TCSS 343, Winter 2021

Programming Assignment (HW4)

Riley Ruckman, Travis Shields, and Jot Virdee

3/12/2021

Contents

Overview

The goal of this homework assignment was to gain experience in implementing and analyzing algorithms. We were tasked with coding solutions to the NP-Complete subset sum problem, where the subset sum problem is defined as following:

Subset Sum (SS):

Input: a list $S[0 \ldots n-1]$ of n positive integers, and a target integer $t \geq 0$.

Output: TRUE and a set of indices $A \subseteq \{0 \ldots n-1\}$ such that $\sum_{i \in A} S[i] = t$ if such an A exists, and FALSE otherwise.

There are three different solutions we used to solve this problem:

- A brute force solution (i.e., finding the powerset of the input set)
- A dynamic programming solution
- A provided "clever" algorithm

Testing Methodology:

The methodology of testing was the same for all three algorithms. First, we started by creating a list filled with random integers within a specified range. The two ranges assessed were r = 1,000 and r = 1,000,000. We also provided the option to guarantee there will or will not be a solution to the given target value. In the case of a guaranteed solution, a random number of random elements were taken from the input list and summed up to create the target value. When there should not be a solution, we summed the entire list and added one to guarantee there cannot be a subset that meets the target value. Every time the driver is executed, it runs all three algorithms and returns their runtime and amount of table space used. We assessed each algorithm with increasing values of n until the runtime of that algorithm reached five minutes. Everything was written in Python, and data was gathered on a Windows 10 machine with a Ryzen 5 3600 @ 4.0GHz.

Brute Force Analysis

The brute force approach involves computing all possible subsets of the input to then find the target value *t*. Computing all subsets yield a total of 2^n combinations when there is no solution. The issue with this is approach is the number of combinations becomes extremely large as *n* increases.

Also, the behavior of the algorithm is almost random when there is a solution in the subset equal to a *t* value (as shown in the appendix). Furthermore, if the solution is the first element in the input list, the algorithm is efficient, yielding a constant runtime. Conversely, if the solution is the sum of the input list, the algorithm's runtime will be $\Theta(2^n)$.

The provided figures paint a better visualization for the reader. Also, the analysis of the pseudocode will give a better understanding of the runtime and space complexity.

```
SSBruteForce(S[1…n], t):
    Let x = 2^n
    For counter = 1 to x do:                    A
        Let sublist = [] an empty array
        Let sum = 0
        For j = 0 to n-1 do:                    B
            If counter & (1 << j) > 0:
                sublist = sublist || (j + 1)
                sum = sum + S[j + 1]
        End For
    If sum == t:
        Return True, sublist
    End For
Return False, []
```

Time Complexity:
The runtime is based on the two for loops in the algorithm: **A** and **B**. Since all possible subsets need to be checked, the loop iterates from 1 to 2^n. Inside for loop **A**, another for loop **B** is nested that will iterate through the list and sum up the total for each subset. The rest of the code contributes a minuscule amount of time to the total runtime, which is ignored in the analysis.

Contribution from each loop (worst case):

Loop **A** and **B** (nested): $\sum_{i=1}^{2^{\wedge}n} \sum_{j=1}^{n} a = \sum_{i=1}^{2^{\wedge}n} an = a * n * 2^n = \Theta(n * 2^n)$

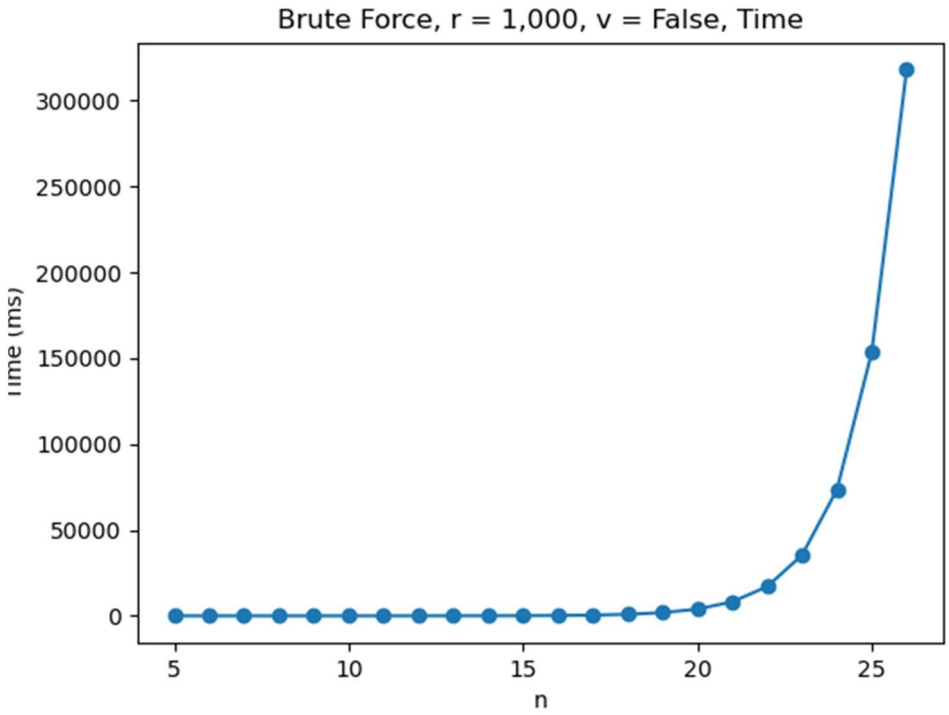


Figure 1: Time Complexity of Brute Force Algorithm with r = 1,000 and v = False

From the plot above, the runtime complexity matches the result of $\Theta(n * 2^n)$ from the analysis. With our hardware setup, we achieved an n value of 26 which is the lowest n value in the algorithms assessed. Furthermore, the algorithm ran in a respectable manner from n equaling 0 to 20. In this range, the results took less than a second to run.

Space Complexity:
In the worst case, the space needed to store the sub list will be the same space needed for the input list, which will be n. Given that the algorithm also needs n amount of space for the input list itself, the total amount of space required, in worst-case, is $2n$. Therefore, the space complexity is $\Theta(n)$.

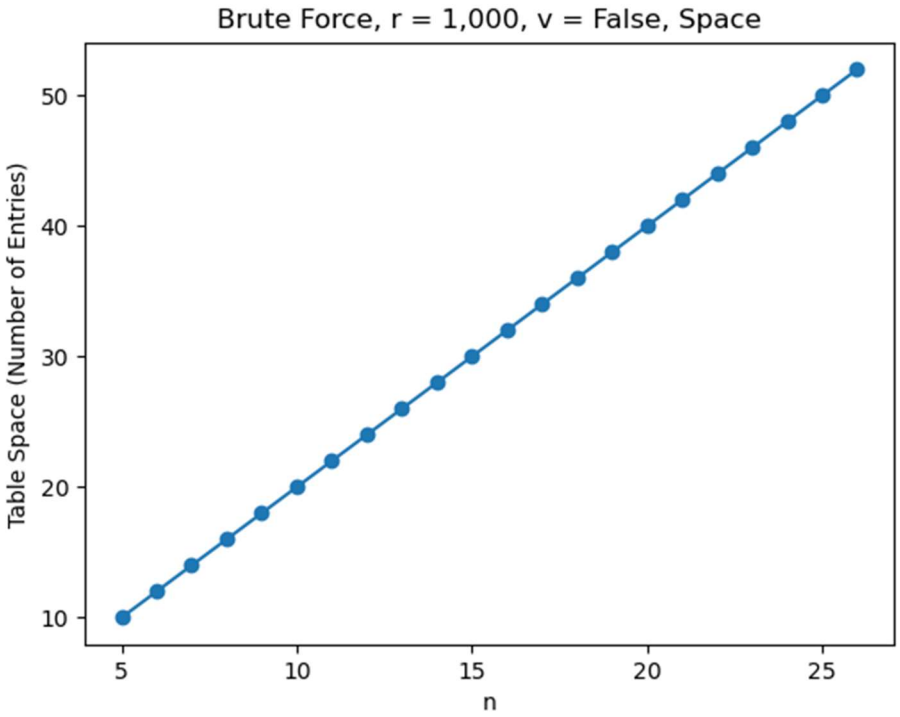


Figure 2: Space Complexity of Brute Force Algorithm with r = 1,000 and v = False

Since the worst case is being computed for each n value, the space taken is the length of the input array. Out of all algorithms, brute force takes the least amount of space. This tradeoff does not give any benefits unless the user's hardware is severely limited on RAM. Consequently, the user must run n values less than around twenty to receive a benefit.

