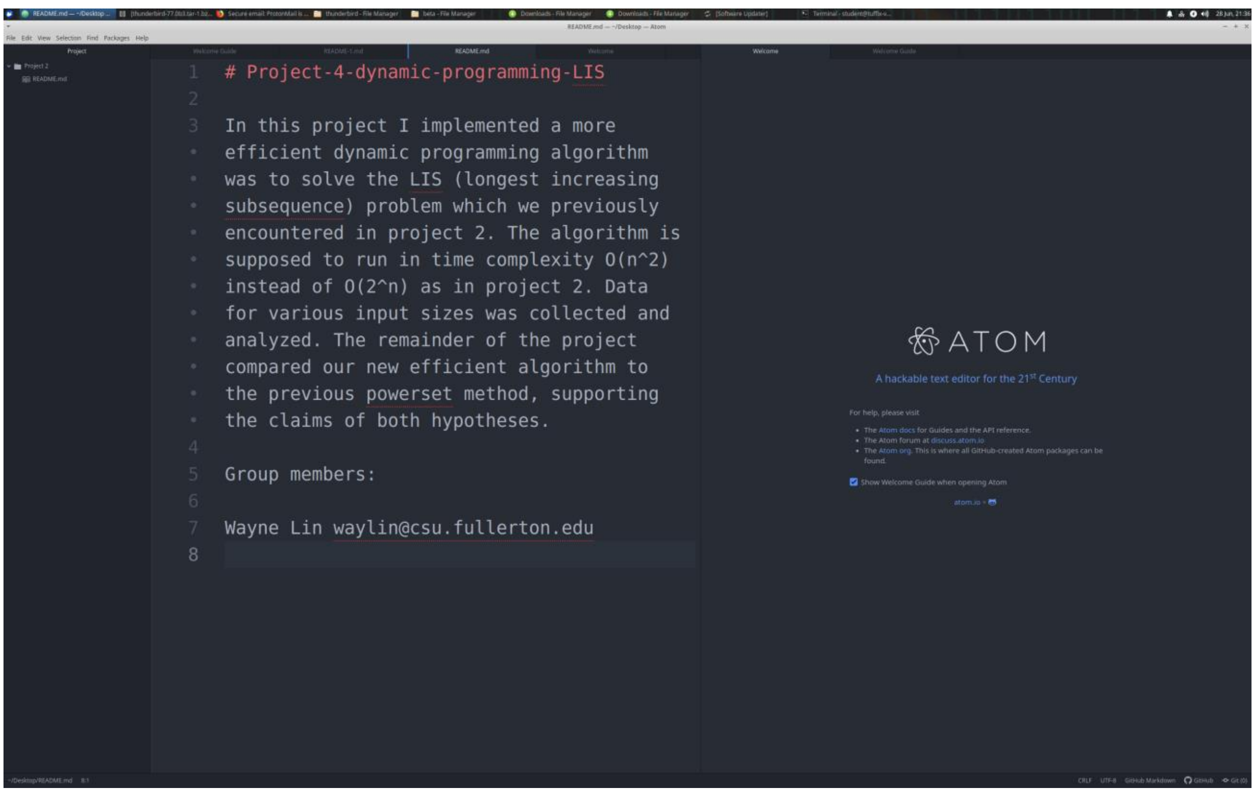
Project 4 Documentation: Longest Increasing Subsequence (LIS) Revisited

Group members:

Wayne Lin   waylin@csu.fullerton.edu



Pseudocode for LIS_(End-to-Beginning method) (Question a)

Input: a vector sequence of numbers A of size n

Output: the longest increasing subsequence in A

```
def lis_end_to_beginning(A):
        n = A.size()
        H = Array(n, 0)

        for i from n-2 to 0 step -1 do
                for j from i+1 to n-1 do
                        if A[j] > A[i] && H[j] >= H[i]
                                then H[i] = H[j] + 1
                        // else do nothing
                        endif
                endfor
        endfor

        max = (the max element in H) + 1
        int array R[max]
        index = max - 1
        j = 0
        for i from 0 to n-1 do
                if H[i] == index
                        R[j] = A[i]
                        index--
                        j++
                // else do nothing
        endfor

return R holding the longest increasing subsequence
```

