
         stack[k-1]++
         k--
     endif
      if (k == 0) break
      sequence candidate
      for i = 1 to k do
         insert A[stack[i]-1] at the end of candidate
      endfor
      if (candidate is an increasing sequence) then
         if (best == sequence{0} || candidate.size() > best.size()) then
            best = candidate
         // else do nothing
         endif
```

```
      // else do nothing
    endif
  endwhile
return best
```

Mathematical Analysis   Step Count

```
def longest_increasing_powerset(A):
  n = A.size()                                                        // 2 steps
  sequence best                                                       // 1 step
  stack = a vector containing (n+1) elements                          // 2 steps
  // stack[0] is ununsed, so there are n relevant elements
  k = 0                                                               // 1 step
  while (true) do                                                     // 2^n times
/* each loop generates a candidate within the powerset, verifies it for being increasing and compares it
to the current best candidate until all the candidates have been generated and compared*/
    if (stack[k] < n) then                                            // 1 step
      stack[k+1] = stack[k] + 1                                       // 2 steps
      ++k                                                             // 1 step
    else
      stack[k-1]++                                                    // 1 step
      k--                                                             // 1 step
    endif
    if (k == 0) break                                                 // 1 step
    // else do nothing                                                // 0
    endif
    sequence candidate                                                // 1 step
    for i = 1 to k do                                  // at most n times since k <= n
      insert A[stack[i]-1] at the end of candidate                    // 2 steps
    endfor
```

```
    if (candidate is an increasing sequence) then                // candidate.size() <=  n steps
      if (best == sequence{0} | | candidate.size() > best.size()) then       // 5 steps
        best = candidate                                                      // 1 step
      // else do nothing                                                      // 0
      endif
    // else do nothing                                                        // 0
    endif
    // repeat loop until break condition is met
  endwhile
 return best                                                                  // 0
```

Step Count Calculation

Outside Operations: 2 + 1 + 2 + 1 = 6 steps

While loop:	2^n times (Powerset)
Set generation block: 1 + max(3, 2) =	4 steps
break check:	1 step
declare candidate:	1 step
For loop:	n times
inside for block:	2 steps
candidate verifier:	n steps
optimization comparison: max(5, 5+1) =	6 steps

Entire while block = 2^n * (4 + 1 + 1 + 2*n + 6*n) = 2^n * (8n + 6)

Entire algorithm = 2^n * (8n + 6) + 6 steps

Proof by definition that (2^n)(8n + 6) + 6 belongs to O(n*2^n):

Let f(n) = (2^n)(8n+6) + 6 and g(n) = n * 2^n.

Then, f(n) = O(g(n)) if there exists some c > 0 and n_0 >= 0 such that f(n) <= c * g(n) for all n >= n_0.

Choose c = 20. Then c * g(n) = 20 * n * 2^n

(2^n)(8n+6) + 6 <?= 20n * 2^n

Try n = 1:

(2^1)(8*1 + 6) + 6 <?= 20 * 1 * 2^1

(2)(14) + 6 <?= 20 * 2

28 + 6 <?= 40

34 <= 40 True

Thus, (2^n)(8n+6) + 6 <= c * (n)(2^n) for c = 20 and all n >= n_0 = 1.

Therefore, f(n) belongs to O(g(n)) = O(n * 2^n)

The longest increasing powerset algorithm runs in exponential time.

Proof by limit theorem that (2^n)(8n + 6) + 6 belongs to O(n*2^n):

Let T(n) = (2^n)(8n+6) + 6 and f(n) = n * 2^n.

Then, T(n) = O(f(n)) if lim_{n->+INF}(T(n)/f(n)) >=0 and a constant (not infinity)

lim_{n->+INF}(T(n)/f(n)) = lim_{n->+INF}[((2^n)(8n+6) + 6)/(n * 2^n)] //

= lim_{n->+INF}[((2^n)(8n+6) + 6)'/(n * 2^n)']

= lim_{n->+INF}[((8n * 2^n + 6 * 2^n) + 6)'/(2^n + n)']

= lim_{n->+INF}[(8 * 2^n + 8n * 2^n ln 2 + 6 * 2^n ln 2)/(n * 2^n ln 2 + 2^n)]

= lim_{n->+INF}[((2^n)(8 + 8n ln 2 + 6 ln 2))/((2^n)(n ln 2 + 1))] // divide by 2^n

= lim_{n->+INF}[(8 + 8n ln 2 + 6 ln 2)/(n ln 2 + 1)]

= lim_{n->+INF}[(8 + 8n ln 2 + 6 ln 2)'/(n ln 2 + 1)']

= lim_{n->+INF}[(8 ln 2)/(ln 2)] = 8 >= 0 and a constant

Since 8 >= 0 and a constant, T(n) belongs to O(f(n)).