Dynamic Programming Analysis

Dynamic programming is a possible solution to the large runtime complexities of this problem. In certain cases, such as computing Fibonacci numbers, dynamic programming outperforms the recursive algorithm (its slower alternative). Because of this, we hypothesis that the dynamic programming solution will run significantly faster than the other algorithms.

Our hypothesis quickly turned false as shown in the analysis below. The dynamic programming may even be a less optimal solution for users with a low amount of RAM because of the high memory usage.

To determine the complexity of an iterative algorithm, we need to evaluate the pseudocode and figure out how n and t relate to the total number of operations.

SSDynamic($S[1 \dots n], t$)

Let $A[1 \dots n][0 \dots t]$ be an array of integers of size $n * (t+1)$.

For i = 1 to n do: **A**

Let A[i][0] = True.

End For

For j = 1 to t do: **B**

If S[1] = j Let A[1][j] = True.

Else Let A[1][j] = False.

End For

For i = 2 to n do: **C**

For j = 1 to t do: **D**

If j ≥ S[i] Let A[i][j] = A[i-1][j] or A[i-1][j-S[i]].

Else Let A[i][j] = A[i-1][j]

End For

End For

If $A[n][t]$ = True:

Let *subset* = [ ]

For i = n to 1 do: **E**

If A[i-1][t-S[i]] = True:

Let $t = t - S[i]$

Let *subset* = *subset* || *i*

Let *subset* = reverse(*subset*)

Return True, *subset*

End For

Return False, []

End SSDynamic

Time Complexity:
Contribution from each loop (worst case):

A: This step is trivial; we are setting *i* amount of indices to True because there is always a solution to summing up to zero (i.e., the empty set). This takes *n* amount of time.

B: If the first index of the input list is equal to any value from 0 to *t*, then that subproblem has a solution. This takes *t* amount of time.

C+**D**: Here we are checking if there is a solution to the current subproblem, where the sum being found is equal to *j*. This is done in time *n*t*.

E: When there is a solution, we recover the correct subset, which takes *n* time.

In the worst case the runtime of this algorithm is equal to the sum of the four sections discussed above, yielding:

$n + t + n * t + n \in \Theta(t * n)$

However, we can get the entire runtime in terms of *n* if we make some assumptions about *t*.

In some cases, $t \approx n$, giving us: $3n + n^2 \in \Theta(n^2)$

For some inputs, the dynamic solution to subset-sum $\in \Theta(n^2)$

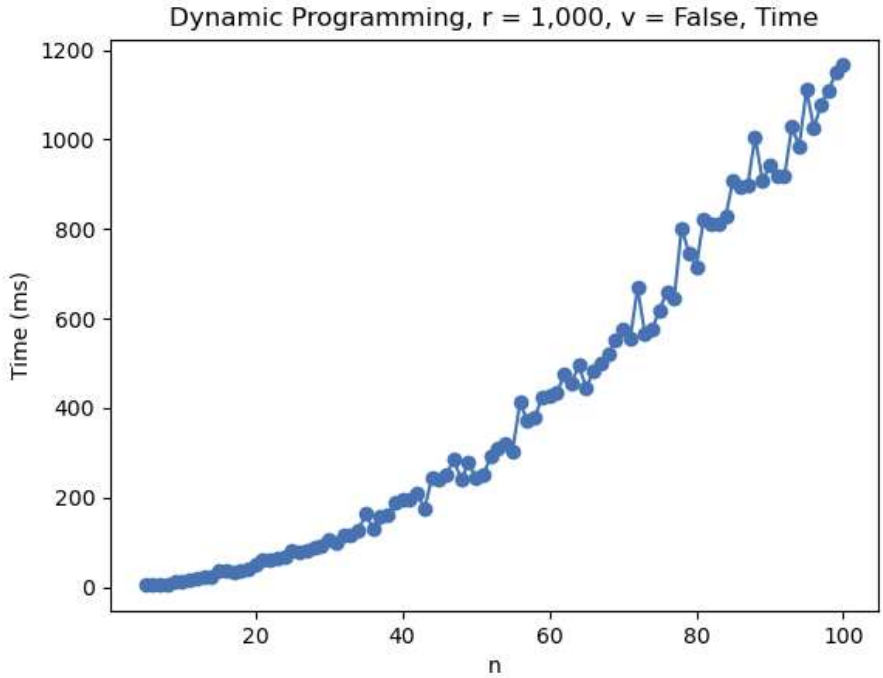


Figure 3: Time Complexity of Dynamic Algorithm with r = 1,000 and v = False

Since the target is found using a random subset of the input list, a lower range will yield a lower target on average. The above plot shows how quickly the dynamic programming solution can solve the subset sum problem for a relatively low range of input values.

In most cases, t is in the range of an n-bit integer, meaning $t \in \Theta(2^n)$, giving us:

$2n + 2^n + n * 2^n$

For most inputs, the dynamic solution to subset-sum $\in \Theta(n * 2^n)$

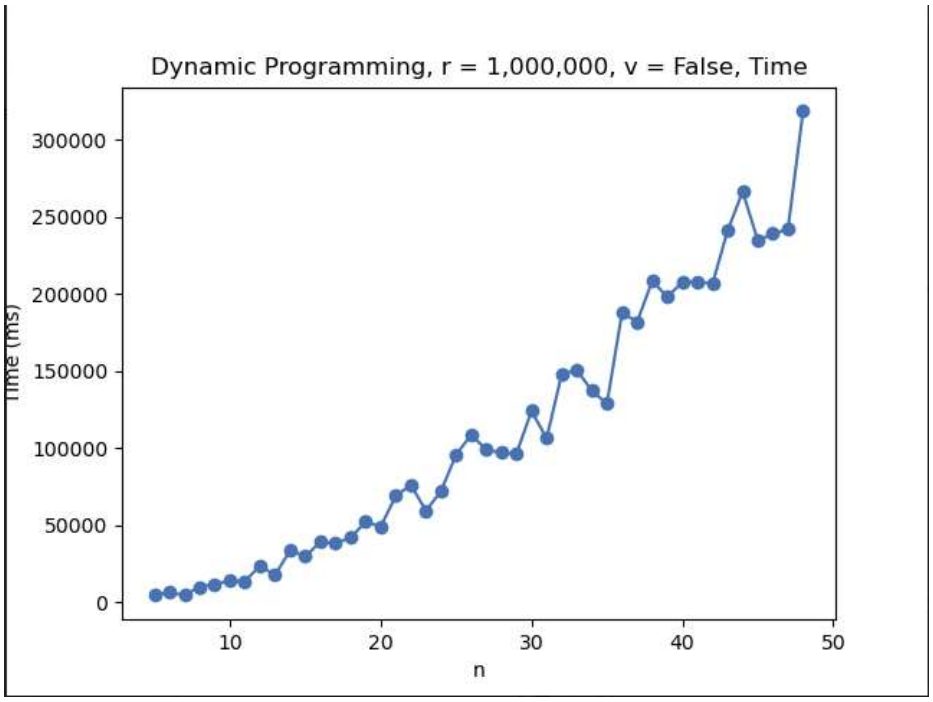


Figure 4: Time Complexity of Dynamic Algorithm with r = 1,000,000 and v = False

The above figure shows the impact of having larger target values: it took 5 minutes to determine there was no solution for n = 48, while with r = 1,000, the algorithm reached n = 100 and still was taking under 2 seconds.

From this data, our hypothesis was incorrect. The analysis shows a $\Theta(n * 2^n)$ runtime complexity. This was to our surprise; we expected a runtime of $\Theta(n)$ like the dynamic programming implementation of computing Fibonacci numbers.

Space Complexity:
The algorithm needs space for the input, space for the computed table, and the space held by the subset when there is a solution (which will be the complete list in the worst case).

