Analysis of Algorithms - Project 2 Report

Sebastian Rivera, Anna Khalus, Siwon Kim

1. How you can break down a problem instance of the omnidroid construction problem into one or more smaller instances? You may use sprocket[t] to represent the number of sprockets used for part t, and you may use req[t] and use[t] to represent the collection of all parts required to build part t and all parts that part t is used to build, respectively. Your answer should include how the solution to the original problem is constructed from the subproblems.

For each step in the omnidroid construction problem, there exists a set of subproblems we can identify and combine together to solve the greater whole. Within sprocket[t] we store the number of sprockets needed to build each part, we initialize this array with zeros. For each step, we need to first identify which part we're building and which part that dependency relies on. If sprocket[t] of the part we're building is set to zero, we can fetch how many sprockets we need for this part from the use[t] array and store it into sprocket[t] so that it is no longer zero. We can then see if the number of parts for a required part has been calculated yet for the current part being assembled. If the value for sprocket[t] of the dependency is zero, we can infer that the part is a primitive part constructed only of sprockets and no other dependencies; in this case, we can fetch the number of sprockets necessary from the use[t] array and add it to our sprocket[t] array for the current part being assembled. These distinct subproblems, summarily described as:

1. Identify if use[t] parts for the current part have already been saved to sprocket[t] and add if needed
2. Figure out the number of sprockets needed for the dependency part of the current part being assembled.
    a. Determine if the part is a primitive part, if so use sprocket count from use[t]
    b. If not primitive part, use sprocket count saved in sprocket[t]

2. What is the base cases of the omnidroid construction problem?

The base case for the omnidroid construction problem occurs when the size of the queue of parts evaluates to zero, otherwise, it calls the recursive method again. During each call of the omnidroid problem, we remove an item from the parts queue and determine how many sprockets are needed, we then evaluate our parts queue again until we have no parts left to consider.

3. How you can break down a problem instance of robotomaton construction problem into one or more smaller instances? You should assume that you are given sprocket and previous arrays that indicate the number of sprockets required for each stage of construction and the number of previous stages used to construct a particular part. Your answer should include how the solution to the subproblems are combined together to solve the original problem.

For each step in the robotomaton construction problem, there exists a set of subproblems we can identify and combine together to solve the greater whole. Robotomaton algorithm starts at robotomaton_wrapper function and sprocketsPerStep[n] array is used as an abstract data structure for memoization. Within sprocketsPerStep[n], we initialize this array with -1. We begin the recursive function by calling robotomaton_recursive function. For each stage (subproblems) in recursive, we store the number of sockets needed depending on the number of previous stages. If there are no previous stages involved, the function will encounter if(stages[n].p == 0) condition and sprocketsPerStep[n] stores only the number of its sprockets. If there are previous stages involved, the function will iterate sprocketsPerStep array from right before the current stage and iterate backward (the number of previous stages) times. While iterating, sprocketsPerStep[previousStage] will be added to sprocketsPerStep[currentStage]. After all iterations, sprocketsPerStep[currentStage] will store the total number of sprockets needed for that stage. These distinct subproblems, summarily described as:

1. Save the number of sprockets involved in the array data structure.
2. Figure out the number of sprockets needed for the current part being assembled.
    a. Determine if the part is a primitive part, if so, store only the number of current sprockets.
    b. If not the primitive part, add up the number of sprockets of previous stages from sprocketsPerStep array.

With these subproblems, we do not need to calculate the number of sprockets needed for the previous stages every time we iterate stages. Instead, we store the number of sprockets needed for the previous stages from the memoization array and add the saved values depending on the number of previous stages involved in the current stage recursively. After all recursive functions are called, each subproblem is combined together to solve the original problem.

4. What are the base cases of the robotomaton construction problem?

The base case in robotomaton is the stage that has no previous stage. Especially, the first stage is always zero since there are no previous stages trivially. After all recursive calls occur, the memoization data structure sprocketsPerStep[n] will store the total number of sprockets needed for each “stage” and the last index will store the total number of sprockets needed for each “robot”.

5. What data structure would you use to recognize repeated problems for each problem (two answers)? You should describe both the abstract data structures, as well as their implementations.