Mathematical Analysis

Step Count

```
def lis_end_to_beginning(A):
    n = A.size()                                    // 2 steps
    H = Array(n, 0)                                 // 1 step

    for i from n-2 to 0 step -1 do                  // (n-1) times
        for j from i+1 to n-1 do                        // (n-i-1) times
            if A[j] > A[i] && H[j] >= H[i]              // 3 steps
                then H[i] = H[j] + 1                    // 2 steps
            // else do nothing                          // 0
            endif
        endfor
    endfor

    max = (the max element in H) + 1                // 3 steps
    int array R[max]                                // 1 step
    index = max - 1                                 // 2 steps
    j = 0                                           // 1 step
    for i from 0 to n-1 do                          // (n) times
        if H[i] == index                                // 1 step
            R[j] = A[i]                                 // 1 step
            index--                                     // 1 step
            j++                                         // 1 step
        // else do nothing                              // 0
    endfor

return R holding the longest increasing subsequence     // 0
```

Step Count Calculation

Outside Operations: $2 + 1 + 3 + 1 + 2 + 1 = 10$ steps

- Outer nested for loop: $(n-1)$ iterations
  - Inner nested for loop: $(n-i-1)$ iterations
    - if-else block: $\max(3+2, 3) = 5$ steps
- Simple for loop: (n) times
  - Simple loop block: $\max(1+1+1+1, 1) = 4$ steps

s.c. $= 10 + 4n + \sum_{i = 1}^{n-1}[\sum_{j = i+1}^{n-1}(5)]$

$= 10 + 4n + \sum_{i = 1}^{n-1}[(n-(i+1)-1+1)(5)]$

$= 10 + 4n + \sum_{i = 1}^{n-1}[(n-i-1)(5)]$

$= 10 + 4n + \sum_{i = 1}^{n-1}(5n-5) - \sum_{i = 1}^{n-1}(5i)$

$= 10 + 4n + (5n-5)(n-1) - 5 * \sum_{i = 1}^{n-1}(i)$

$= 10 + 4n + (5n-5)(n-1) - 5 * (n-1)(n)(1/2)$

$= 10 + 4n + 5n^2 - 5n - 5n + 5 - 5/2 * (n^2 - n)$

$= 5n^2 - 5/2n^2 + 4n - 10n - 5/2n + 5 + 10$

$= 5/2\, n^2 + 6n - 5/2n + 5 + 10$

$= 5/2\, n^2 + 17/2n + 15$

$= \frac{1}{2} (5n^2 + 17n + 30)$

Proof by definition that ½ $(5n^2 + 17n + 30)$ belongs to $O(n^2)$:

Let $f(n) = \frac{1}{2}(5n^2 + 17n + 30)$ and $g(n) = n^2$. 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 $c = 26$. Then $c * g(n) = 26 * n^2$.

$\frac{1}{2}(5n^2 + 17n + 30) <?= 26 * n^2$

$5n^2 + 17n + 30 <?= 52n^2$

$17n + 30 <?= 47n^2$

$47n^2 - 17n - 30 >= 0$ True when $n >= 1$

Thus, $\frac{1}{2}(5n^2 + 17n + 30) < c*n^2$ for $c = 26$ and all $n >= n_0 = 1$.

Therefore, we have proven that $f(n)$ belongs to $O(n^2)$.

The dynamic programming longest increasing subsequence algorithm runs in quadratic time.

Proof by limit theorem that ½ $(5n^2 + 17n + 30)$ belongs to $O(n^2)$:

Let $T(n) = \frac{1}{2}(5n^2 + 17n + 30)$ and $f(n) = n^2$.

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

Derivatives can be taken so long as the current limits give INF/INF, INF/0, 0/INF nondeterminate forms.

$\lim_{n\to+INF}(T(n)/f(n)) = \lim_{n\to+INF}[(\frac{1}{2}(5n^2 + 17n + 30) / (n^2)]$

$= \lim_{n\to+INF}[(5n^2 + 17n + 30) / (2n^2)] \Rightarrow INF/INF$

$= \lim_{n\to+INF}[(5n^2 + 17n + 30)' / (2n^2)']$

$= \lim_{n\to+INF}[(10n + 17) / (4n)] \Rightarrow INF/INF$

$= \lim_{n\to+INF}[(10n + 17)' / (4n)']$

$= \lim_{n\to+INF}(10 / 4) = 10/4 = 5/2 >= 0$ and a constant

Since $5/2 >= 0$ and a constant, $T(n)$ belongs to $O(f(n))$.