$n + n * t + n$

In the average case, $t \approx n$, we get:

$2n + n^2 \in \Theta(n^2)$

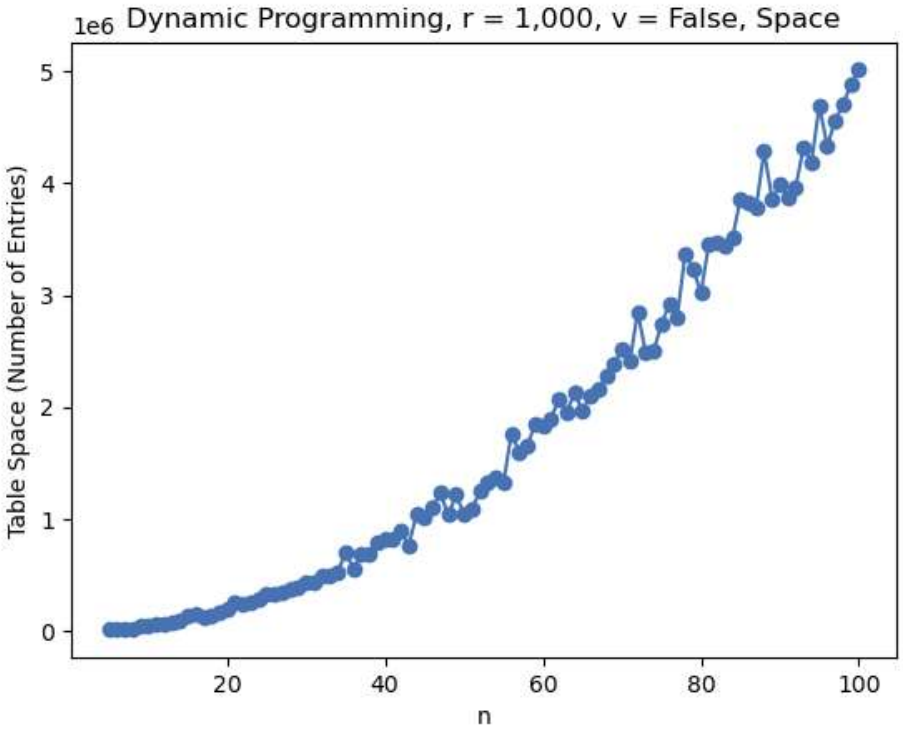


Figure 5: Space Complexity of Dynamic Algorithm with r = 1,000 and v = False. Note:

In most cases, where t ∈ $\Theta(2^n)$, we have:

$2n + n * 2^n$, for most inputs, the space used by the dynamic solution ∈ $\Theta(n * 2^n)$

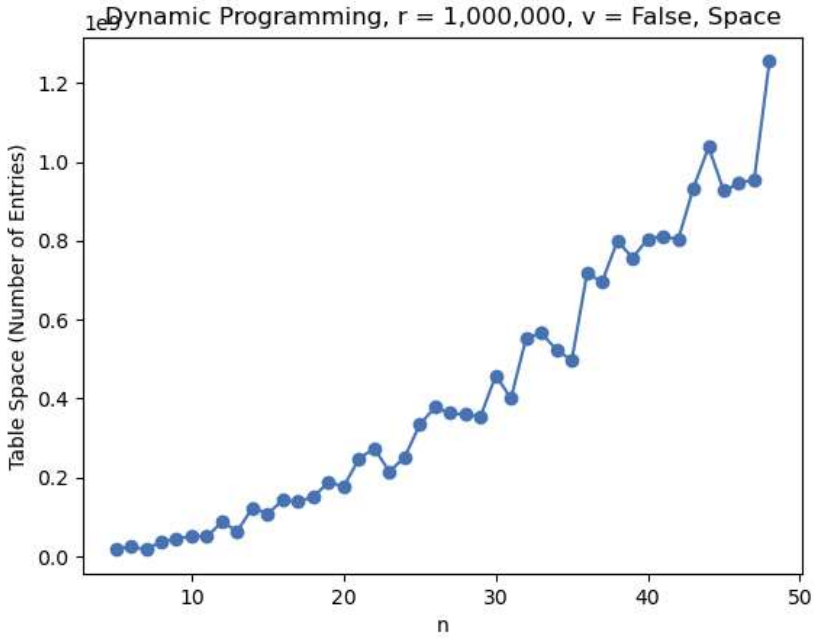


Figure 6: Space Complexity of Dynamic Algorithm with r = 1,000,000 and v = False

Clever Algorithm Analysis

The clever algorithm approach involves computing all combinations for two halves of the input list, and if the sum is not found in one of the halves, then the combinations made by the two halves are summed together to find the sum. (This algorithm essentially uses the Brute Force algorithm on the two halves.) Splitting the combinations into two halves computes $2 * 2^{\frac{n}{2}}$ combinations.

Like the other algorithms, the behavior of the algorithm is almost random when there is a solution in the subset equal to a t value.

Also, to reduce the time complexity, the combinations in both halves that sum up to less than the target value are appended to a dictionary. After this step, the algorithm checks the combinations in the dictionary and returns True if it matches the target; else there is no solution in the respective subset.

The provided figures show how the algorithm still runs in a 2^n time complexity like the Brute Force algorithm. Because of the lower number of combinations, the time complexity is slightly faster than the Brute Force.

```
SSClever(S[1 ... n], t)
    Let L = {1 ... ⌊n/2⌋}
    Let H = {⌊n/2⌋ + 1 ... n}
    Let T be an empty dictionary
    Let l = len(L)
    For counter = 1 to 2^l do:                                  A
        Let sublist = [] an empty array
        Let sum = 0
        For j = 0 to l do:                                      B
            If counter & (1 << j) > 0:
                sublist = sublist || L[j + 1]
                sum = sum + S[L[j + 1]]
        End For
        If sum == t:
            Return True, sublist
```

```
        Elif sum < t:
            T.update({sum:sublist})   //Put sublist in dictionary, with sum as key
End For
Let W = an empty dictionary with key/value pairs
Let h = len(H)
For counter = 1 to 2^h do:                                          C
        Let sublist = [ ] an empty array
        Let sum = 0
        For j = 0 to h do:                                          D
            If counter & (1 << j) > 0:
                sublist = sublist || H[j + 1]
                sum = sum + S[H[j + 1]]
        End For
        If sum == t:
            Return True, sublist
        Elif sum < t:
            W.update({sum:sublist})
End For
For key in T.keys():                                                E
        For sorted_key in sorted(W.keys()):                         F
            If key + sorted_key > t:
                Break
            If key + sorted_key == t:
                Return True, T.get(x) + W.get(n)
        End For
End For
```

Time Complexity:
Contribution from each loop (worst-case):

A+B: Computes the powerset of the lower half of the input list, yielding $n * 2^{\frac{n}{2}}$ time:

$\sum_{i=1}^{2^{\frac{n}{2}}} \sum_{j=1}^{\frac{n}{2}} b = \sum_{i=1}^{2^{\frac{n}{2}}} b * \frac{n}{2} = b * \frac{n}{2} * 2^{\frac{n}{2}} \in \Theta(n * 2^{\frac{n}{2}})$

C+D: Computes the powerset of the upper half of the input list, yielding $n * 2^{\frac{n}{2}}$ time:

$\sum_{i=1}^{2^{\frac{n}{2}}} \sum_{j=1}^{\frac{n}{2}} c = \sum_{i=1}^{2^{\frac{n}{2}}} c * \frac{n}{2} = c * \frac{n}{2} * 2^{\frac{n}{2}} \in \Theta(n * 2^{\frac{n}{2}})$

E+F: Checks if any of the computed subsets can combine to reach the target, takes $2^{\frac{n}{2}} * 2^{\frac{n}{2}}$ time.

$\sum_{i=1}^{2^{\frac{n}{2}}} \sum_{j=1}^{2^{\frac{n}{2}}} d = \sum_{i=1}^{2^{\frac{n}{2}}} d * 2^{\frac{n}{2}} = d * 2^{\frac{n}{2}} * 2^{\frac{n}{2}} = d * 2^{n} \in \Theta(2^{\frac{n}{2}})$

In the worst-case, the runtime of this algorithm is equal to the sum of the three sections discussed above, yielding:

$\frac{n}{2} * 2^{\frac{n}{2}} + \frac{n}{2} * 2^{\frac{n}{2}} + 2^{\frac{n}{2}} * 2^{\frac{n}{2}} = 2^{n} + n * 2^{\frac{n}{2}} \in \Theta(2^{n})$,

We were expecting a runtime of $\Theta(n * 2^{\frac{n}{2}})$, but the analysis shows the runtime is $\Theta(2^{n})$ since $2^{n} > n * 2^{\frac{n}{2}}$ when n approaches ∞.