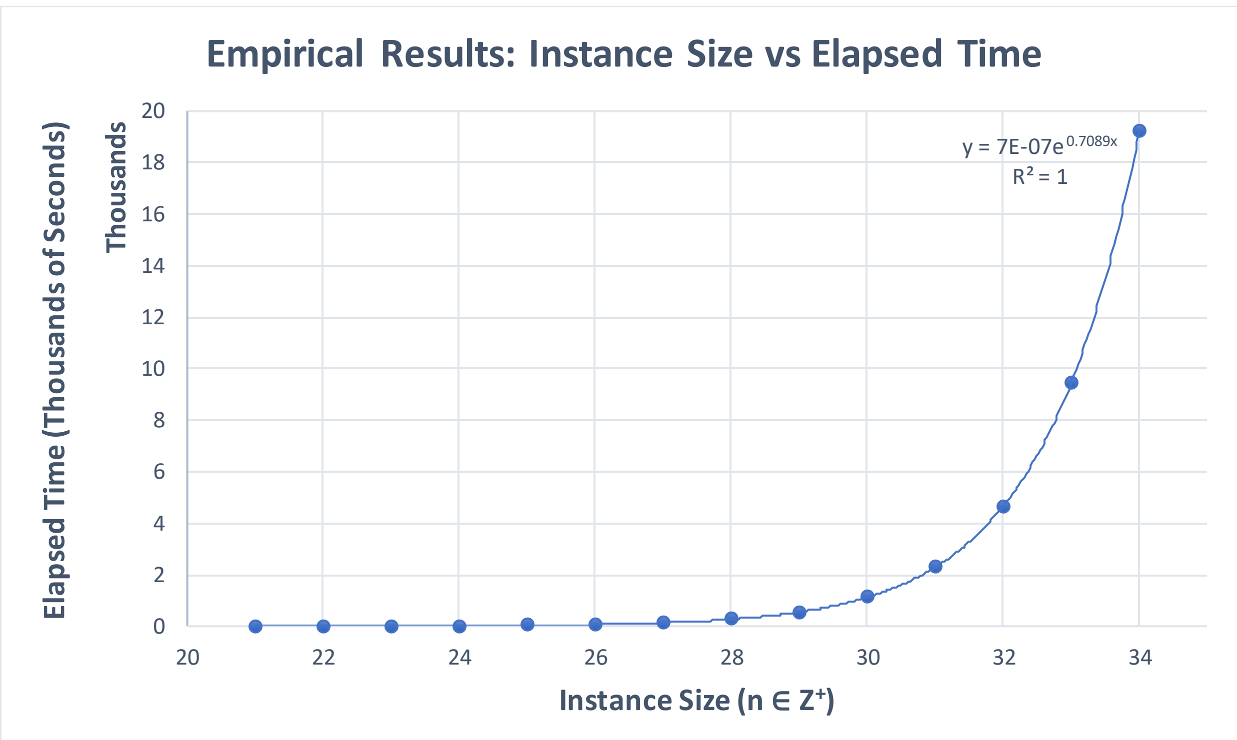
Thus, (2^n)(8n + 6) + 6 belongs to O(n * 2^n).

Our algorithm runs in exponential time.

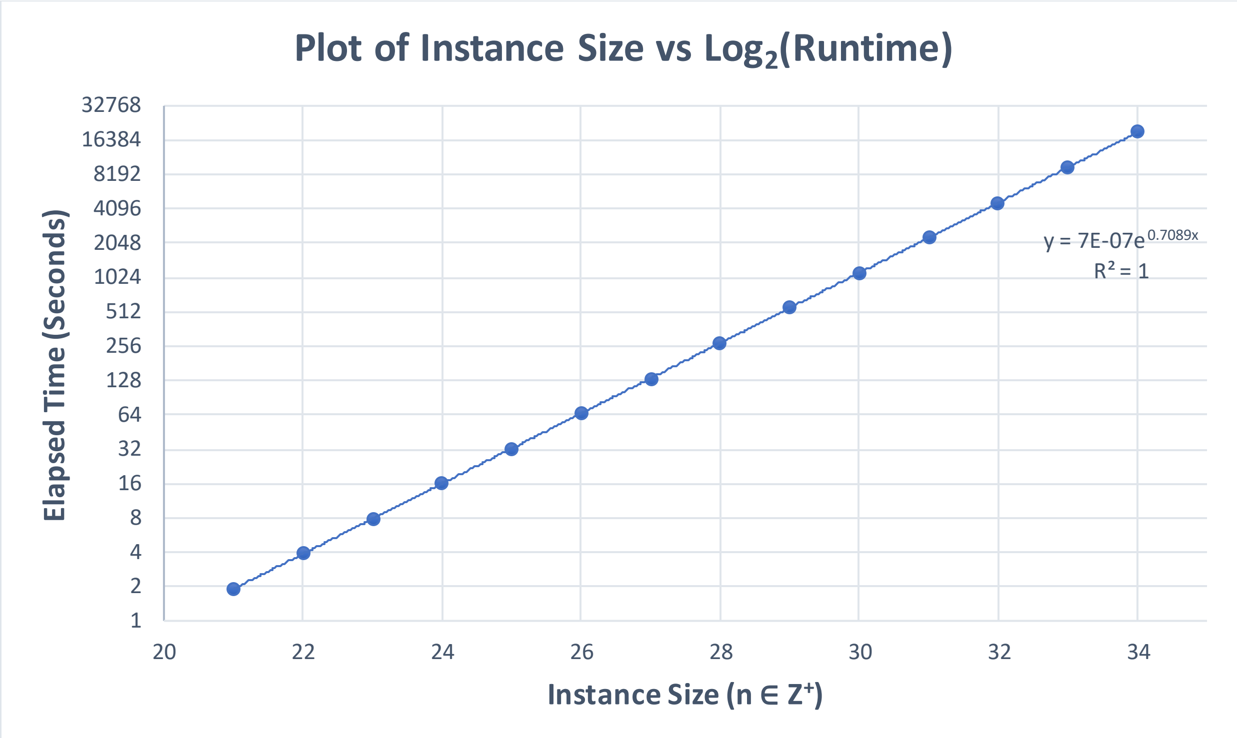
Empirical Analysis

The inputs chosen to represent the best-fit plot were those n-values for which the algorithm's elapsed time exceeded 1 second, but still completed within a (somewhat) reasonable amount of time.

