Algorithm Engineering

Project 2: Polynomial Versus Exponential Time
December 8, 2025

Marinah Rubalcaba
mrubalcaba@csu.fullerton.edu

Professor: Dr. Kevin Wortman

Submitted to the lecturer of CPSC 335 in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

CSUF | COLLEGE OF Engineering and Computer Science

Mathematical Analysis

Maximum Subarray Exhaustive Search

```
maximum_subarray_exh(L):
   b = 0, e = 1
   for i from 0 to n-1:
      for j from i+1 to n:
         if sum(L[i:j]) > sum(L[b:e]):
            b = i
            e = j
   return (b, e)
```

Inner Loop: $(n-1)(n+2) = n^2 + 2n - n - 2 = n^2 + n - 2$
Outer Loop: $n(n^2 + n - 2) = n^3 + n^2 - 2n$
Entire Algorithm: $n^3 + n^2 - 2n + 1 + 1 = n^3 + n^2 + 2$
$T(n) = n^3 + n^2 + 2$

Lemma: $n^3 + n^2 + 2 \in O(n^3)$
Proof: By properties of O,

$$\begin{aligned} n^3 + n^2 + 2 &\in O(n^3 + n^2 + 2) && \textit{trivial} \\ &\in O(n^3) && \textit{dominating term} \end{aligned}$$

Conclusion: maximum_subarray_exh takes $O(n^3)$ time.

Maximum Subarray Decrease-by-Half

Note: Because maximum_subarray_dbh calls maximum_subarray_recurse, and maximum_subarray_recurse calls maximum_subarray_crossing, proofs were performed for each function in sequence and used to analyze the others.

```
maximum_subarray_crossing(L, low, middle, high):
   left_sum = right_sum = $-\infty$
   sum = 0
   for i from middle down to low:
      sum += L[i]
      if sum > left_sum:
         left_sum = sum
         b = i
   sum = 0
   for i from middle + 1 to high:
      sum += L[i]
      if sum > right_sum:
         right_sum = sum
         e = i
   return (b, e + 1)
```

$T(n) = 4 + 5n + 5(n-1) = 4 + 5n + 5n - 5$
$T(n) = 10n - 1$

Lemma: $10n - 1 \in O(n)$
Proof: By properties of O,

$$\begin{aligned} 10n - 1 &\in O(10n - 1) && \textit{trivial} \\ &\in O(n) && \textit{dominating term} \end{aligned}$$

Conclusion: maximum_subarray_crossing takes $O(n)$ time.

```
maximum_subarray_recurse(L, low, high):
   if low == high:
      return (low, low + 1)
   middle = (low + high) / 2
   entirely_left = maximum_subarray_recurse(L, low, middle)
   entirely_right = maximum_subarray_recurse(L, middle + 1, high)
   crossing = maximum_subarray_crossing(L, low, middle, high)
   return the best of:  entirely_left, entirely_right, and crossing
```

Since this function is recursive, I will use the Master Method to prove its efficiency class.

General Case: $T(n) = 3 + 2T\left(\frac{n}{2}\right) + n \qquad \forall\, n > 1$
Base Case: $T(n) = 2 \in O(1) \qquad \forall\, n = 1$

$c(n) = 3 + n \in O(n)$
$r = 2 \qquad \geq 1$ ✓
$d = 2 \qquad \geq 1$ ✓
$t = 1 \qquad \geq 0$ ✓
$k = 1$, because $c(n) \in O(n) = O(n^1) \geq 0$ ✓

Comparing the value of r to the value of d^k, we get

$$2 = 2$$

Therefore, by Case 2 of the Master Theorem

$$T(n) \in O(n^k \log n)$$

Plugging in the value of k gives

$$O(n^1 \log n) = O(n \log n)$$

Conclusion: maximum_subarray_recurse takes $O(n \log n)$ time.

```
maximum_subarray_dbh(L):
   return maximum_subarray_recurse(L, 0, n-1)
```

Since maximum_subarray_dbh makes one function call to maximum_subarray_recurse, its runtime will be the same as maximum_subarray_recurse's runtime, which was proved above to be $O(n \log n)$.