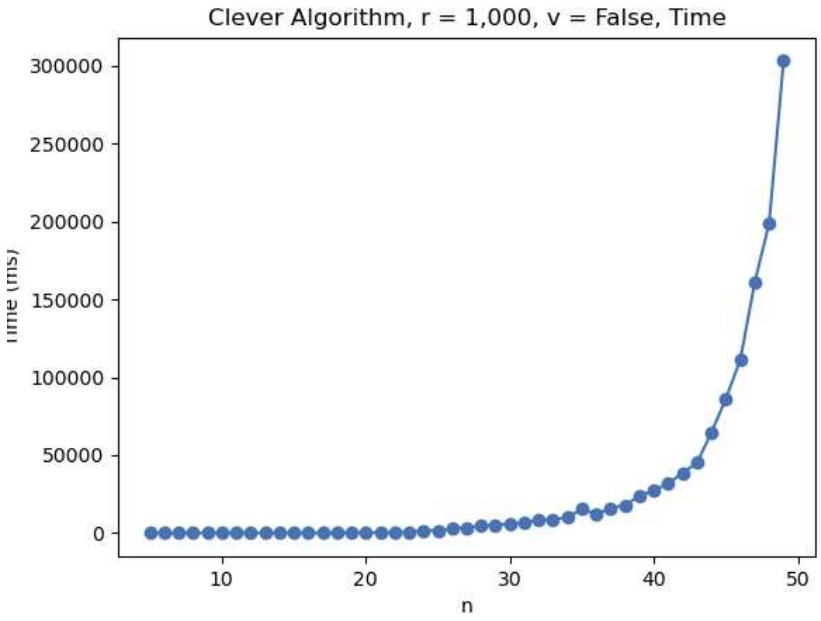


Figure 7: Time Complexity of Clever Algorithm with r = 1,000 and v = False

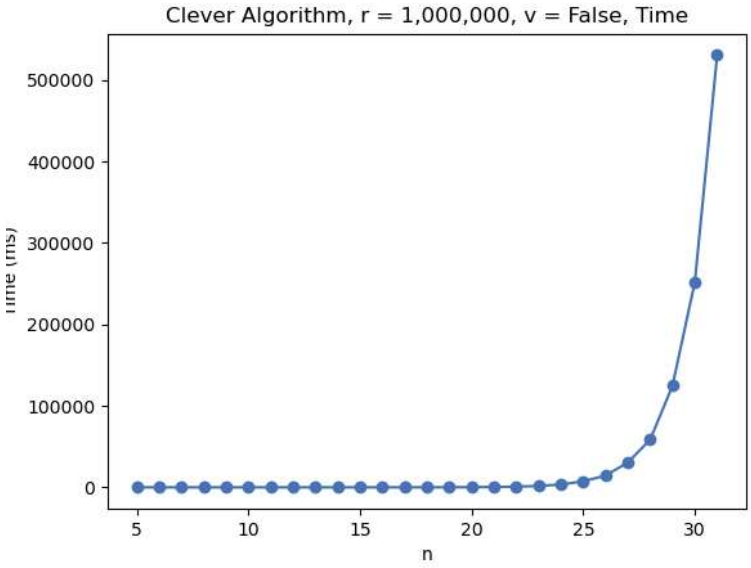


Figure 8: Time Complexity of Clever Algorithm with r = 1,000,000 and v = False

When the range increases the time taken by the clever algorithm increases. We are not sure why this is the case because the algorithm is only dependent on the input array.

Space Complexity:

In the worst-case, this algorithm needs the space used by the input, the length of the input again (*L*+*H*), as well as the space used by the powerset of the lower and upper halves. The powersets are stored in dictionaries, which use key/value pairs, so each dictionary entry is calculated as using two spaces in a table. In the worst case, the amount of space needed by this algorithm will be as follows (assuming that *n* is even):

$$n + \frac{n}{2} + \frac{n}{2} + 2 * 2^{\frac{n}{2}} + 2 * 2^{\frac{n}{2}} = 2n + 2^{\frac{n}{2}+2} \in \Theta(2^{\frac{n}{2}})$$

Note: *n* is chosen as even to simplify the analysis.

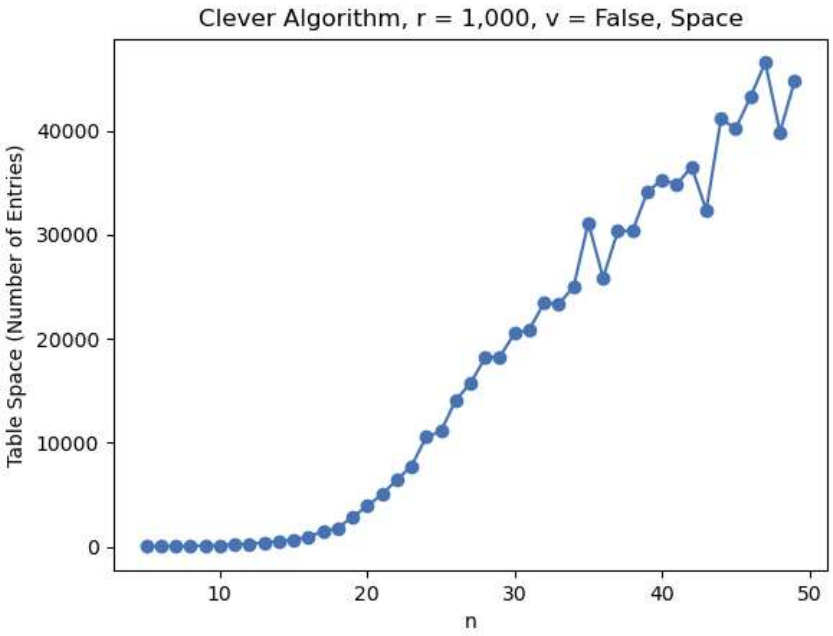


Figure 9: Space Complexity of Clever Algorithm with r = 1,000 and v = False

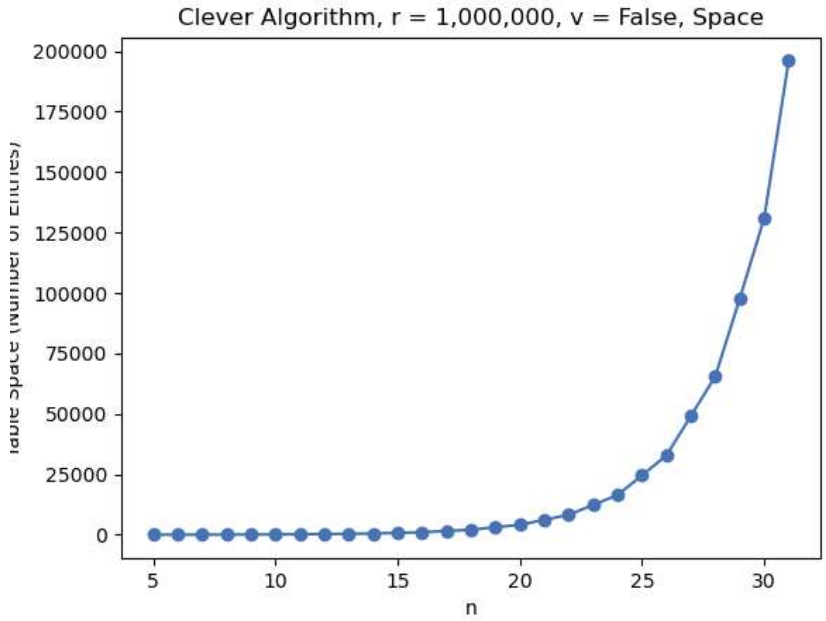


Figure 10: Space Complexity of Clever Algorithm with r = 1,000,000 and v = False

Comparison of Algorithms

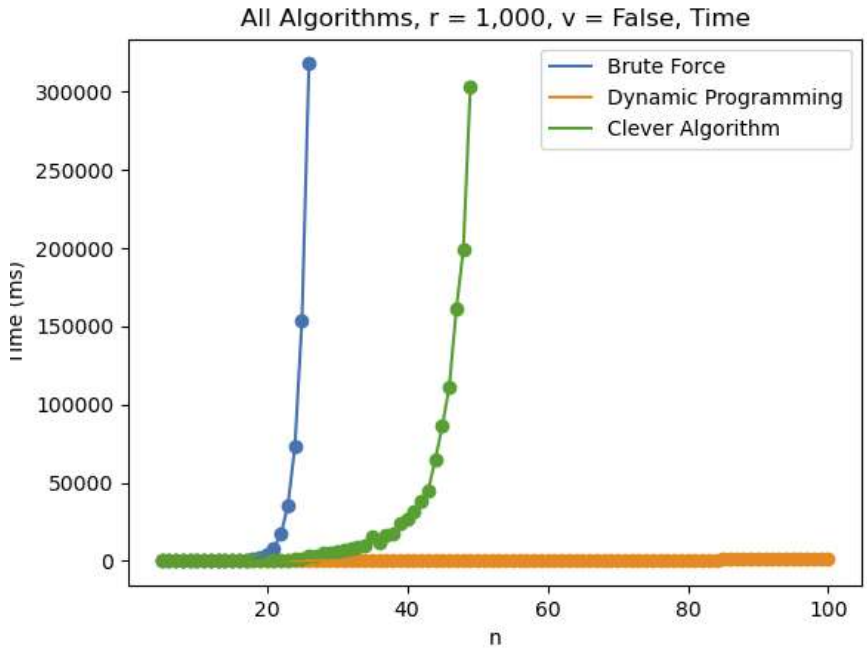


Figure 10: Time Complexity of All Algorithms with r = 1,000 and v = False

Here we can see just how well the Dynamic algorithm performs with low values of *r* when compared to the other two. The Clever algorithm performs better than the Brute Force, as expected.