Thus, $\frac{1}{2}(5n^2 + 17n + 30)$ belongs to $O(n^2)$.

The Dynamic Programming end-to-beginning LIS algorithm runs in quadratic time.

Answer to Question b.

From my earlier mathematical analysis, the step count approximation was $\frac{1}{2}(5n^2 + 17n + 30)$. I proved using the definition method and limit theorem this algorithm belongs to the efficiency class $O(n^2)$.

Empirical Analysis

I selected a wide albeit informative range of n-values ranging between 30000 and 500000 for the input sizes. All input values used in the plot had running times between 3 seconds and 20 minutes to minimize error. The algorithm performed reasonably well and only began to slow down around n = 100000. Quadratic time remains "viable" up to n = 1 million or more if one is willing to wait the 1+ hours running time for such a result, but an even more efficient O(n log n) algorithm should ideally be used in those scenarios. The scatter plot is on the following page.

Answer to Question c.

Project 4 saw a gargantuan improvement in efficiency for the improved dynamic programming algorithm over the previous powerset algorithm. The exponential-time powerset algorithm was useless by n = 35. The polynomial-time DP algorithm remains practical beyond n = 100000. The staggering running time difference between the two algorithms was surprising but wholly expected from their computational complexities.

The DP algorithm is extremely fast in computing small values of n, and remains viable for much larger n-values than the powerset algorithm. To list a few examples, all inputs n < 500 took the DP algorithm <1ms, versus only n < 10 for the naïve algorithm. Computing for n = 15000 took about as long as the powerset algorithm took to compute n=20. The DP algorithm solved for n = 500000 in less time than the powerset algorithm solved for n = 30.

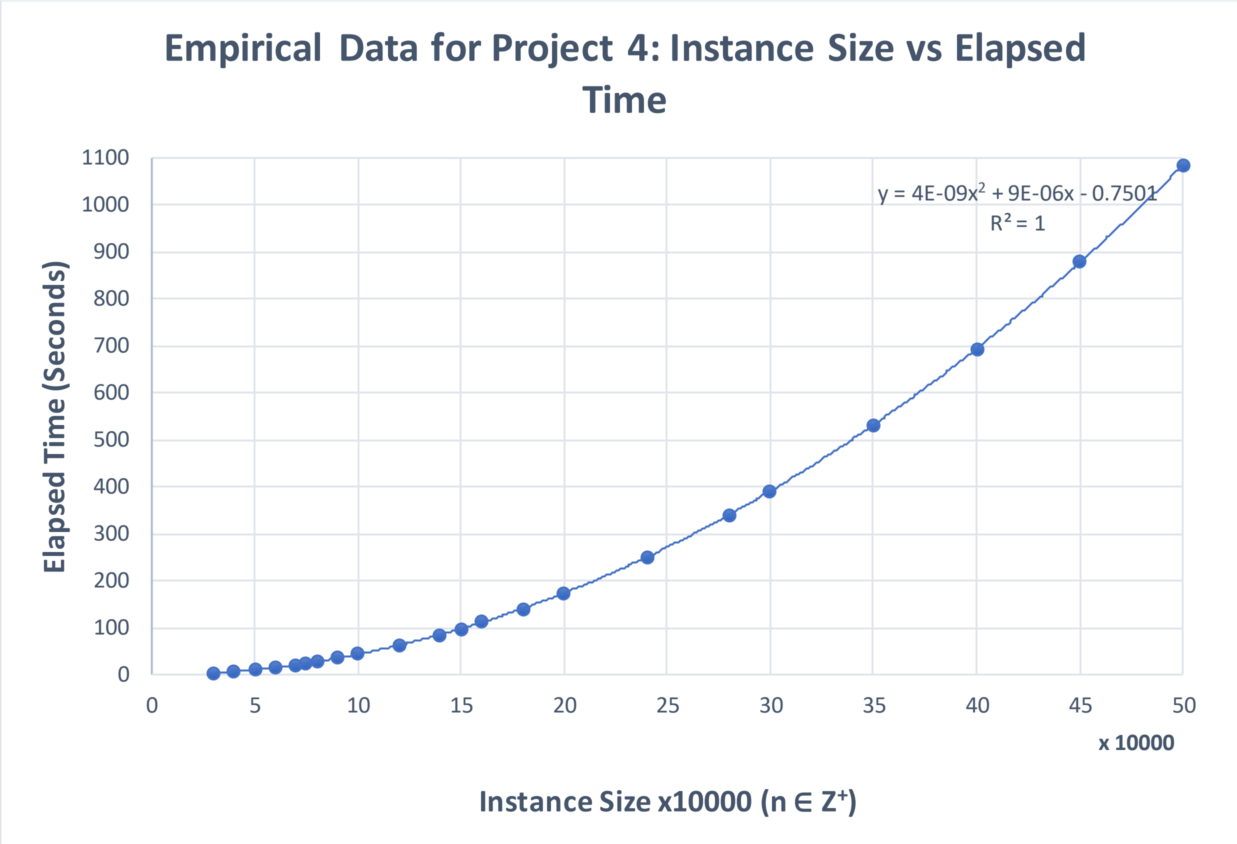
In project 2, I calculated the powerset algorithm to be O(n*2^n). Elapsed time would more than quadruple when n increased by 2. On the contrary, the dynamic programming algorithm can handle double the input by only quadrupling its running time.