Conclusion: maximum_subarray_dbh takes $O(n \log n)$ time.

Subset Sum Exhaustive Search

```
subset_sum_exh(U, t):
   for candidate in subsets(U):
      if |candidate| > 0 and sum(candidate) == t:
         return candidate
   return None
```

$T(n) = n \times 2^n + 1$

Lemma: $n \times 2^n + 1 \in O(n \times 2^n)$
Proof: By properties of O,

$$\begin{aligned} n \times 2^n + 1 &\in O(n \times 2^n + 1) && \textit{trivial} \\ &\in O(n \times 2^n) && \textit{dominating term} \end{aligned}$$

Conclusion: subset_sum_exh takes $O(n \times 2^n)$ time.

Empirical Analysis: Scatter Plots

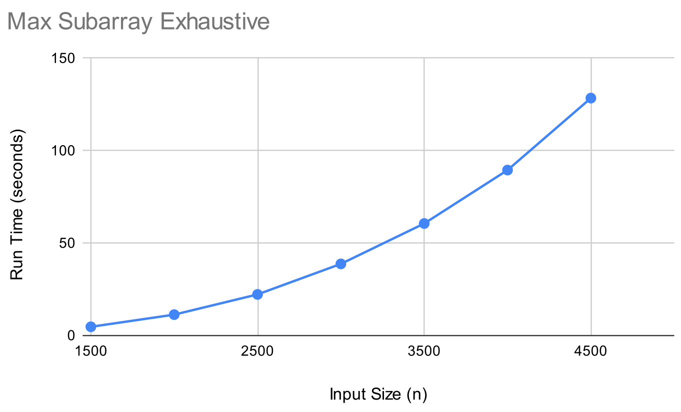

Figure 1: Maxiumum Subarray Exhaustive Search Scatter Plot

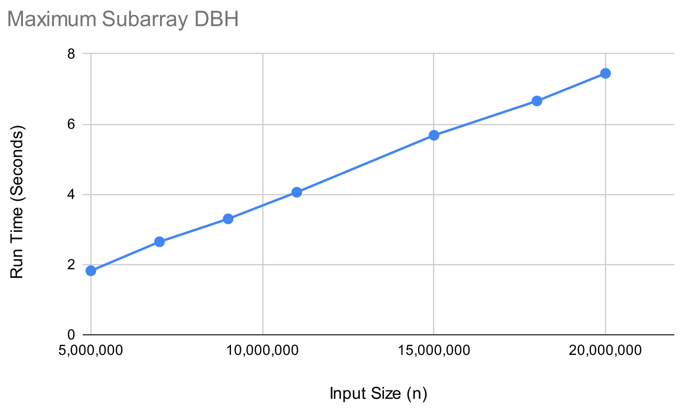

Figure 2: Maxiumum Subarray Decrease-by-Half Scatter Plot

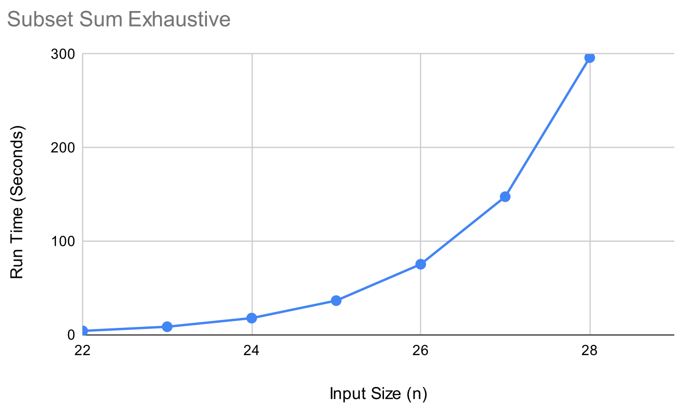

Figure 3: Subset Sum Exhaustive Search Scatter Plot

Question Answers

1. Is there a noticeable difference in the performance of the three algorithms? Which is fastest, and by how much? Does this surprise you?