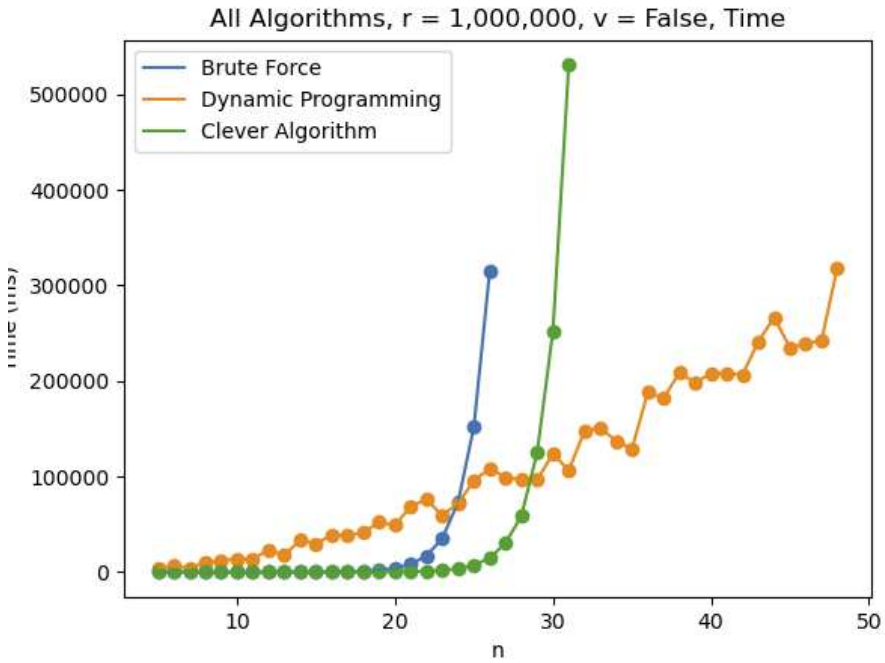


Figure 11: Time Complexity of All Algorithms with r = 1,000,000 and v = False

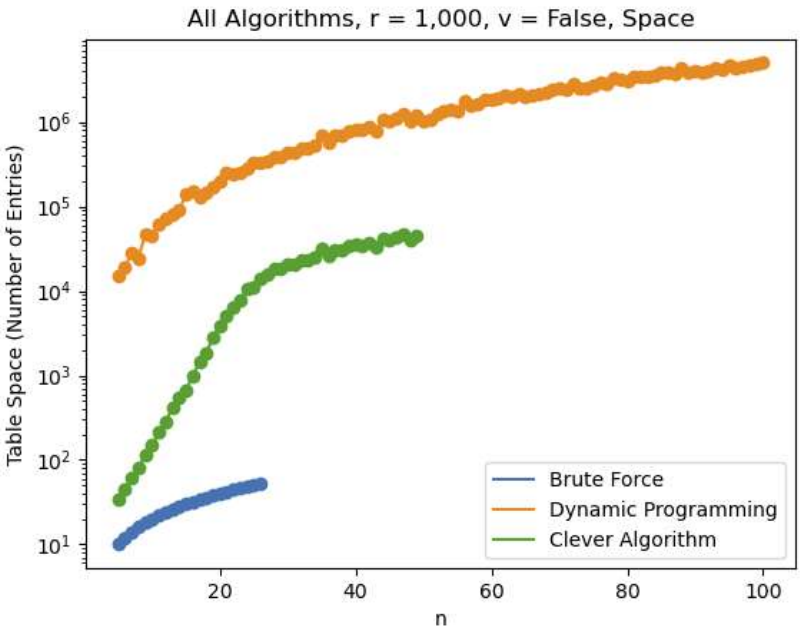


Figure 12: Space Complexity of All Algorithms with r = 1,000 and v = False

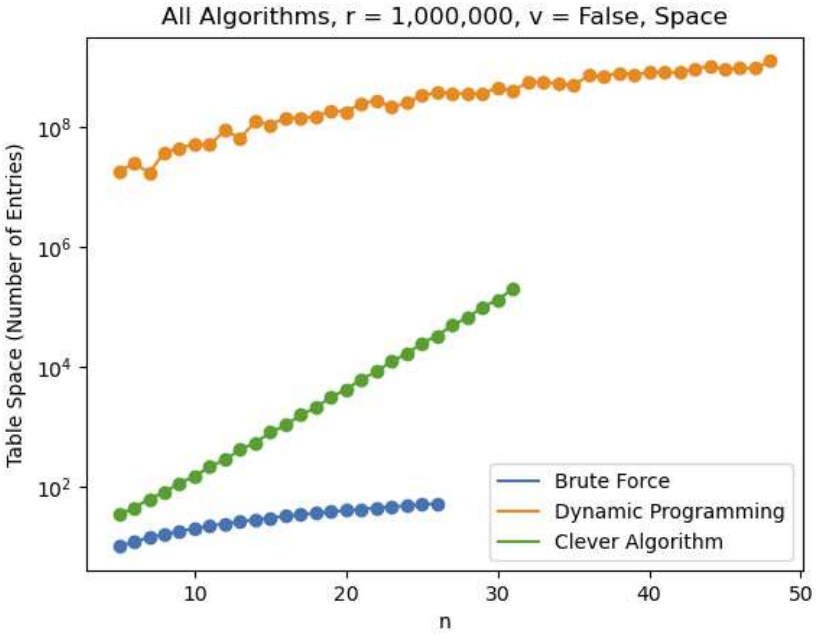


Figure 13: Space Complexity of All Algorithms with r = 1,000,000 and v = False

Division of Labor

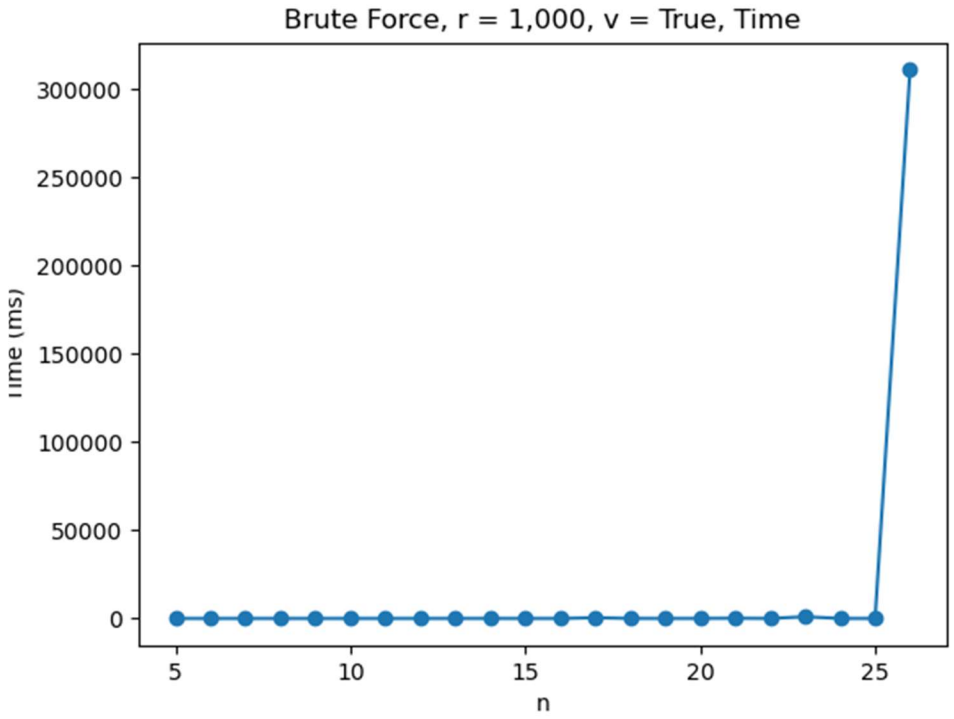
Riley Ruckman – Wrote driver code that automatically plots the data and helped in optimizing the algorithm implementations.