- Omnidroid: An int array of size n that keeps track of how many sprockets each part needs to be built. This allows us to access the total sprockets for a part that later ends up being a dependency, without having to recalculate the sprockets needed for that part.
- Robotomaton: An int array of size n that keeps track of how many sprockets each stage needs to be built. This allows us to access the total sprockets for a stage, without having to recalculate the sprockets needed for that part.

6. Give pseudocode for a memoized dynamic programming algorithm to calculate the sprockets needed to construct an omnidroid.

```
Algorithm: Omnidroid(), RecOmnidroid()

Inputs:
partQueue - Queue of parts {(partToBuild, Dependency)}
sprocketsPerPart - Array of sprockets per part

Output:
TotalSprockets[n] - Array of final count of sprockets needed to construct omnidriods

Code:
Omnidroid():
    Initialize all values in TotalSprockets[] to 0
    Call RecOmnidroid(partQueue, sprocketsPerPart, TotalSprockets)
    Return TotalSprocketsl[n-1];

RecOmnidroid():
    tempPart = partQueue.front()
    Pop from partQueue
    Int Dependency = tempPart.partToBuild;
    Int partToBuild = tempPart.Dependency;

    If (TotalSprokets[partToBuild] == 0)
        Assign TotalSprockets[partToBuild] = sprocketsPerPart[partToBuild]
    End

    if(TotalSprokets[Dependency] == 0)
        TotalSprockets[partToBuild] += sprocketsPerPart[Dependency]
    end
    Else
```

```
                TotalSprokets[partToBuild] += TotalSprokets[Dependency]
        End

        Repeat until queue is not empty: Call RecOmnidroid(partQueue,
sprocketsPerPart, TotalSprockets)
```

7. What is the worst-case time complexity of your memoized algorithm for the omnidroid construction problem?

The worst-case time complexity for the omnidroid problem is O(m) since the number of recursive calls made is determined by the number of parts in the partsStack (defined by m). Since our algorithm does not make use of any loops (aside from the recursive call) and all our other commands run in O(1) time (accessing an array, writing to an array, getting front of a queue, popping from a queue), our algorithm is dominated by O(m).

8. Give pseudocode for a memoized dynamic programming algorithm to calculate the sprockets needed to construct a robotomaton.

```
Algorithm: Robotomaton(), RecRobotomaton()

Input: stageList - Array of n stages {(sprocketsNeeded, prevStagesNeeded)}

Output: Total number of sprockets

Robotomaton():
        Define array sprStage[] and initialize all values to -1
        Int TotalSprockets =  RecRobotomaton(n-1, stageList, sprPerStage)
        Return TotalSprockets

RecRobotomaton(stage#, stageList, sprPerStage):
        If sprPerStage[stage#] != -1
                return sprPerStage[stage#]
        end

        If stageList[stage#].prevStagesNeeded == 0
                sprPerStage[stage#] = stageList[stage#].sprocketsNeeded
                return sprPerStage[stage#]
        End

        Else
                sprPerStage[stage#] = stageList[stage#].sprocketsNeeded
                Int prevStage = stageList[stage#].prevStagesNeeded
```

```
            For (int j = stage# - 1; prevStage > 0; j--)
                  sprPerStage[stage#] = sprPerStage[stage#] + RecRobotomaton(j,
      stageList, sprPerStage)
                  prevStage = prevStage - 1
            End

            return sprPerStage[stage#]
```

9. Give pseudocode for an iterative algorithm to calculate the sprockets needed to construct a robotomaton. This algorithm does not need to have a reduced space complexity, but it should have asymptotically optimal time complexity.

```
Algorithm: Robotomaton()

Input: stageList - Array of n stages {(sprocketsNeeded, prevStagesNeeded)}

Robotomaton():
      Define sprPerStage[n] and initialize all values to 0

      For each stage from stage# = 0 to stage# = n-1
            sprPerStage[stage#] = stageList[stage#].sprocketsNeeded

            If (stageList[stage#].prevStagesNeeded == 0)
                  Do nothing and go to the next iteration
            End

            Int prevStage =  stageList[stage#].prevStagesNeeded
            For (int j = stage# - 1; j >= stage# - prevStage; j--)
                  sprPerStage[stage#] += sprPerStage[j]
            End
      End

      Return sprPerStage[n -1]
```