COP4533 - Programming Assignment
Milestone-1 Report

1 Team Members

- Robin Anstett
- Ari Tramont

2 Algorithm Design and Analysis

2.1 Algorithm 1 (ProblemS1)

2.1.1 Description

We can solve the vault selection S1 problem (vaults are ordered from least to greatest treasure value) using the following greedy strategy.

Greedy strategy: Always choose the last available vault among those compatible with the already chosen vaults.

Step by step example: Suppose we have n= 7 and k = 2, with the vaults:

$$v1 = 1, v2 = 5, v3 = 9, v4 = 14, v5 = 17, v6 = 19, v7 = 21.$$

1. Select v7, the last available vault.
2. The next compatible vault is v4.
3. Finally, choose v1.

The resulting set is $\{v1 = 1, v4 = 14, v7 = 21\}$ with a value of 36.

Algorithm 1 Vault Selection - ProblemS1 (Last Vault Value First)

```
1: procedure VaultSelection(v1, ..., vn)
2:     S ← ∅                                   ▷ set of vaults selected
3:     for v = n to 0 do
4:         if vault v is compatible with S then
5:             S ← S ∪ {j}
6:         end if
7:     end for
8:     return S
9: end procedure
```

2.1.2 Correctness Proof

Theorem.

The last-vault-value-first greedy algorithm produces an optimal solution to the vault selection problem for special case 1.

Proof. [by contradiction]

Assume that the greedy algorithm is not optimal.

1. Let $g_1, g_2, \ldots, g_k$ denote the set of vaults selected by the greedy algorithm.
2. Let $o_1, o_2, \ldots, o_m$ denote the set of vaults in an optimal solution with maximal r such that $o_i = g_i$ for all $i \leq r$.
3. Consider the vault at position $r + 1$. By construction, the greedy choice g_{r+1} is the greatest treasure value vault available (due to the special setup of ProblemS1) compatible with $g_1, g_2, \ldots, g_r$ and $o_1, o_2, \ldots, o_r$. Hence $treasure\_value(o_{r+1}) \leq treasure\_value(g_{r+1})$.
4. Replace o_{r+1} with g_{r+1} to form a new solution. This replacement preserves feasibility because g_{r+1}'s treasure value is no less than o_{r+1} and is compatible with $o_1, \ldots, o_r$.
5. The new solution has size at least as large as the optimal solution O, but it agrees with the greedy solution on $r + 1$ vaults, contradicting the maximality of r.

Hence, the greedy algorithm is optimal.

2.1.3 Runtime Analysis

Let n be the number of vaults. The algorithm scans through the list once, adding each compatible vault to the solution. The algorithm then ends after all the vaults have been visited. This takes $O(n)$ time.

Therefore, the greedy algorithm running time is O(n).

2.2 Question 1 - ProblemG

Step by step example: Suppose we have n= 6 and k = 2, with the vaults:

$$v1 = 5, v2 = 8, v3 = 1, v4 = 3, v5 = 9, v6 = 2.$$

1. Select v6, the last available vault.
2. The next compatible vault is v3.
3. There are no more available vaults that are compatible.

The resulting set is $\{v3 = 1, v6 = 2\}$ with a value of 3. The maximum value is 17 by selecting the set $\{v2 = 8, v5 = 9\}$. Thus, the greedy algorithm that solves special case 1 cannot be applied generally.

2.3 Question 2 - ProblemS2

Step by step example: Suppose we have n= 8 and k = 1, with the vaults:

$$v1 = 12, v2 = 8, v3 = 3, v4 = 4, v5 = 5, v6 = 8, v7 = 13, v8 = 14.$$

1. Select v8, the last available vault.
2. The next compatible vault is v6.
3. The next compatible vault is v4.
4. Finally, choose v2.

The resulting set is $\{v2 = 8, v4 = 4, v6 = 8, v8 = 14\}$ with a value of 34. The maximum value is 38 by selecting the set $\{v1 = 12, v4 = 4, v6 = 8, v8 = 14\}$. Thus, the greedy algorithm that solves special case 1 cannot be applied to special case 2.

2.4 Algorithm 2 (ProblemS2)

2.4.1 Description

We can solve the problem where the vaults have a local minimum with the following greedy algorithm.