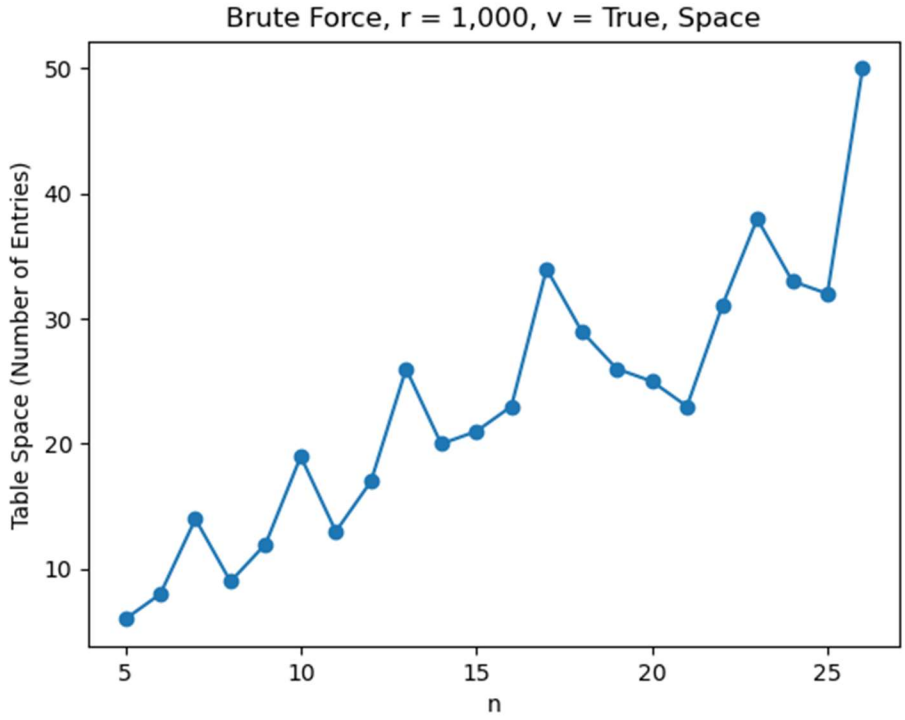
Travis Shields – Wrote code for Brute Force and Dynamic solutions.

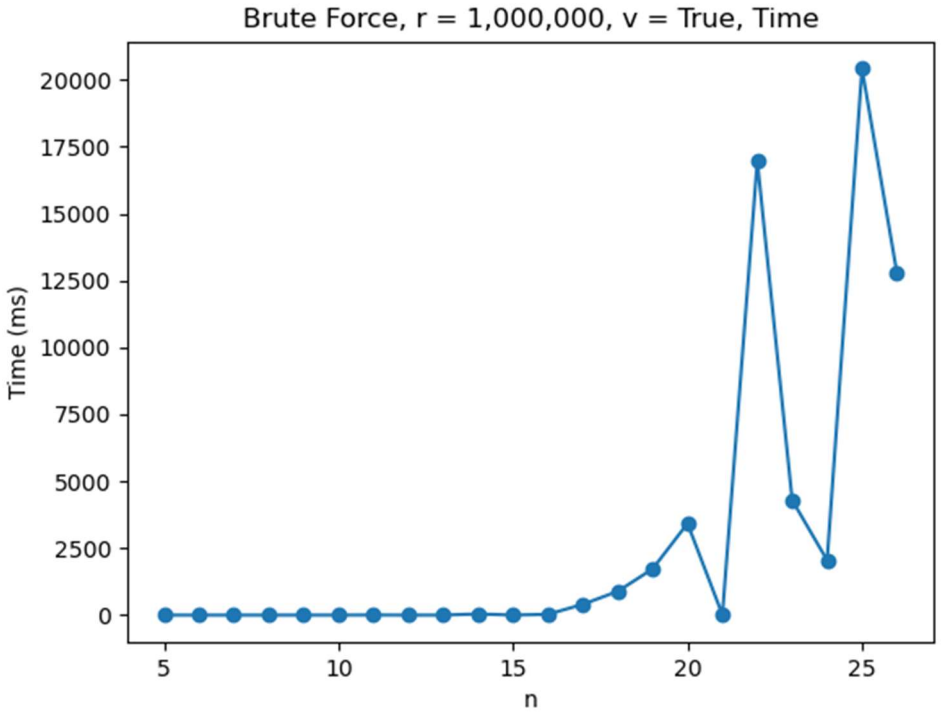
Jot Virdee – Wrote code for Clever Algorithm.

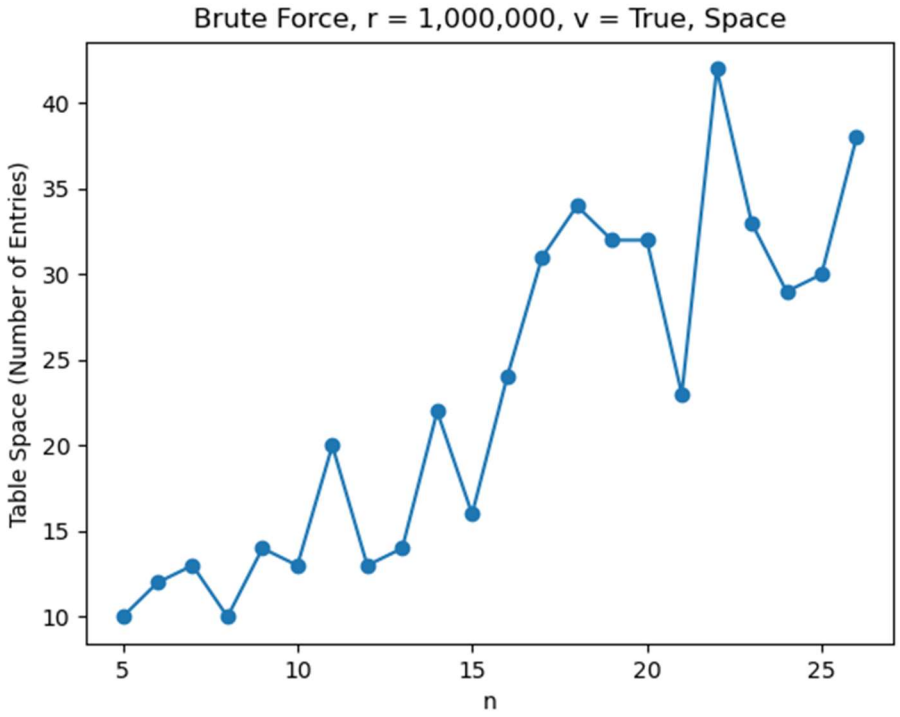
All members worked on the report.

