Joanna McDonald
Homework 3

Problem 1

We have three containers whose sizes are A pints, B pints and C pints, respectively, where A, B, C are all positive integers. In the beginning, the A-pint container has a pints of water, the B-pint container has b pints of water, and the C-pint container has c pints of water, where a, b and c are non-negative integers. (For example, we might have $A = 10$, $B = 7$, $C = 4$ and $a = 0$, $b = 6$, $c = 4$.) We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that, in the end, leaves exactly k pints of water in any of the three containers. (So the answer we seek is either YES or NO.)

Your task:

1) Model this problem as a graph problem: give a precise definition of the graph involved, and state the specific question about this graph that needs to be answered.
2) Design an efficient algorithm to solve the above problem.

Main idea/model the problem:

The graph involved will be a tree, so, a directed acyclic graph. Each node will be a list of 3 different objects, each object representing one of the three glasses. The object will store the current volume (a, b, c) and the maximum volume (A, B, C) of the container. At each node there will be 6 nodes branching off, each representing the 6 possible pours for each step. As the tree continues to grow for different pour possibilities, the different "states" (nodes) should be stored. That is, the values of the 3 pint containers. All of the possible pour state nodes as they're made are added to the queue under certain conditions. If a new node created has the same "state" as one that has already been identified, the tree should not continue to make or build off that node and will not be added to the queue. After one cycle of traversing the possible new state nodes branching off of one node, the cycle will continue, following the order of the queue. To do this, the queue.poll function is used on the queue to set the current state to the value of the element at the front of the queue. The question to be answered is, will the tree, as it grows, reach a state where one of the pint volumes in a state is the same as the value of k? Or, will the tree come to a halt, meaning no new states can be built and the queue is empty? The graph problem will be solved using BFS, breadth-first search.

main idea, pseudo code, proof of correctness and time complexity analysis) are always needed.

Proof of Correctness

Proof 1:

Theorem: Given an initial node (a, b, c), there are no more or less than 6 possible combinations of adding one part (either a, b, or c) of the node to another. For each index, (0=a, 1=b, 2=c), there are two distinct combinations to make a unique value while using the item in the index.

Proof by Contradiction:
Assumption: Given an initial node (a, b, c), there are no more or less than 6 possible combinations of adding one part (either a, b, or c) of the node to another.
For each index, (0=a, 1=b, 2=c), there are three distinct combinations to make a unique value while using the item in the index.

Given 3 distinct combinations for each index, and 3 indices, there must be 3*3 = 9 possible combinations of adding one part of the node to another. Thus, we have reached a contradiction. The only flaw in our reasoning is the initial assumption that for each index, (0=a, 1=b, 2=c), there are three distinct combinations to make a unique value while using the item in the index. Thus, we conclude that the original theorem is correct. ■

Proof 2:
In a BFS, the order in which vertices are removed from the queue is always such that if u is removed before v, then dist[u] ≤ dist[v].

Theorem: When the algorithm's queue is empty, then breadth first search of the tree is complete.

Assumption: When the algorithm's queue is empty, the breadth first search of the tree is not complete.

By definition of a queue, the queue is a first-in-first-out data structure. In breadth first search, the first item "in" the queue is the parent node, and the following items in the queue are the child nodes of the first item in the queue.
In BFS of our tree, the queue is filled each time a new layer of the tree is made, and new entries to our queue will only be added if they are unique nodes that have not occurred before.
If our queue is empty, this must mean that there are no longer unique nodes to create and analyze. Therefore, with an empty queue, no further search of the tree is possible. Thus, we have reached a contradiction. The only flaw in our reasoning is the initial assumption that when the algorithm's queue is empty, the breadth first search of the tree is not complete. ■

<u>Pseudocode</u>
Define initial node (a, b, c), target pint volume k, and max volumes of the containers (A, B, C)
If (maximum value of A, B, C is less than k, OR k == 0):
    Then return False
If current node includes k at one of the values:
    Then return True
Make "usedStates" to store the states that have already been made as nodes.

```
Add initial node to “usedStates”
Make a queue, “queue” for BFS
Add initial node to “queue”
“isPossible” = False
While( queue is not empty):
        “currentState” = queue poll
        If(currentState[0] == k || currentState[1] == k || currentState[2] == k):
                isPossible = True
                break
        For i in the range from (0 to length of currentState):
                For j in the range from (0 to length of currentState):
                        If(i doesn’t equal j) //not trying to add same pint volumes
                                “cupI” = value for cup i in currentState
                                “cupJ” = value for cup j in currentState
                                “availablePour” = value for max pints in current container J – cupJ
                                If (cupI < availablePour):
                                        “toPour” = cupI
                                Else “toPour” = availablePour
                                If(cupI != 0):
                                        “newState” = copy currentState
                                        newState[i] = cupI – toPour
                                        newState[j] = newState[i] + toPour
                                if(newState
                                if(usedStates does not contain newState):
                                        add newState to usedStates
                                        add newState to queue

return isPossible
```

Time Complexity Analysis:

By using breadth first search, and a tree directed acyclic graph structure, layer by layer of tree will be built until a solution is reached. As each layer is built, all different states(nodes) in that layer will be checked by the algorithm. It is a connected graph. Therefore, the time complexity depends on the number of edges and will be O(E).