There is a noticeable difference in the run time of each of the three algorithms in this project. The *Maxiumum Subarray Decrease-by-Half* algorithm has the fastest run time at $O(n \log n)$. For very large values of n, the run time is only a few seconds. However, for *Maximum Subarray Exhaustive Search*, which runs at $O(n^3)$ and *Subset Sum Exhaustive Search*, which runs at $O(n \times 2^n)$, larger, and even smaller, values of n take minutes to run. One value of n I used for *Subset Sum Exhaustive Search* for example was 27, which had a run time of 147 seconds. For *Maximum Subarray Exhaustive Search*, one value of n I tested was 2500 which had a run time of around 22 seconds. For *Maximum Subarray Decrease-by-Half*, however, I tested the very large n value of 50,000,000 and the run time was only 1.8 seconds. Despite this being the largest input value compared to the others, the run time was the fastest. This difference in performance is very surprising to see. Recursive functions seem as though they would be very slow due to having to call themselves several times, but, as evidenced by this project, that is far from true.

2. Are your empirical analyses consistent with the predicted big-O efficiency class for each algorithm? Justify your answer.

My empirical analyses are consistent with the predicted big-O efficiency classes for each of the three algorithms. The fit line for the *Maximum Subarray Exhaustive Search* algorithm has an increasing slope as the value of n grows. This is consistent with the predicted $O(n^3)$ efficiency class. The fit line for the *Maximum Subarray Decrease-by-Half* algorithm is nearly linear with an upward bend. This matches its predicted $O(n \log n)$ efficiency class. The fit line of the last algorithm, *Subset Sum Exhaustive Search*, has an exponential growth, which matches the shape of its predicted $O(n \times 2^n)$ efficiency class.

3. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.

The first hypothesis for this project is: "Exhaustive search algorithms are feasible to implement, and produce correct outputs." After creating exhaustive search algorithms for two of the three algorithms in this project, *Maximum Subarray Exhaustive Search* and *Subset Sum Exhaustive Search*, I can conclude that this hypothesis is true. I was able to successfully implement these algorithms using the pseudocode provided. Additionally, both algorithms produced correct outputs as verified by the tests in the testing file for this project.

4. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

The second hypothesis in this project is: "Recursion is not always slow, because recursion-based algorithms can be faster than loop-based algorithms." The evidence found in this project is consistent with this hypothesis. My mathematical and empirical analyses both found *Maxiumum Subarray Decrease-by-Half*, which is a recursive-based algorithm, to be $O(n \log n)$. I tested seven very large values of n for the algorithm (from 50,000,000 to 200,000,000) and all of them had a run time of less than 8 seconds. Comparing this to the loop-based algorithm *Maximum Subarray Exhaustive Search*, which was found to be $O(n^3)$, it is much faster. I had to use significantly smaller values of n (from 1500 to 4500) for *Maximum Subarray Exhaustive Search* as the run time got increasingly slow. One specific value of n I used was 4000, which had a run time of around 89 seconds. For *Maximum Subarray Decrease-by-Half*, however, for an n value of $70,000,000$ the run time was merely 2.6 seconds.

5. Is this evidence consistent or inconsistent with hypothesis 3? Justify your answer.

The third hypothesis in this project is: "Algorithms with exponential or factorial running times are extremely slow, probably too slow to be of practical use." The evidence from my analyses proved this hypothesis to be true. The *Subset Sum Exhaustive Search* algorithm runs at exponential time, specifically $O(n \times 2^n)$. I could not use large values of n for my empirical analysis on this algorithm as the run time was far too slow. The seven values of n I used to test the run time of the algorithm were between 22 and 28, with the last value still producing a run time of 296 seconds.