In other words,

T(n+2) >≈ 4T(n) for the exponential-time powerset algorithm

T(2n) ≈ 4T(n) for the quadratic-time dynamic programming algorithm

T(2n) ≈ 2n(2^(2n)) = 2n*(2^(n))^2 ≈ (T(n))^2 for the exponential-time powerset algorithm



Best-fit line:

$y = 4E\text{-}09x^2 + 9E\text{-}06x - 0.7501$ is consistent with T(n) = O(n^2).

Evaluation of Scatter Plot and Efficiency Classes (Question d)

The empirical scatter plot yielded a quadratic regression line of $y = 4E\text{-}09x^2 + 9E\text{-}06x - 0.7501$ with $R^2 = 1$. The best-fit line is consistent with the mathematical prediction that the dynamic programming LIS algorithm should have time complexity O(n^2).

As in project 2, constants and constant value multipliers can be ignored in asymptotic analysis due to their reliance on CPU instruction and clock speeds. One can easily prove ax^2 + bx + c is upper bounded by (a+b+c)*x^2 and thus belongs to O(n^2).

Alternatively, taking the limit as n->INF of $y(n) = 4E\text{-}09n^2 + 9E\text{-}06n - 0.7501$ against O(n^2) and taking the derivative twice results in a tiny constant >= 0, proving that y(n) = O(n^2)

Evaluation of the Hypotheses and Conclusion (Question e)

The Hypotheses

1. 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..

2. Polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem.

The empirical results support Hypothesis (1) since every doubling of n yields approximately a quadrupling of runtime as expected for the mathematically derived $O(n^2)$ algorithm. This pattern can be seen for other multipliers as well, such as $T(5n) \approx 5^2(T(n)) = 25T(n)$, or $T(10n) \approx 10^2(T(n)) = 100T(n)$.