Greedy strategy: Always choose the rightmost available vault until the minimum is found, then always choose the leftmost available vault.

Step by step example: Suppose we have n = 8 and k = 1, with the vaults:

$$v_1 = 12, v_2 = 8, v_3 = 3, v_4 = 4, v_5 = 5, v_6 = 8, v_7 = 13, v_8 = 14.$$

1. Select v_8, the last available vault.

2. The next compatible vault is v_6.

3. The next compatible vault is v_4.

4. Notice that v_3 is the "turning point," or the minimum value.

5. Select v_1, the first available vault.

6. There are no more available vaults that are compatible.

The resulting set is $\{v_1 = 12, v_4 = 4, v_6 = 8, v_8 = 14\}$ with a value of 38.

Algorithm 2 Vault Selection - Problem S2 (Rightmost to Leftmost)

```
    procedure VaultSelection(v_1, ..., v_n)
2:      S ← ∅                                   ▷ set of vaults selected
        for j = n to 0 do
4:          if vault j < (j − 1) then
                while vault j > (j + 1) do      ▷ search for the minimum
6:                  j = j + 1
                end while
8:              break                           ▷ minimum found
            end if
10:         if vault j is compatible with S then
                S ← S ∪ {j}
12:         end if
        end for
14:     i ← 0
        while i < j do
16:         if vault j is compatible with S then
                S ← S ∪ {i}
18:         end if
        end while
20:     return S
    end procedure
```

2.4.2 Correctness Proof

Theorem. The rightmost-to-leftmost greedy algorithm produces an optimal solution to the vault selection problem for special case 2.

Proof. [by contradiction]

Assume that the rightmost-to-leftmost algorithm is not optimal.

1. Let $v_1, v_2, ..., v_k$ denote the set of vaults chosen by the greedy algorithm.

2. Let $o_1, o_2, ..., o_m$) denote the set of vaults chosen in an optimal solution with maximal r such that $o_i = v_i$ for all $i \leq r$.

3. Consider the vault at position $r+1$ in the sets. The greedy choice v_{r+1}, based on the algorithm's design, is the vault with the greatest value compatible with both $v_1, ..., v_r$ and $o_1, ...o_r$. The value of $v_{r+1} \leq o_{r+1}$.

4. Swap the vaults at $r+1$ between the two sets (o_{r+1} and v_{r+1}). This should be possible, as $v_1, ..v_r$ and $o_1, ..., o_r$ are compatible and g_{r+1} is no less than o_{r+1}.

5. The size of the solution created by the swap is no smaller than the size of the optimal solution O, but it agrees with the greedy solution on $r+1$ vaults, contradicting the maximality of r.

Hence, the greedy algorithm is optimal.

2.4.3 Runtime Analysis

Let n be the number of vaults. Going over the list from the left takes O(n) time, and searching for the minimum during this go-over could take O(n) time. These combine to O(n^2) time. Going over this list from the right takes O(n) time.

Therefore, the overall running time is

$$O(n^2) + O(n) = O(n^2)$$

The search for the minimum during the first go-over dominates, so the greedy rightmost-to-leftmost algorithm runs in O(n^2) time.