Problem 2

Problem

Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, m_1, m_2, $\cdots$, m_n, where each m_i is an integer (for $i = 1, 2, \cdots, n$). The constraints are as follows:

1) At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \cdots, n$.
2) Any two restaurants should be at least k miles apart, where k is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

Main Idea

This algorithm uses dynamic programming and recursion to compute the maximum expected total profit subject to the given constraints. It uses 2 for loops to navigate through the possible maximum expected profits at each location i. The expected profits are used to compare values and generate the maximum expected total profit. It works to determine which locations should be included to maximize the expected total profit.

In the start of the first for loop, the current value for the maximum expected total profit is initialized as the expected profit from opening a restaurant at that location i. At each index location i, the 2nd for loop checks previous locations at indices j to see if the distance between them is greater than or equal to k, the required distance between any two restaurants. It then navigates through those valid locations' previously calculated maximum expected profits at the j index and adds the expected profit for the current location. This value is compared to the current calculation for the maximum expected profits at index i, to determine if it is eligible to replace the current calculation for the maximum expected profits. After exiting the second for loop and before checking the next index with the first for loop, the current maximum expected total profit is compared to the maximum expected total profit found at the previous index. Whichever value is greater is assigned as the current maximum expected total profit for this index. This continues for the j indices until the maximum expected profit for the location at index i is found. This repeats for each i to cover all n locations and find the maximum expected profit.

The definition of S is the list of maximum expected profit for the location of the current index, and S[i] is the maximum expected profit at location i. For i>j:

$$S[i] = max\begin{cases} p_i \\ p_i\left(gr(m_i, m_j)\right) + S[j] \end{cases}$$

$$\text{def } gr\left(m_i, m_j\right) = \begin{cases} 1 \; if \; m_i - \; m_j \; \geq k \\ 0 \; if \; m_i - \; m_j < k \end{cases}$$

Proof of Correctness

Theorem: If mi-mj < k, then the maximum expected profit at location i is the same as the maximum expected profit at location j.
If S[i] and S[j] are equal, and S[i] > pi, then mi-mj < k.

Proof by contradiction:
Assumption: If $m_i - m_j \geq k$, and $S[i] > p_i$, then $S[i]$ and $S[j]$ are equal.

By the definition of $gr(m_i, m_j)$), it is equal to 1 if $m_i - m_j \geq k$ and equal to 0 if $m_i - m_j < k$.
Given that $m_i - m_j \geq k$, $gr(m_i, m_j)$ = 1.
By the definition of $S[i]$, it is equal to the maximum between two values: p_i, and $p_i\left(gr(m_i, m_j)\right) + S[j]$. We just solved for $gr(m_i, m_j)$ and found it is equal to 1.
So, $p_i\left(gr(m_i, m_j)\right) + S[j] = p_i(1) + S[j] = p_i + S[j]$.
The other value to compare this to is p_i.
F $p_i < p_i + S[j]$. So S[i] is equal to $p_i + S[j]$ when $m_i, m_j \geq k$.
Thus, we have reached a contradiction. The only flaw in our reasoning is the initial assumption that if $m_i - m_j \geq k$, and $S[i] > p_i$, then $S[i]$ and $S[j]$ are equal. Thus, we conclude that the original theorem is correct. ■

Assume that mi-mj < k.

The maximum expected profit at location j, S[j],

If mi-mj < k, then
<u>Pseudocode</u>

P = $[p_1, p_2, p_3, \ldots, p_i, \ldots, p_n]$ //list of expected profits at location i = 1, 2, …, n
L = $[m_1, m_2, m_3, \ldots, m_i, \ldots, m_n]$ //list of locations
S = []
lengthL = length of list L

```
//initialize list to put max profit of each location with values of 0
For i in P:
        S[i] = 0
For i in range(1, n+1):
        S[i] = P[i]
        For j in range(i -1):
```

If L[i] – L[j] is greater than or equal to k: //if space between locations ≥ k
//space b/t locations is ≥ k, so space is more than min distance
S[i] = maximum of (S[i], P[i] + S[j])
//equals either current max price value for i, or value for prev max price val plus expected profit value for i

//max profit for a restaurant at position i if it exists or for previous restaurant
S[i] = maximum of (S[i], S[i-1])
Return S[n-1] //last two values will have same value

Time Complexity Analysis

The time complexity of the first for loop, which just defines an initialized list with all zeros as the values, is O(n). The second for loop, the outermost one "for i in range(1, n+1)", has a time complexity of O(n). The nested for loop, has a time complexity of $O(n^2)$, and when establishing time complexity in this case, the worst case should be assumed, so the overall time complexity of the algorithm is $O(n^2)$.