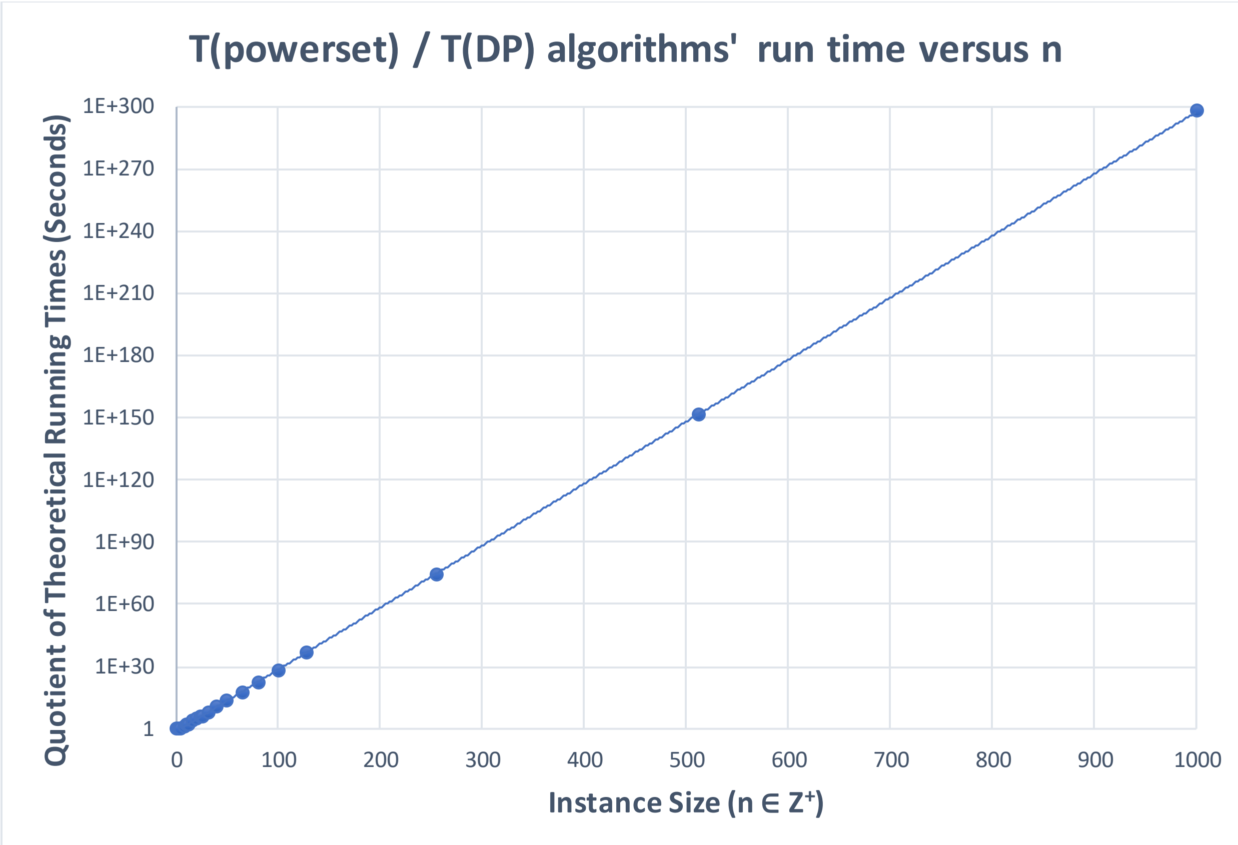
Hypothesis (2) vastly understates the difference between polynomial-time and exponential-time algorithms. For any values of n above 10, our $O(n^2)$ dynamic programming algorithm would solve the problem long before the powerset algorithm does - if it does so at all. The empirical data shows that using the $O(n^2)$ LIS algorithm, we can compute inputs of up to n=100,000 relatively efficiently. The powerset algorithm cannot even compute n=50 in 50 years.

If Moore's Law holds, a processor 3-4 years from now will be 4x as fast and able to compute twice the input of a quadratic algorithm in the same amount of time. An exponential algorithm would only be able to compute a size (n+2) input in the same amount of time. Therefore, polynomial-time DP algorithms are much more efficient than exponential-time exhaustive brute force algorithms that solve the same problem.

The difference in growth rates between $n * 2^n$ (powerset) and n^2 (dynamic programming) is so drastic that taking their quotient results in the following equation.

$$f(n) = T(powerset(n))/T(DP(n)) = n(2^n) / n^2 = 2^n / n$$

The below plot illustrates the struggle of exponential-time algorithms and why they fall out of favor once a polynomial-time solution is found. NP-complete problems such as 0-1 Knapsack (exponential in W), TSP and 3-SAT currently have no such easy solutions.



A large gap in efficiency and practical feasibility divides algorithms that can solve a problem in polynomial time versus those that require superpolynomial time (primarily exponential and factorial). Since the LIS problem can be solved in O(n^2) time or even O(n log n) time via binary search, it is deemed tractable and in class P. The powerset algorithm implemented in project 2 is only informative for comparison and useless in practice.