COP4533 - Programming Assignment
Milestone-2 Report

1 Team Members

- Robin Anstett
- Ari Tramont

2 Algorithm Design and Analysis

2.1 Algorithm 3

2.1.1 Description

The naive exponential-time algorithm uses a brute force method to check every subset of vaults. To begin with, the algorithm creates an array of every potential sum of the given vaults. Then, it validates which vaults meet the appropriate distance constraints. Finally, it selects the maximal value from the valid subsets, and extracts the chosen indices in a 1-indexed format.

Algorithm 3 Brute Force Algorithm $(n, k, v_1, \ldots, v_n)$

```
1: subset_sums ← array of size 2^n initialized to 0
2: for mask ← 0 to 2^n − 1 do
3:     if mask = 0 then
4:         subset_sums[mask] ← 0
5:     else
6:         lsb ← mask & (−mask)
7:         i ← bit position of lsb (least significant bit)
8:         subset_sums[mask] ← subset_sums[mask ⊕ lsb] + v_i
9:     end if
10: end for
11: best_value ← 0
12: best_mask ← 0
13: for mask ← 0 to 2^n − 1 do
14:     valid ← True
15:     shifted ← mask
16:     for j ← 1 to k do
17:         shifted ← shifted ≫ 1
18:         if mask & shifted ≠ 0 then
19:             valid ← False
20:             break
21:         end if
22:     end for
23:     if valid and subset_sums[mask] > best_value then
24:         best_value ← subset_sums[mask]
25:         best_mask ← mask
26:     end if
27: end for
28: chosen ← empty list
29: for i ← 0 to n − 1 do
30:     if (best_mask ≫ i) & 1 = 1 then
31:         append (i + 1) to chosen
32:     end if
33: end for
34: return best_value, chosen
```

2.1.2 Correctness Proof

Theorem. The brute force algorithm always returns the maximal subset of vaults that satisfies the distance constraint.

Proof.

We show that the brute force algorithm is correct because of three facts:

Fact 1: All subsets are considered. The algorithm sums every possible subset of vaults, from an empty subset all the way up to a full subset that includes every vault. There is no potential subset of vaults that the algorithm

does not sum.

Fact 2: Every subset is validated. If any 2 vaults within the subset do not satisfy the distance constraint, the subset is considered invalid. There is no valid subset that is considered invalid, no invalid subset that is considered valid, and no subset that is unchecked.

Fact 3: The maximum value of the valid subsets is selected.

With these three facts in mind, the algorithm exhaustively proves that the selected vaults are the most correct solution. If every subset is checked for maximal value and validity, the algorithm must select the correct solution every time.

2.1.3 Runtime Analysis

The algorithm consists of 4 loops, one of which are nested. Calculating the sum of every subset of vaults takes $\Theta(2^n)$. Validating the maximal sum $\Theta(k * 2^n)$ takes The algorithm also takes $\Theta(n)$ time to extract the correct indices.

Therefore, the overall time complexity of the algorithm is $\Theta(2^n + k * 2^n + n)$, which simplifies to $\Theta(2^n)$. This can be simplified because k is a constant factor that is small relative to n, and 2^n will always grow faster than n.

2.2 Algorithm 4

2.2.1 Description

A solution for Problem G of the vault scenario can be defined recursively and iteratively because of its optimal substructure. Let OPT(i, k) be the optimal value of Problem G with unordered vaults $v_1, ..., v_i$ and a buffer between valid vaults k. Any vault can be chosen for the optimal subset as long as they are at least k vaults apart. Chosen vaults are stored in array S and calculated sums are stored in array C.

For both recursion and iteration, one of the last k vaults will "begin" the summation. Each subsequent selection will be a choice between the current sum and a potential sum, where the larger of the two is chosen. Due to memoization and tabulation respectively, calculated sums are stored and can be recalled if needed. This boils down to the following **recursive formulation:**

$$OPT(i,k) = \begin{cases} 0, & \text{if } i < 0 \\ v_1, & \text{if } i = 0 \\ C[i], & \text{if } C[i] \text{ already exists} \\ max(v_i + OPT(i-k,k), OPT(i-1,k)), & \text{otherwise.} \end{cases}$$

Will return the optimal value, excluding the formation of the optimal subarray (for readability). This shows the choices available at each vault and will form the basis of both dynamic programming solutions, recursively and iteratively.

