Report on Omnidroid Construction using Memoized Dynamic Programming

All the necessary files that are required to run the program are attached with this file in the zip folder.

- Input folder(It has all inputs)
- main.py(the main functionality)
- Readme.pdf (All the explanation is in this pdf)

First let me give a brief description of the problem

- The main task is to calculate the total sprocket cost that requires to assemble an omnidroid
- The problem needs to be solved using dynamic programming, recursion and memoization which is required to efficiently find the cost of assembling parts.

Here is the brief overview of the Algorithm

The solution is designed in such a way that

- Dynamic programming is used to solve by breaking into smaller problems, where the cost of assembling for each part happens recursively.
- Memoization, a dictionary(memo) is used to store the computed costs for parts making sure that each part is being used only once.
- Recursion is used to calculate the cost of each part by summing the sprocket cost of the part and the part and the cost of its dependencies.

Answer to the Project Questions:

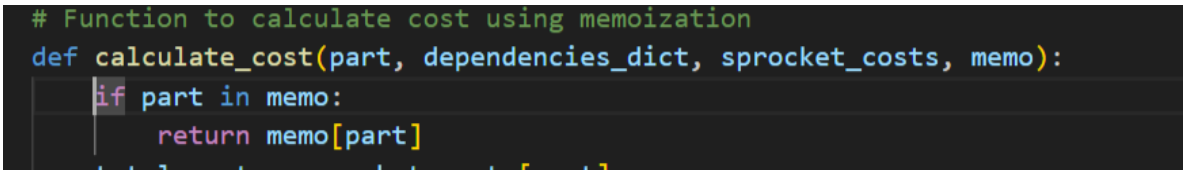
1. The solution is using memoized dynamic programming to break the problem into smaller instances.

- The problem is
  - The total cost is calculated by breaking into smaller parts.
  - Each part is calculated as the sprocket cost itself and the cost of all parts i.e., dependencies.
  - The relationship is cost(i) = sprocket_costs[i]+ $\sum_{j \in dependencies[i]}$ cost(j)
  - The recursion helps to solve incrementally starting from simple parts and progressively making the final omnidroid.
- Subproblems are solved as
  - The cost of each part is computed recursively with the function first resolving the costs of all dependencies.
  - For instance to do the cost of the part i, the algorithm recursively calculates each part of j in its dependency list and adds their cost and stores the result.

- Constructing the solution from subproblem
    - The solution for the final problem is calculated by combining the solution of subproblems.
      Firstly, basic parts with no other dependencies available provide the direct sprocket costs and then the middle parts are computed by adding the sprockets costs to the sum of costs of their dependencies.
    - The recursion finally see the cost of omnidroid i.e., part n-1 after calculating the costs of all its dependencies.
- Memoization role is
    - The memoization stores the precomputed cost of each part in the memo.
    - If cost of the part is calculated already and stored in memo, the function retrieves it instead of recalculating it.
    - This make sure that the cost part is calculated only once avoiding redundant computations.
- The solution efficiency
    - The recursion and memoization ensures solution is both correct and efficient.
    - By solving each subproblem only once and reusing the results the algorithm avoids the exponential complexity of naive recursion.

2. The base case of solution are

- The base case for memoization is that, if a part cost is already calculated then it is stored in memo. The function retrieves this precomputed value and terminates further recursion in that part.



```
# Function to calculate cost using memoization
def calculate_cost(part, dependencies_dict, sprocket_costs, memo):
    if part in memo:
        return memo[part]
```

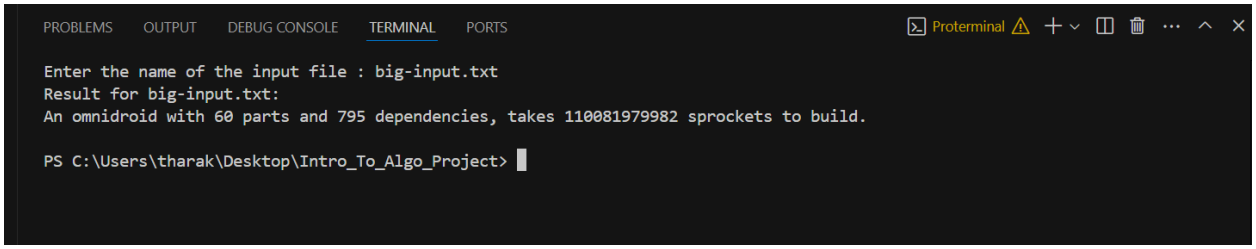
- Base part for base case is that, the cost is directly taken from the sprocket_costs because there is no subparts for further process(Here base parts are with no dependencies). The base cases ensure efficiency and simplicity.

3. The worst case is O(n+m), n is number of parts in omnidroid assembly and m is number of dependencies between the parts. This is because

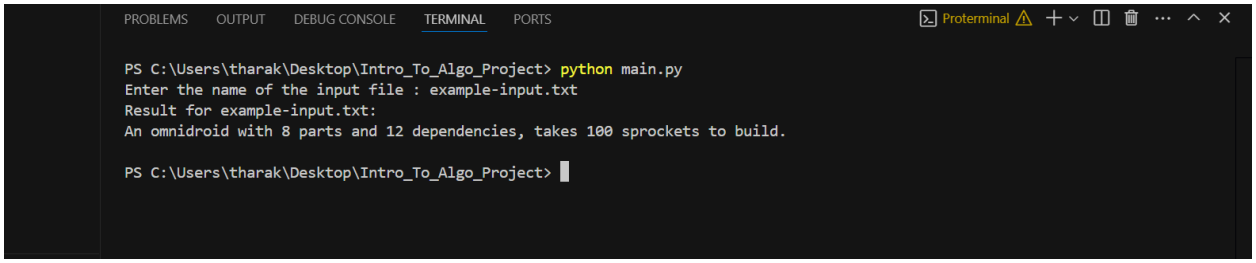
- If we process each part, we know each part is processed only once and cost is calculated recursively. It is stored in memo to avoid redundant calculations. So work done for parts is proportional to n.

- The dependencies are stored in a dictionary and each dependency is traversed once during recursive computation of cost of part. Here there are m dependencies so O(m).
- If we combine both then O(n+m)

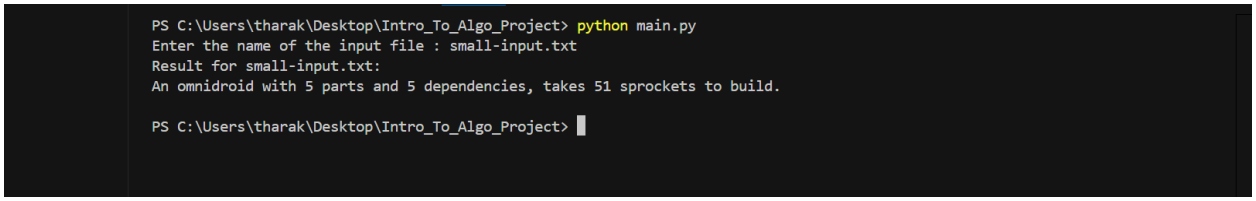
Output :



This is for big-input file



This is for example-input file



This is for small-input file