Project 2 Report

Alexander Ashmore, Cameron Perdomo, Daniel Somoano

Question 1

How can you 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.

The omnidroid problem takes in an input of the number of parts, the number of sprockets required to build those parts, and the dependencies of those parts. The goal is to calculate the total number of sprockets required to build the completed omnidroid robot.

There are three categories that can describe each part (different from basic or intermediate):

Case 1

- The part can represent the fully constructed omnidroid: t = (ID n - 1).
- The value of sprockets[t] for part t is not empty.
- This part will also have some value of req[t], describing all parts required to build part t, the omnidroid.
- This part will also have no values for use[t] as this is the final constructed part and is not used to build anything else.

Case 2

- The part represents a basic part, being made of no intermediate parts and only of some number of sprockets.
- The value of sprockets[t] for part t is not empty.
- The value of req[t] is empty.
- There will be some value for use[t].

Case 3

- This part represents all other parts of construction that are intermediate parts but are not the final omnidroid.
- The value of sprockets[t] for part t is not empty.
- The value of req[t] has some value or multiple values, needing some number of parts to construct case 3's part.
- The value of use[t] has some value as some other part is dependent on it.

For each instance, we need to check the values for sprockets[t], req[t], and use[t]. If we are in case 1, that means use[t] (t = (ID n - 1)) is empty and the part is not used to build anything else. For each value stored in req[t] of the previous subproblem we will check the values of req[t] for each subsequent subproblem.

The part will now either be a case 2 or case 3. If it is a case 3, for each value stored in req[t] of the previous subproblem we will check the values of req[t] for each subsequent subproblem. This will repeat until the subproblem becomes a case 2, in which there are no values stored in req[t] for that part. This

means that part t is a basic part and requires no other parts are required to build it besides the stored sprockets[t]. We now can check the value stored in sprockets[t] and add that value to the total sprocket count. We go up to the previous subproblem and check the value stored in sprockets[t] and add the value to the total sprocket count. This will be repeated for every subproblem until we get to case 1 where use[t] is empty. We check the value stored in sprockets[t] and add the value to the total sprocket count when use[t] is empty. If use[t] is empty, that means we fully traversed all dependencies and the total
number of sprockets to construct the omnidroid is calculated. By traversing each part's req[t] until req[t] is empty, then traveling back on the path while adding sprockets[t] for each part to the total sprocket count, we can solve the original problem from multiple subproblems.

Question 2

What are the base cases of the omnidroid construction problem?

The base case for omnidroids is when the part, t, is not dependent on any other parts to build it.
In terms of question 1, there are no req[t] for part t, meaning the part will be a basic part, based solely on combining together some number of sprockets in the correct configuration.

Question 3

How can you 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 is combined together to solve the original problem.

The robotomaton construction problem gives us the number of sprockets, s, and the previous arrays, p, indicating the number of stages used to construct part t.
Each problem in the robotomaton construction problem can be broken into two case:
Case 1:

- The stage is dependent on some number, p of previous stages to construct.
- The stage requires some number of sprockets, s, to construct itself(excluding previous stage sprocket values).

Case 2(base case):

- The stage is not dependent on some number, p, of previous stages to construct.
- The stage requires some number of sprockets, s, to construct.

Given:

- stageReq[] stroes number of previous stages used to construct a particular part.
- Sprockets[] stores the number of sprockets required for each stage of construction.

We first check stageReq[] for every problem. If stageReq[i] is 0, that stage is a case 2, not dependent on any other stages to calculate. If this is true, then we can look directly at sprockets[i] at that stage and return that value. If stageReq[i] is not 0, that means that that stage is dependent on some number of stages p. We can look p stages back and check for each stage from I to (i – p) stages if stageReq[i] is 0. For each subsequent stage, we repeat the steps above until we hit a case 2 where stageReq[i] = 0. The original problem can be solved by summing the number of sprockets for each stage once we hit the base

case and return the total number of sprockets. The original problem can be solved by traversing back up the consequent problems while summing the number of sprockets, s, by checking sprockets[i] for each stage.

Question 4

What are the base cases of the robotomaton construction problem?
The base case of the robotomaton problem is when the previous arrays used to construct a part is 0. This means there are no previous stages required to calculate the number of sprockets of that stage.

Question 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:
We used a map to store the number of basic parts for each intermediate part. Then to flatten to 1d array and memoized through that. We used a vector for lookup table of the memorized function.

Robotomaton:
We can use a 1-dimensional array to store the values of the sprockets calculated at each stage. The nth position in the array will correspond with the final stage of construction and store the total amount of sprockets to create the robotomaton. To recognize repeated problems, the array should be initialized to –1 as the amount of sprockets should never be negative. The final value of sprockets for the stage will be stored in the corresponding position in the array. If that stage is found again during the algorithm process, it should return that value rather than computing the problem again.

Question 6

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

Input: p: vector of PartsByID
Input: s: vector of Sprockets from Input File
Input: t: int of total parts from Input File
Output: MemoizedSprocket's output
Algorithm 6.1 TotalSprocketWrapper

```
create vector for sprocket lookup (sl)
for i to t + 1 do
        populate sl with sentinel value(-1);

return memoizedSprockets(p,  s, sl, t);
```

Input: p: vector of PartsByID
Input: s: vector of Sprockets from Input File
Input: sl: vector of Sprocket Lookup

Input: t: int of total parts from Input File
Output: Outputs Final Sprocket Count
Algorithm 6.2 memoizedSprocket

```
if t < 0 then
        return 0;
if sl[t] == -1 then

        if t < 0 then
                return sl[t];
        else
                sl[t] = s[t] * p[t] + memorizedSprockets(p, s, sl, t -1);
return sl[t];
```

Question 7

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

The worst case time complexity for the memoized omnidroid problem is T(n) + O(n).

Question 8

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

Input: n: integer of number of stages
Input: stageReq[]: array of stage dependency
Input: sprockets[]: array of number of sprockets per stage dependency
Output: Outputs total sprocket count in final stage
Algorithm: calcRoboto

```
Cost = [n];
Initialize cost to -1;
Return memoRobotomaton(n, stageReq, sprockets)
```

Algorithm: memoRobotomaton

```
Int i  = n -1;      //tracks index
int total = 0;      //sum of all sprockets needed

if cost[i] != -1 then
        //base case
        If(stageReq[i] == 0) then
                Total = sprockets[i];
                Return total;
        Else
                Int j = stageReq[i];
                For (; j>0; j--) do
```

```
                    Total = sprockets[i] + memoRobotomaton(I, stageReq, sprockets);
                End for
            End else
            Cost[i] = total;
            Return total;
        End if
```

Question 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.

```
Input: n: int of number of stages
Input: stageReq[]: array of stage dependency
Input: sprockets[]: array of number of sprockets per stage dependency
Output: iterateRobotomaton[n – 1]: Outputs total sprocket count in final stage
Algorithm iterateRobotomaton

    int i = 0;
    while i < n do
        if stageReq[i] != 0 then;
            int j = stageReq[i];
            while j > 0 do
                iterateRobotomaton[i] = iterateRobotomaton[i] + iterateRobotomaton[i -j];
                j--;
            end while
        end if
        i++;
    end while
    return iterateRobotomaton[n – 1];
```