Algorithm 4 Top-Down Recursion $(n, k, i = n-1, C, S, v_1, \ldots, v_n)$

if $i < 0$ **then**
    return *0, empty list*
end if
if $i = 0$ **then**
    return $v_0, [0]$
end if
if *C[i]* exists **then** ▷ Where C is the cache
    return $C[i], S[i]$ ▷ Where S is the list of chosen vaults
end if
$max(v_i + OPT(i-k, k), OPT(i-1, k))$
if max is $v_i + OPT(i-k, k)$ **then**
    return $C[i]$, $S[i]$
else
    return $C[i-1]$, $S[i-1]$
end if

Algorithm 4 Iterative Bottom-Up $(n, k, C, S, v_1, \ldots, v_n)$

$S[n-1] = v_{n-1}$
for $i = n-2$ to -1 **do**
    if $n > i > n-k$ **then**
        $max(v_i, C[i+1])$
        if $max = v_i$ **then**
            $S[i] = [i]$
        else
            $S[i] = [i+1]$
        end if
    else
        $max(v_1 + C[i+k], C[i+1])$
        if max is $C[i+1]$ **then**
            $S[i] = i+1$
        else
            $S[i] = [i] + S[i+k]$
        end if
    end if
end for
return $C[0], S[0]$

2.2.2 Correctness Proof

Theorem. Both dynamic programming algorithms compute the optimal solution for Problem G.
Note: Although both algorithms are implemented differently, they can be reduced to the same recursive formulation. Thus, they will share a proof.

Proof.
We show that Problem G satisfies the optimal substructure property, and as such, will be correctly expressed using the recursive formulation discussed.

Let O_n be an optimal subset of vaults **k** apart for a list of **n** length.

Case 1: Vault n is not included in O_n. Then O_n is an optimal subset of the first $n-1$ to $n-k$ vaults with the same **k**. If a different subset within the same range had a higher value, then it is more optimal than O_n, which should not be possible.

Case 2: Vault n is included in O_n. Then the subset created by removing vault n from O_n should still be optimal; otherwise, there would be another subset that has a higher sum. If this subset is compatible with n, it would have a greater sum than O_n, which should not be possible. If n isn't compatible, then there would be another subset that has a greater sum, meaning O_n is not optimal.

Using these two cases, the optimal solution for the overall problem must use one of the two subproblems:

Including n, continuing the current sum **or** *excluding* n, starting a new sum.

As these are two simple subproblems that are derived from the overall problem presented in Problem G, it stands that Problem G has the optimal substructure property. This means that it can be solved using the recursive formulation.

2.2.3 Runtime Analysis

There are two different algorithms to analyze.

The **iterative** algorithm consists of a for loop that iterates through the given list of vaults of size **n**. Within each loop, one of two *max()* is selected and values from the cache and the vault list are compared (and assigned). Gathering a value from an array by index and adding values to an array are both constant-time operations, $\Theta(1)$, and can be ignored. However, the *max()* function takes $\Theta(n)$ time. It is important to note that the same problem is never solved twice due to tabulation, so the time complexity is not bloated with repeated solves. Sums are "pre-calculated" in the sense that, once calculated, they are stored to be recalled later. The time complexity is then $\Theta(n * n)$, which simplifies to $\Theta(n^2)$.

The **recursive** algorithm consists of a call to itself twice followed by a *max()* call. This function is called over every element in the vault list. Normally, a recursive function of this nature would have an incredibly large runtime; but, due to memoization, the recursive call on a repeated problem returns before any calculations can run. The result of each problem is stored in an array that can be referenced if needed. Therefore, no problem will run more than once, trimming redundancy. As stated previously, referring and adding to an array is constant time, so it does not impact the overall time complexity. The recursive function will be called **n** times, and a function of $\Theta(n)$ time is used in each "completed" run. This results in a time complexity of $\Theta(n*n)$, which simplifies to $\Theta(n^2)$.