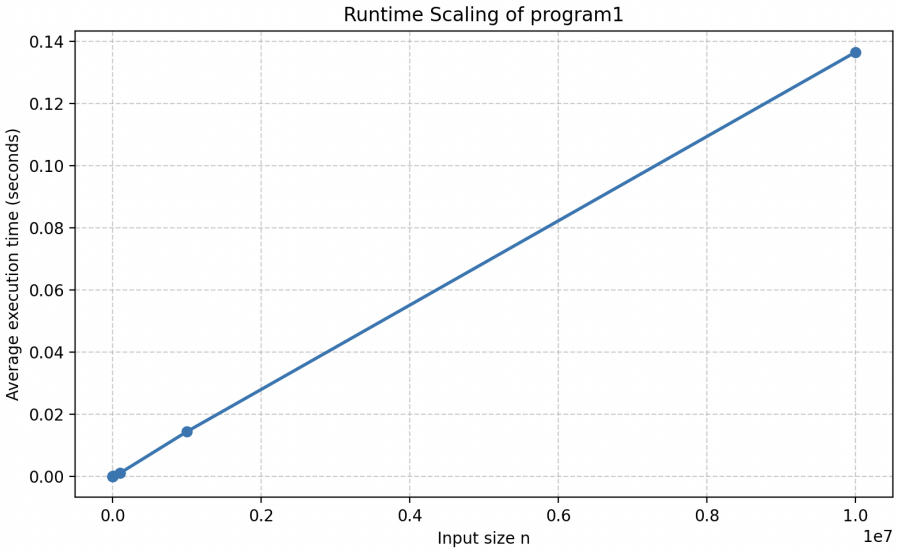
3 Experimental Comparative Study

3.1 Experimental Setup

To test our functions, we generated random k values and vault treasure values based on a set n value. The n values we tested were: [10, 100, 1000, 10000, 100000, 1000000, 10000000]. This allowed a good range of multiple n values to see how our algorithms scaled across a wide range of input sizes. The k values were allowed to be anywhere from 1 to n-1 and were randomly chosen at the beginning of each input test.

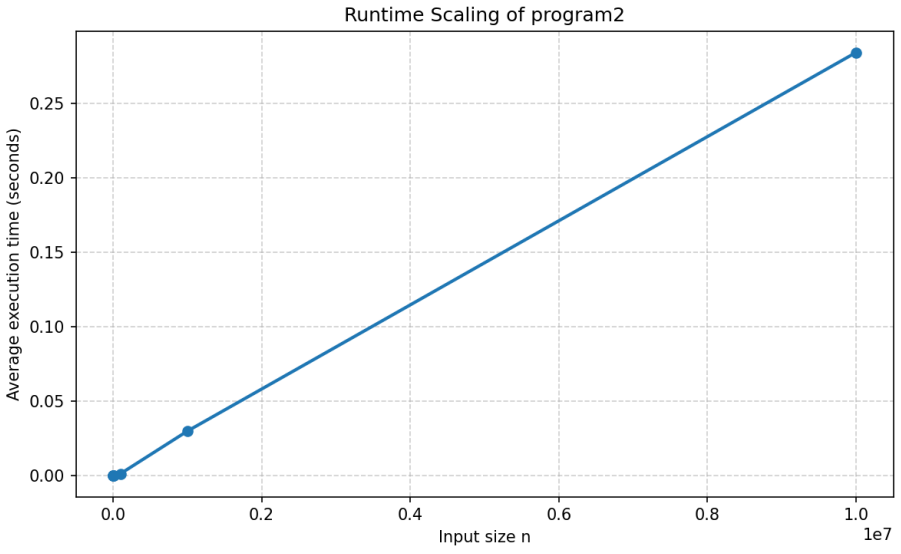
To correctly set up the treasure chest values, program1 generated values in ascending order. Program2, on the other hand, took the final list of generated values and split it in half. One half was sorted in ascending order and the other in descending order; they were then concatenated by placing the descending half in front of the ascending half. This created a local minimum to replicate the situation of S2.

3.2 Plot 1



The graph (produced from Python's MatPlotLib tool) shows the average runtime of program1 at multiple different inputs. . Each input was tested 500 times to get an average runtime for each input. As you can see from the graph, the program grows linearly. This follows that the program runs in O(n) time.

3.3 Plot 2



The graph (produced with the same tool as Plot 1) shows the average runtime of

the rightmost-to-leftmost algorithm with varying input sizes. This experiment resulted in a linear growth pattern, which implies that the program runs in O(n) time. This lies within the expected O(n^2) boundary calculated previously.

3.4 Observations/Comments

Program1:

- All of the input sizes ran fairly quickly, despite the range of input values tested.

Program2:

- The graph for program2 is comparable in growth shape to program1, but with considerably higher values.

Overall:

- There may be some limitations in how the generator was creating k values. The experimental study did not consider how a range of k values could impact the algorithm, as only one k value for each input size was tested.

- There may be some limitations in how the generator created the values of the treasure vault. The experimental study prioritizing setting up the correct order of vaults based on project specifications, so all the values for each input size tested is the same.

4 Conclusion

Milestone 1 was a fun challenge. Although implementing the algorithms was not difficult, measuring and comparing performance was. It required research into many Python libraries. A real complication was to figure out how to generate values to get accurate performance results but not sacrifice runtime, then to display them. There was discussion around the creation and analysis of the algorithms. This proved to be quite time consuming.