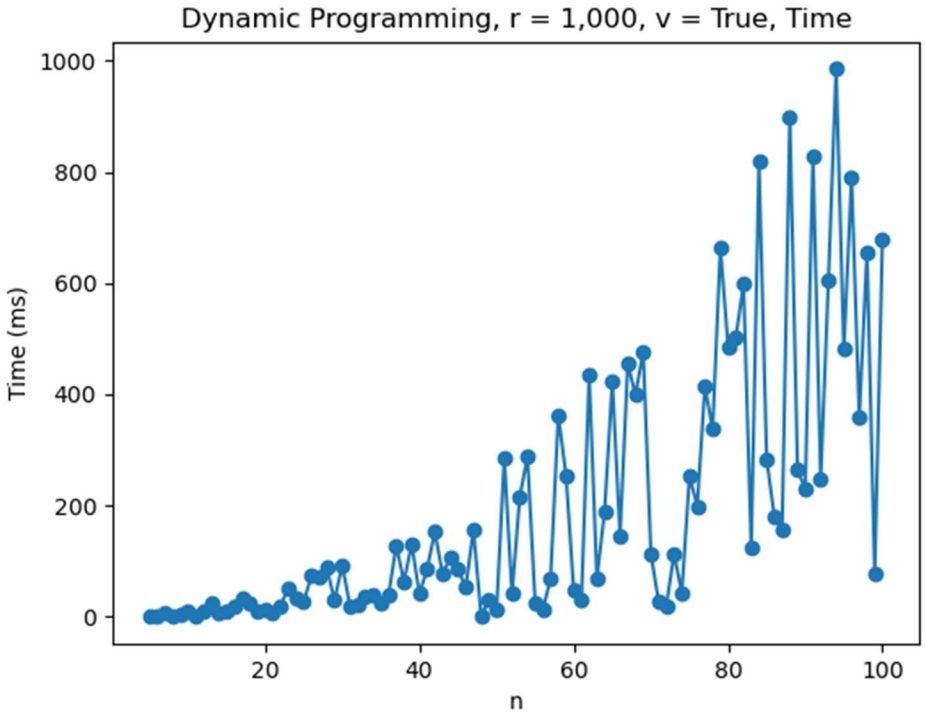
Appendix

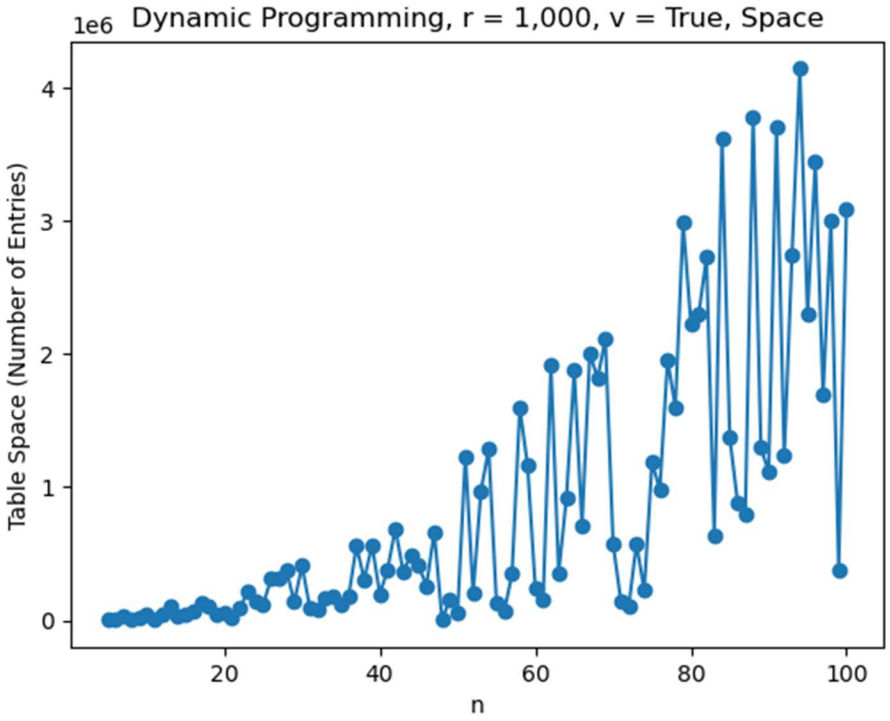

Brute Force, r = 1,000, v = True, Time
Time (ms)
300000
250000
200000
150000
100000
50000
0
5
10
15
20
25
n

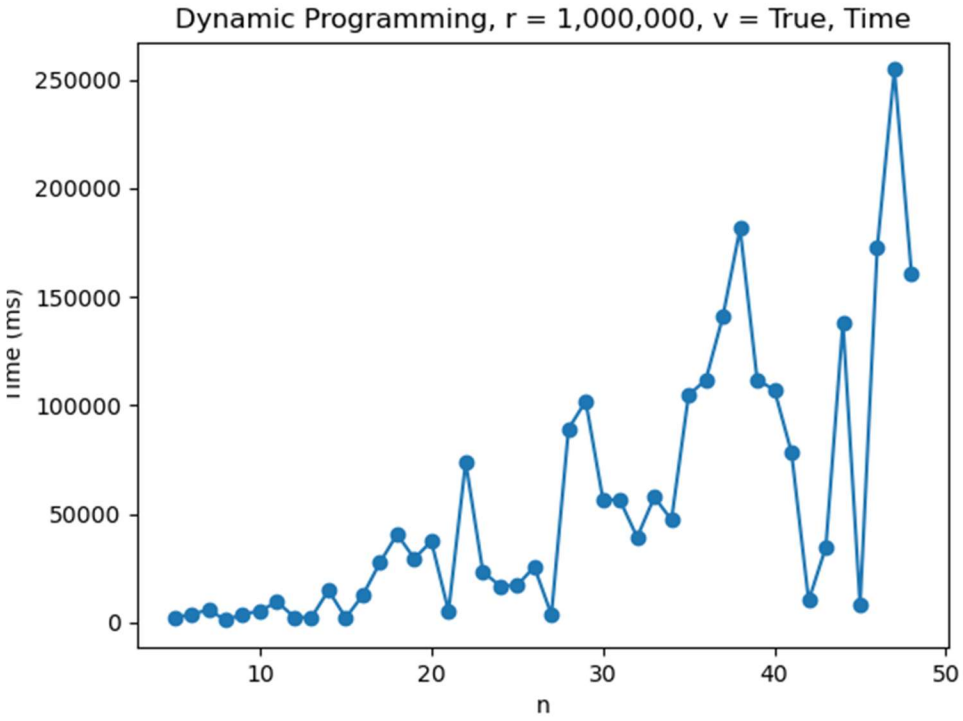

Brute Force, r = 1,000, v = True, Space
Table Space (Number of Entries)
50
40
30
20
10
5
10
15
20
25
n

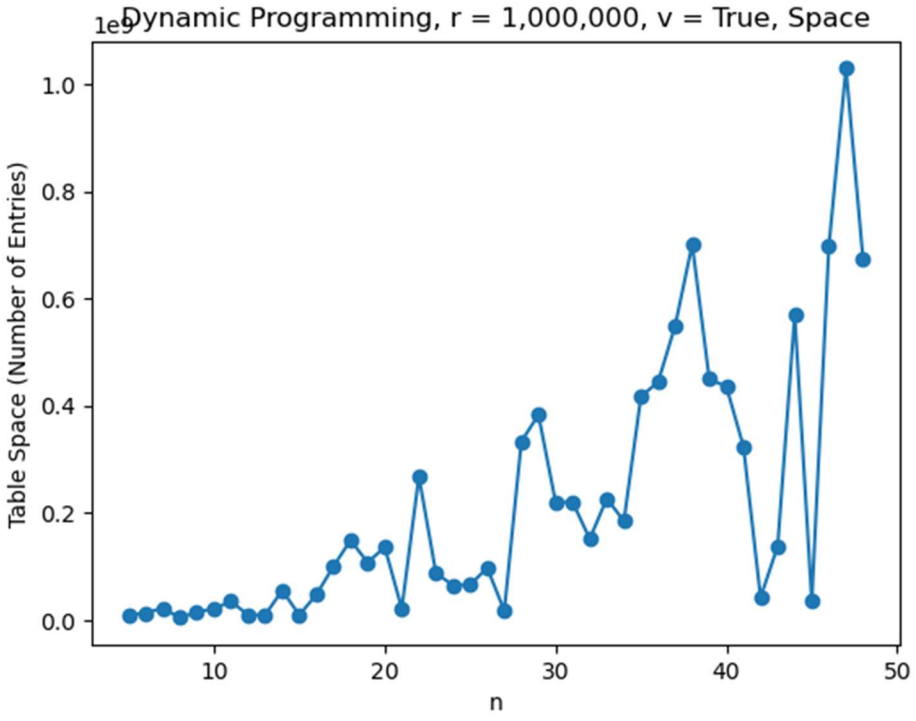

Brute Force, r = 1,000,000, v = True, Time
Time (ms)
20000
17500
15000
12500
10000
7500
5000
2500
0
5
10
15
20
25
n

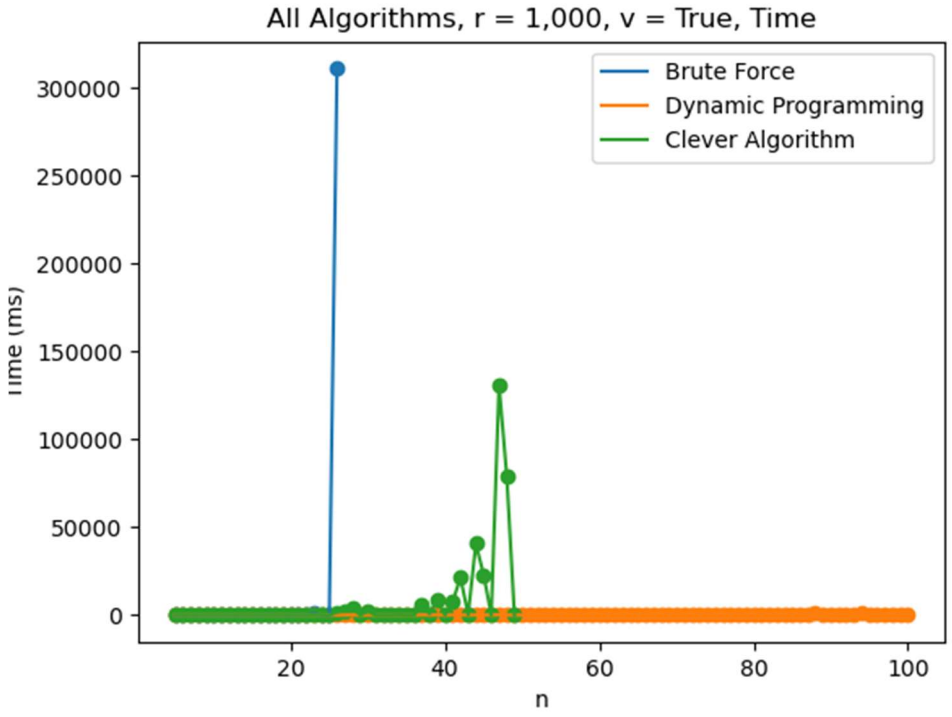

Brute Force, r = 1,000,000, v = True, Space
Table Space (Number of Entries)
40
35
30
25
20
15
10
5
10
15
20
25
n