2.3 Algorithm 5

2.3.1 Description

The dynamic programming solution for the vault selection problem involves creating a choice array to allow for backtracking through to extract the correctly chosen vaults.

$$OPT(i,k) = \begin{cases} 0, & \text{if } i < 0 \\ v_1, & \text{if } i = 0 \\ C[i], & \text{if } C[i] \text{ already exists} \\ max(v_i + OPT(i-(k+1),k), OPT(i-1,k)), & \text{otherwise.} \end{cases}$$

Algorithm 5 Dynamic Programming Algorithm $(n, k, v_1, \ldots, v_n)$

```
1: if n ≤ 0 then
2:     return 0, empty list
3: end if
4: dp[0...n] ← array of zeros
5: choose[0...n] ← array of False
6: for i ← 1 to n do
7:     include ← v_i
8:     prev ← i − k − 1
9:     if prev ≥ 0 then
10:        include ← include + dp[prev]
11:    end if
12:    if include > dp[i − 1] then
13:        dp[i] ← include
14:        choose[i] ← True
15:    else
16:        dp[i] ← dp[i − 1]
17:        choose[i] ← False
18:    end if
19: end for
20: indices ← empty list
21: i ← n
22: while i ≥ 1 do
23:    if choose[i] then
24:        append i to indices
25:        i ← i − (k + 1)
26:    else
27:        i ← i − 1
28:    end if
29: end while
30: reverse(indices)
31: return dp[n], indices
```

2.3.2 Correctness Proof

Theorem. The dynamic programming algorithm (creating a choice array) always returns the maximal subset of vaults that satisfies the distance constraint.
Proof.
We show that the dynamic algorithm is correct:

Base Case: In the case of there being 0 vaults to choose from, the algorithm returns the maximal value of 0 and an empty set of vaults.

Subproblem: The algorithm creates a set of overlapping subproblems. Each step of the algorithm considers the maximum value of a valid subset of vaults in the set of vaults seen thus far. For each vault i seen there is one of two cases:

Case 1: We do not include vault i. Thus, the vault must not be part of the optimal solution. So, we prefer OPT(i-1).

Case 2: We do include vault i. Thus, all vaults within k must not be a part of the optimal solution. So, we prefer $v_i + OPT(i(k+1))$

These cases show that $OPT(i) = max(v_i + OPT(i(k+1)), OPT(i-1))$. For every subset of i vaults. Thus, the overlapping problems are solved to create an optimal solution for the total set of vaults.

2.3.3 Runtime Analysis

The algorithm consists of 2 loops, none of which are nested. Creating the choice array takes $\Theta(n)$. Reconstructing the solution from the choice array takes $\Theta(n)$.

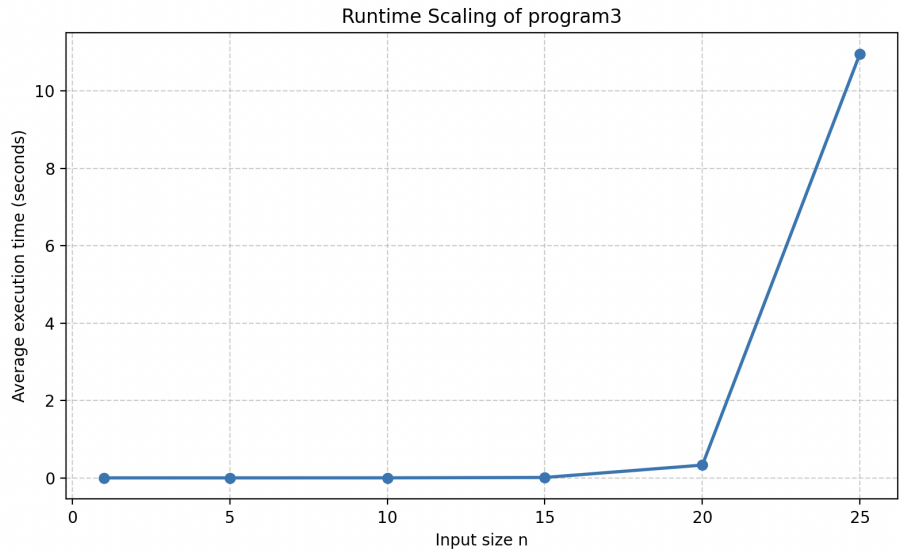
Therefore, the overall time complexity of the algorithm is $\Theta(n+n)$, which simplifies to $\Theta(n)$.