The range of n-values from 21 to 34 led to runtimes over 1 second that were more suitable for empirical analysis. A trendline regression equation was calculated to be (7/10^7) * e^(0.7089n) with a strong correlation of R^2 = 1.



The functions 2^n and e^(0.7089n) are very similar, with e^0.7089 ~ 2.03. The $\log_2$ plot below delineates this near-perfect doubling of T(n) per incremental increase of n. Since O(n*2^n) grows slightly faster than O(2^n), the empirical results strongly align with my mathematical prediction that this algorithm belongs to O(n*2^n).



I hypothesized that it is likely that 2^n * (1 + cn), for some small positive constant c << 1, can closely approximate the function e^(0.7089n).

e^0.69315 is a very close approximation of 2^1.

2^n ~ e^(0.69315n) and e^(0.7089n) = e^(0.69315n) * e^(0.01575n). Therefore,

2^n ~ e^(0.69315n) * e^(0.01575n). The difference in growth is approximately e^(0.01575n) ~ 1 + 0.02n for our sample range of inputs.

Both approximations are at least 97% accurate for all tested values of n. The crucial approximation of 2^n ~ e^(0.69315n) is more than 99.99% accurate for any practical input of n.

Therefore, (7/10^7) * e^(0.7089n) ~ (7/10^7) * 2^n (1 + 0.02n)

We can drop the tiny constant factor 7/10^7 from both sides. This is dependent on CPU instructions per clock and clock speeds.

Thus, e^(.7089n) ~ (2^n)(1 + 0.02n) = O(n * 2^n)

Proof:

Let T(n) = (2^n)(1 + 0.02n) and f(n) = (n * 2^n).

If lim_{n->+inf}(T(n)/f(n)) >= 0 and a constant then T(n) = O(f(n)).

lim_{n->+inf}(T(n)/f(n)) = lim_{n->+inf}[(2^n)(1 + 0.02n) / (n * 2^n)] // divide by 2^n

= lim_{n->+inf}[(1 + 0.02n) / (n)]

= lim_{n->+inf}[(1 + 0.02n)' / (n)']

= lim_{n->+inf}(0.02 / 1) = 0.02 >= 0 and a constant.

Thus, (2^n)(1 + 0.02n) = O(n * 2^n)

From our step count analysis, (2^n)(8n + 6) + 6 also belongs to O(n * 2^n).

Evaluation of Hypotheses

Hypothesis 1 states: *For large values of n, the mathematically-derived efficiency class of an algorithm accurately predicts the observed running time of an implementation of that algorithm..*

Hypothesis 2 states: *Algorithms with exponential or factorial running times are extremely slow, probably too slow to be of practical use.*

The observed empirical results are strongly consistent with expectations from the mathematical derivation, both of which support the exponential complexity of this algorithm and the claim of Hypothesis 1. The best-fit regression trendline closely matches our prediction of the algorithm’s complexity. We can disregard constant multipliers, however infinitesimal, in asymptotic analysis because they depend on other factors such as computational speed. They do not influence the order of a function.

This algorithm is unbearably slow beyond n = 30 and useless around n = 40 compared to polynomial-time or faster algorithms, as seen in the results by the rapidly growing elapsed time with respect to the input size n. Our exponential algorithm solved inputs below n = 21 in under 1 second, n = 21 in 1.9 seconds, and n = 24 in 16 seconds, but for n = 34 it took over 5.3 hours. From the empirical data, an increase of 10 in the input size led to a 1200-fold increase in the runtime! This supports Hypothesis 2 which predicts that algorithms with exponential (or factorial) running times are so slow as to be virtually unusable for real-world scenarios. A more optimal polynomial or linearithmic time algorithm could solve even n = 50 in milliseconds in the worst case; our exhaustive optimization algorithm would still be running after 50 years!