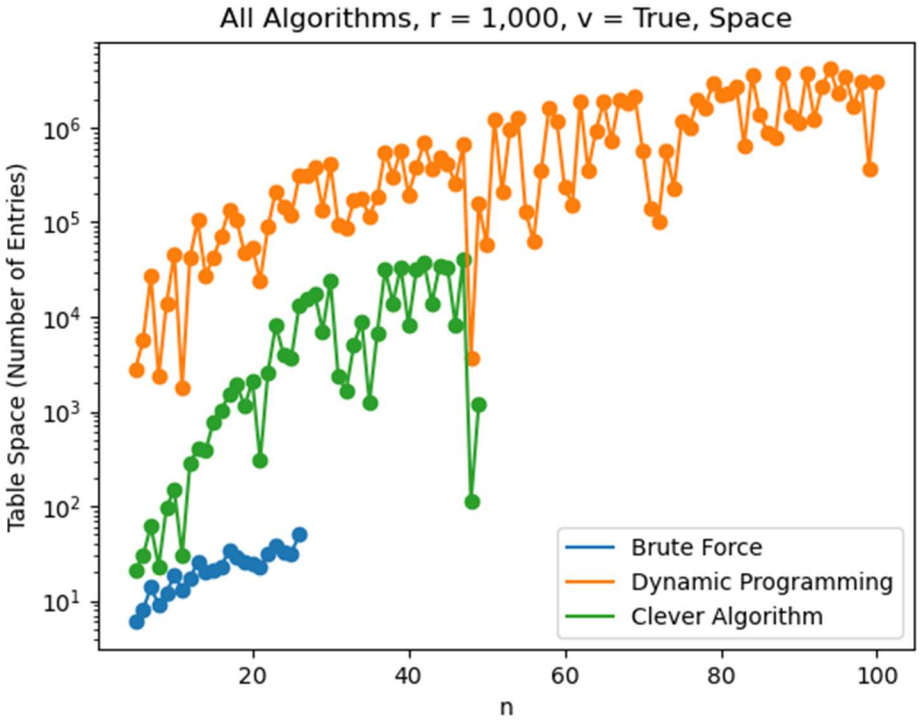

Dynamic Programming, r = 1,000, v = True, Time
Time (ms)
1000
800
600
400
200
0
20
40
60
80
100
n

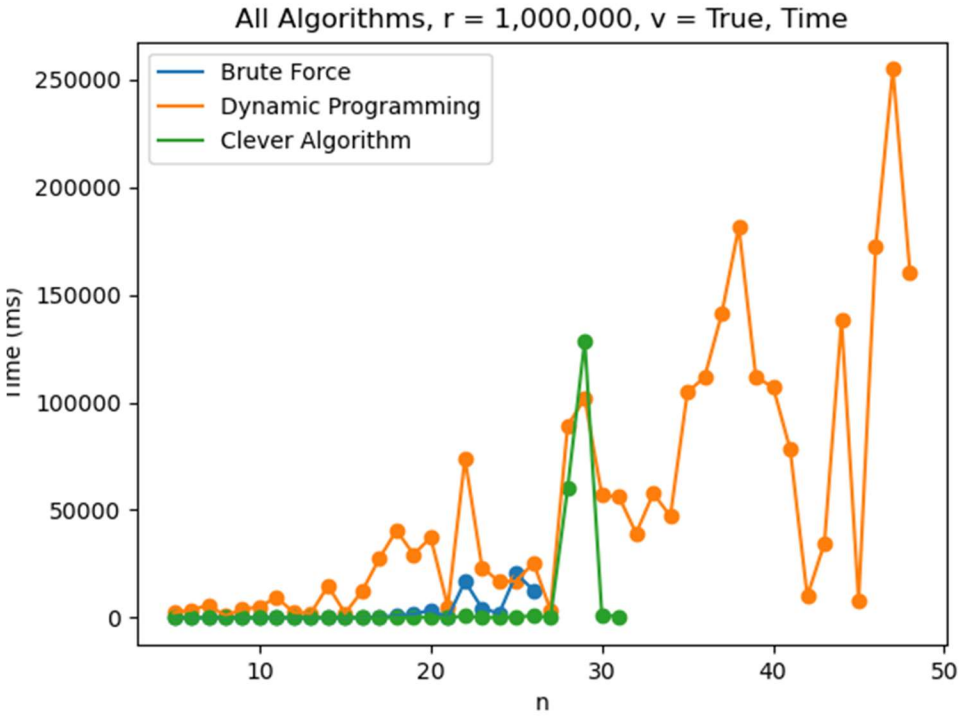

Dynamic Programming, r = 1,000, v = True, Space
1e6
Table Space (Number of Entries)
4
3
2
1
0
20
40
60
80
100
n

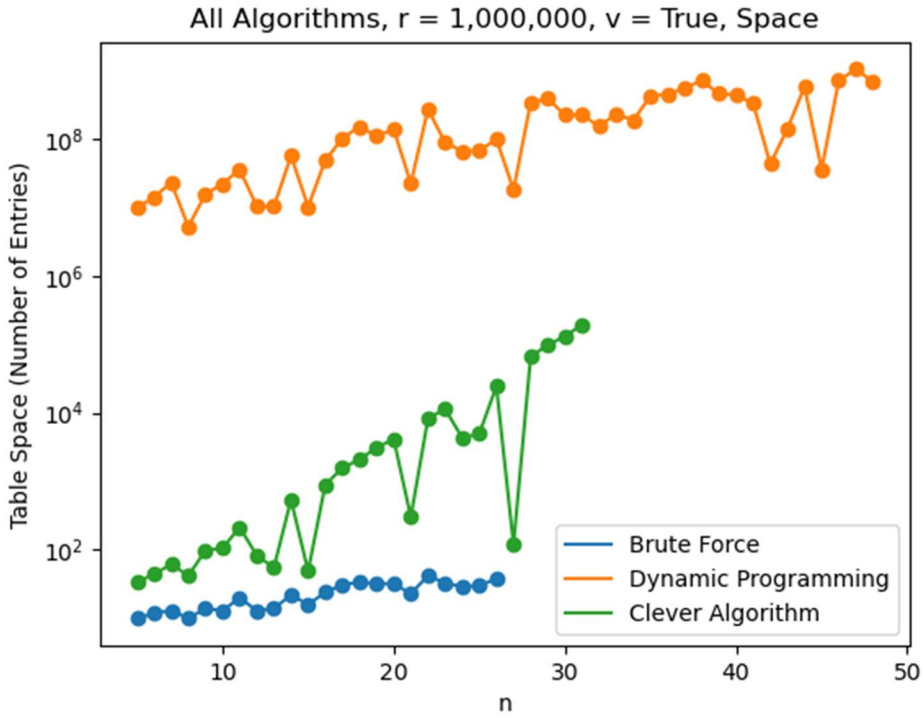

Dynamic Programming, r = 1,000,000, v = True, Time
Time (ms)
250000
200000
150000
100000
50000
0
10
20
30
40
50
n


Dynamic Programming, r = 1,000,000, v = True, Space
1e9
Table Space (Number of Entries)
1.0
0.8
0.6
0.4
0.2
0.0
10
20
30
40
50
n


All Algorithms, r = 1,000, v = True, Time
Brute Force
Dynamic Programming
Clever Algorithm
300000
250000
200000
150000
100000
50000
0
20
40
60
80
100
n


All Algorithms, r = 1,000, v = True, Space
Table Space (Number of Entries)
10^6
10^5
10^4
10^3
10^2
10^1
Brute Force
Dynamic Programming
Clever Algorithm
20
40
60
80
100
n


All Algorithms, r = 1,000,000, v = True, Time
Brute Force
Dynamic Programming
Clever Algorithm
250000
200000
150000
100000
50000
0
Time (ms)
10
20
30
40
50
n


All Algorithms, r = 1,000,000, v = True, Space
10^8
10^6
10^4
10^2
Table Space (Number of Entries)
Brute Force
Dynamic Programming
Clever Algorithm
10
20
30
40
50
n