3 Experimental Comparative Study

3.1 Experimental Setup

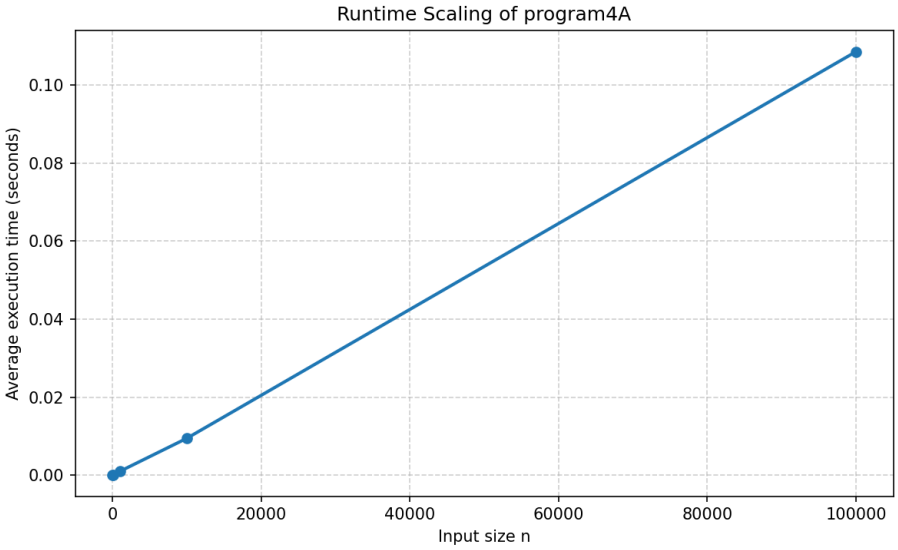
This experiment had to be designed to accommodate the general vault problem. Thus, unlike Milestone 1, we generated truly random values for each of the vaults. For ease, the vaults were randomly assigned a value from 1 to 100. This allowed us to generate multiple general versions of the vault problem. The input sizes of n were selected carefully based on each algorithm. (More information present under each algorithm). A maximum input size of 100, 000 was selected for n, as the team felt this gave us a wide range of values to show the immense differences in average runtime. To ensure more accuracy, we also used repetitions to ensure an average runtime of all input sizes. Number of repetitions varied from program to program due to timing constraints, but at least 100 repetitions were used for every plot.

3.2 Plot 3



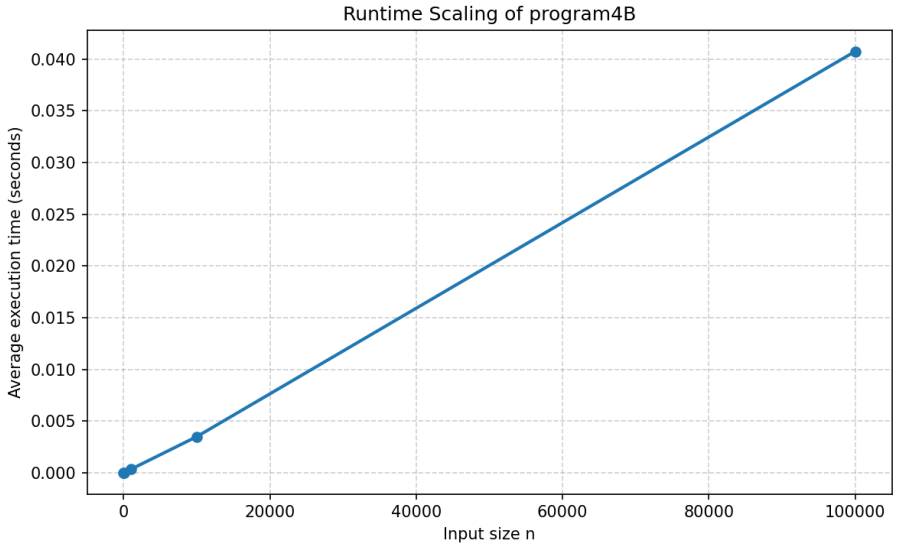
The graph shows the average runtime of Program 3 - Brute Force Algorithm. The input range tested is extremely small, as the algorithm takes an extremely long time to run on larger input sizes. Even with the small range, however, you can see the exponential increase in average runtime as the program grows from an input of 20 to 25. As shown by the plot, this algorithm is not effective for large input sizes of n. $\Theta(2^n)$

3.3 Plot 4



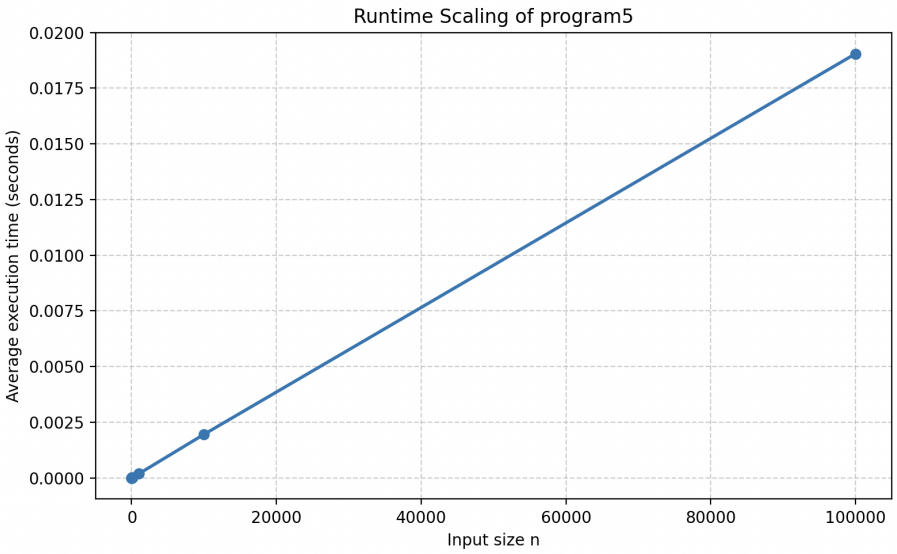
The graph shows the average runtime of Program 4A - Top-Down Recursive with varying input sizes. Python has a recursive depth limit of 1,000, but that limit was changed to 100,000 to have a better representation of the runtime while remaining feasible. The line is seemingly straight, implying a linear growth; however, there is a notable dip around 10,000. The numbers are disproportionately represented, making 10,000 to 100,000 look linear while the range from 10 to 10,000 is minuscule. This suggests that the plot's display is skewed and a more proportional x-axis would display an exponential growth. $\Theta(n^2)$

3.4 Plot 5



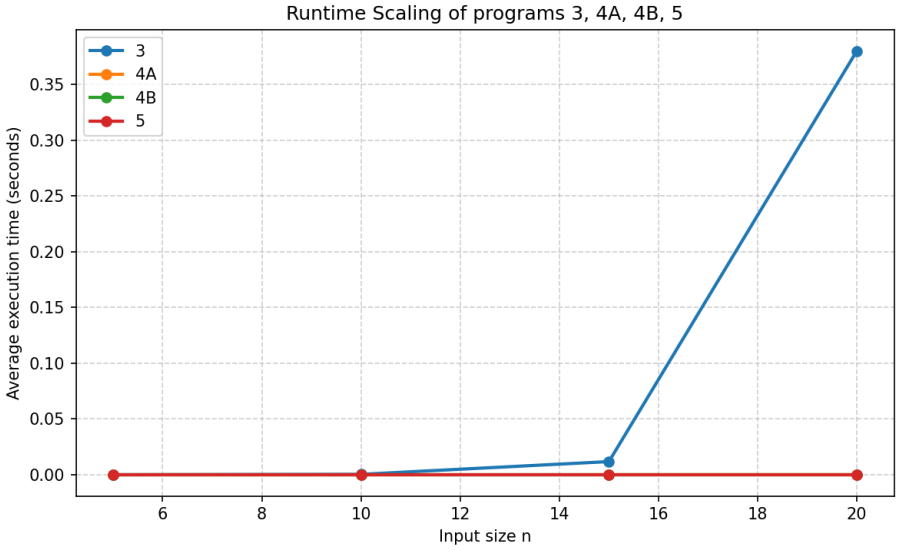
The graph shows the average runtime of Program 4B - Iterative Bottom-Up with varying input sizes. No changes to the parameters were needed, unlike 3 and 4A. The input maximum remained 100,000 for consistency. Like 4A, the line is deceptively linear. This follows the same pattern of a small dip in the 10 to 10,000 range while 10,000 to 100,000 is over-represented. Following the same reasoning as 4A, it is very likely that 4B also runs in exponential time. However, it is considerably faster than 4A. $\Theta(n^2)$

3.5 Plot 6



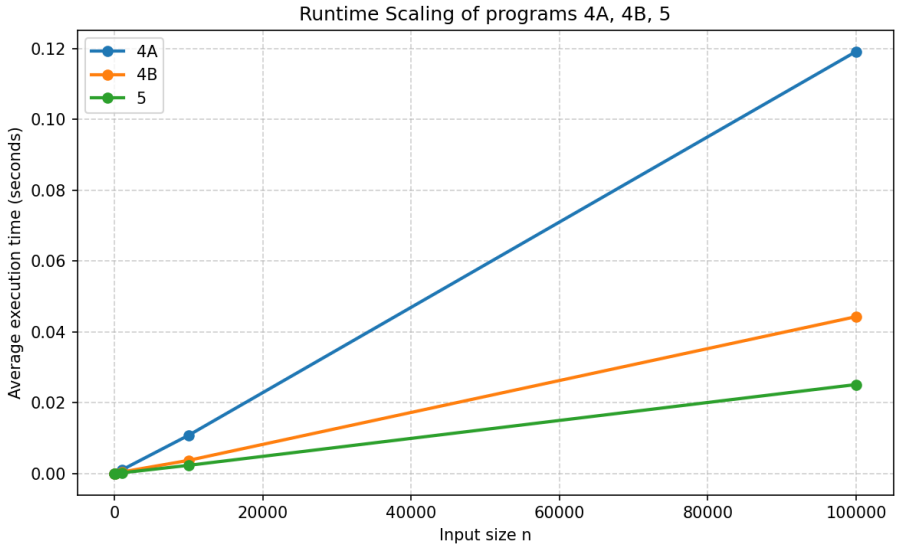
This graph shows the average runtime of Program 5 - Dynamic Programming (Choice Array) Algorithm. No changes to the parameters were needed, unlike 3 and 4A. The input maximum remained 100,000 for consistency. The line of the plot is linear and well represents the algorithm (which is runs in linear $\Theta(n)$ time).

3.6 Plot 7



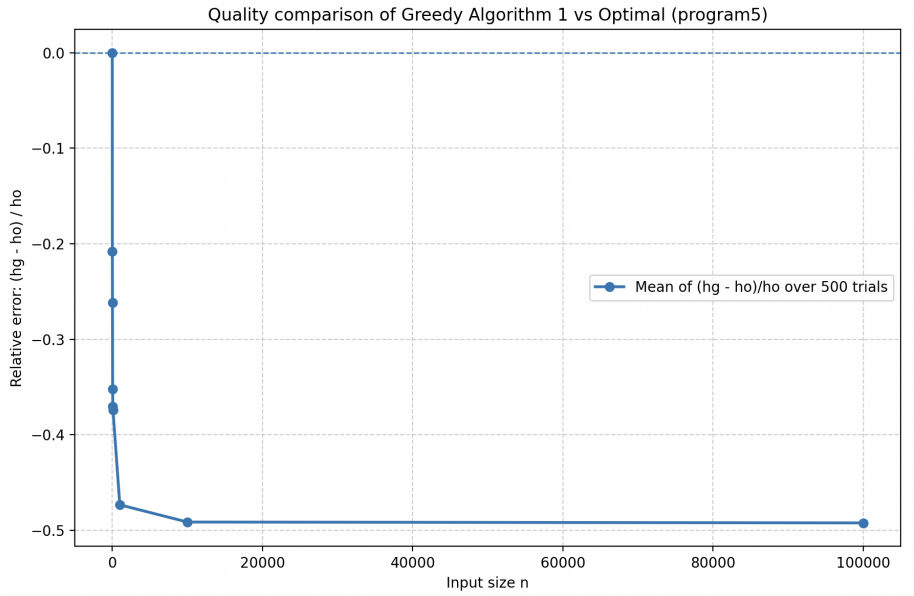
This graph shows the average runtime of all the programs made for Milestone 2. The parameters were changed to have inputs from 5 to 25, as Program 3 took too long for any greater input. The most obvious result is the large gap between Program 3 and the rest. Program 3 grew considerably quicker than Programs 3, 4A, 4B, and 5. The difference was large enough to collapse the 3 programs into seemingly one line. This supports the idea that Program 3 has a complexity of $\Theta(2^n)$, while the others are significantly faster. As the programs other than Program 3 are all within the same line, it is impossible to compare the three to each other using this plot, nor say any specific complexity. The only information is that, for inputs up to 20, these three programs solve Problem G nearly instantly.

3.7 Plot 8



This graph shows the average runtime of Programs 4A, 4B, and 5. The parameters match those used in Plot 4, with a recursive depth limit changed to support an input range from 10 to 100,000. In order from slowest to fastest, the programs rank 4A, 4B, and 5. Programs 4B and 5 have similar runtimes with about a difference of 0.02 seconds. Program 4A is significantly slower than both (while still remaining incredibly faster than Program 3). The difference between 4A and 4B does make sense. Recursion, even with memoization, is slower than iteration with tabulation.

3.8 Plot 9



This graph shows the average quality comparison of Algorithm 1 and Algorithm 5 over a wide range of n values. As you can see from the graph, the quality of Algorithm 1 decreases with higher values of n. This makes sense, as Algorithm 1 is designed for Problem Special Case 1, in which values are sorted least to greatest. Thus, the algorithm simply picks the last values and continues selecting vaults until there are no more left available. However, when the vaults are not sorted, the algorithm is not maximizing the treasure value of the vaults selected. This causes less errors on smaller n values where the optimal solution is likely not far off from the subset chosen based on position. However, with increasingly large values of n, the margin for error becomes much greater. Thus, we can see with large input sizes the quality comparison of Algorithm 1 and 5 decreases dramatically.

3.9 Observations/Comments

It was difficult to test multiple programs against each other, as the maximum input size each algorithm could handle varied widely. Thus, this makes it hard to compare them using a graph, when we could only test them on a small range of inputs. However, it was interesting to learn how functions with a bad time complexity are limited to inputs of only 25-30 at best.

4 Conclusion

Milestone 2 was more complex than Milestone 1. Conceptualizing and implementing recursion is always a challenge, so completing Program 4A was time consuming. Analysis was relatively straightforward. Another snag in this milestone was creating Plot 7. As Program 3 was significantly less efficient than the other programs, there is no proper "test" that will demonstrate Program 3 and the other programs perfectly. Trends for Programs 4A, 4B, and 5 are most evident with large inputs (100,000); Program 3 struggled to process 20. This resulted in the graph shown, where there is no detailed information shown for Programs 4A, 4B, and 5. It was also interesting to see that Python has an innate recursion depth limit (1000), presumably to maintain efficiency and prevent memory errors. Thankfully, Program 4A was efficient enough that bypassing this limit did not affect the system negatively whatsoever.