Project 2: Dynamic programming: The Lazy Mover

Group 3: Joseph Cox, Adrian Borrego, and Ricardo Montanez

NOvember 9, 2018

General Overview of the Program

We have two files *move.hpp* and *moveImp.cpp*

1. **Move.hpp**: This file contains the recursive function called *findGreatest()* that solves each problem set inside the input.txt. It also has helper functions that work with it to modulate some of the logic and memoize the algorithm. There are no calsses in this file, it only contains the necessary functions to solve the problem. The functions are all global.

2. **MoveImp.cpp** This file contains the *main()* and reades the input.txt with a string stream, and creates an output.txt. The string stream feeds the *findGreatestHealper()* function which calls the *findGreatest()* function inside of it. It returns an integer with the correct answer to each problem set. The correct answer is then appended to the end of the file with a new line chracter. At the end of the *main()* function all the files are closed

Complier Information

We use the gcc c++14 compiler, however we tested the progam on the student cluster and it works on the student cluster utilizing the *g++ -Wall move.hpp moveImp.cpp* and no warnings or errors have emerged. However a makefile has been provided for convinence. **To use the makefile on the student cluster you must change the CC from *CC=g++ -std=c++14* to *CC=g++* the student cluster does not run c++14 or c++11**

Makefile

This make file has 5 functions that can be exucted in the following way:

1. $make - this will compile the program without running it.
2. $make run - this will run the program only after make(the above command) has been called
3. $make clean - this will delete all the object files and it will also delete the executable
4. $make cleanAll - this will delte all the objectfiles, the executable and it will delete the output.txt
5. **$make it - this will compile the program and run the program all at once, since the program is ran it will generate an output.txt**

Report Problems

Problem 1

A single problem instance of the lazy mover problem must look at the boxes in front of it to see what can be picked up. If the lazy mover chooses box number n_i. The lazy mover should be able to look at the options of boxes in front of it choose the best box. To do this we must already know what boxes can be picked up starting at n_{i+1} and n_{i+2}... n_n.In short if we can figure out what boxes can be picked up starting at a box in front of n then we know what boxes can be picked up at n. This means all of the sub problems occur in front of n at n_n which is at the back of the list

Problem 2

The base case of this recurrence is one box at n_n which is at the end of the list. If the lazy mover starts at this box it will only be able to pick up 1 box because it is the very last box of the list. This means that we can solve the problem recursively going backwards through the list starting at at n_n. We can find out how many boxes can be picked up then move through the list backwards. This means the we will then have a better idea of how many boxes can be picked up at n_{n-1},n_{n-2} ... until we reach n_0

Problem 3

The data structure that we could use is a direct map, this could be implemented with an array or vector. These data structures would work well with this problem because the lazy mover only cares about the maximum number of boxes that can be picked up at box number n_i. This means that a mapping between the list of weights and the the number of boxes starting at each weight could be accomplished easily and quickly because lookup for an array or vector is O(1) constant time.

Problem 4

```
def healper(boxList,size,index):
    #Algorithm: findGreatestHelper
    #Input: boxList: vector of integers containing sequence of box values
    #                read from input.txt (positive value)
    #Input: size: size of boxList (positive integer)
    #Input: index i to start memoized  DP algorithm
    #Output: findGreatest(boxList, size, myBoxes, i)
    myBoxes.build(size, -1)
    return findGreatest(boxList, size, myBoxes, i)
```

```
def RecursiveFunction(boxList,size,index):
    #Recursive Function:
    #Algorithm: findGreatest
    #Input: boxList: vector of integers containing sequence of box
    #       values read from input.txt (positive value)
    #Input: size: size of boxList (positive integer)
    #Input: myBoxes: vector of integers containing sub-results of
    #               recursive calls on boxList
    #Input: i: starting index of boxList
    currentIndex = i
    arraySize = size
    max_count = 0
    count = 0

    if (index < size – 1):findGreatest(boxList,size,myBoxes,index + 1)

    else if(current index is the last box in boxList and
            myBoxes at index has sentinel -1):

        increment max_count by 1
        store max_count at myBoxes[currentIndex]



    else if(myBoxes[currentIndex + 1] has been solved
            and myBoxes[currentIndex] has sentinel -1):

        iterate through myBoxes for greatest
        value <= boxList[currentIndex] and
        keep track of greatest count in myBoxes
        from myBoxes[currentIndex + 1…n]
        store (count + 1) at myBoxes[currentIndex]

    else:

        perform naïve algorithm to find
        greatest number of boxes able to
        be picked up starting at current
        indexstore (count + 1)
        at myBoxes[currentIndex]

    return function call to mostBoxes() to return answer.
```

Problem 5

Since the memoized function uses a vector to store sub-results of recursive calls on index i and iterates through the vector to find the greatest number after nth recursive calls. The memorized function has a worst-case complexity of O(n2).

Problem 6

```
def NaïveIterativeAlgorithm():
    perform naïve algorithm on starting at index boxList[i]:
    if (boxList.size() is equal to zero):
        no boxes can be picked up
        return 0
        else:
        for (i to n):
            if (boxList[i] >= boxList[i + 1]):
                count + 1
                set i equal to i + 1 for new box on top
    return count + 1
```

Problem 7

Since we are using an iterative algorithm without memoization, the space complexity is only bounded by the container that contains the values which the algorithm will iterate through, there is no external storage dependencies where sub-results are stored. The algorithm will simply iterate through the input container that contains the sequence of boxes and perform the logic that is needed to find the greatest number of boxes that the mover is able to pick up. In this case, when we use a memoized approach to solving the problem we are trading space complexity over time complexity.

Problem 8

```
def Helper(boxList,size):
    # Algorithm: findGreatestHelper
    # Input: boxList: vector of integers containing sequence of box values
    #        read from input.txt (positive value)
    # Input: size: size of boxList (positive integer)
    # Output: findGreatest(boxList, size, myBoxes, index)
    myBoxes.build(size, -1)

    return findGreatest(boxList, size, myBoxes, index=0)
```

```
def RecursiveFunction(boxList,size,myBoxes,index):
    #Algorithm: findGreatest
    #Input: boxList: vector of integers containing sequence of box values
    #       read from input.txt (positive value)
    #Input: size: size of boxList (positive integer)
    #Input: myBoxes: vector of integers containing
    #       sub-results of recursive calls on boxList
    #Input: index: starting index of boxList

    currentIndex = index
    arraySize = size
    max_count = 0
    count = 0

    if (index < size – 1):findGreatest(boxList,size,myBoxes,index + 1)

    else if(current index is the last box in boxList and myBoxes at index
            has sentinel -1):

        increment max_count by 1
        store max_count at myBoxes[currentIndex]


    else if(myBoxes[currentIndex + 1] has been solved and
            myBoxes[currentIndex] has sentinel -1):

        iterate through myBoxes for greatest
        value <= boxList[currentIndex] and
        keep track of greatest count in myBoxes from
        myBoxes[currentIndex + 1…n]store (count + 1)
        at myBoxes[currentIndex]

    else:

        perform naïve algorithm to find greatest number of boxes able
        to be picked up at current indexstore (count + 1)
        at myBoxes[currentIndex]

    return function call to mostBoxes() to return answer.
```

Problem 9

```
def boxOrder(boxList,size,myBoxes,index)
    #Algorithm: BoxOrder
    #Input: boxList: vector of integers containing sequence of
    #       box values read from input.txt (positive value)
    #Input: size: size of boxList (positive integer)
    #Input: myBoxes: vector of integers containing sub-results
    #       of recursive calls on boxList
    #Input: index: starting index of boxList

    Array boxOrder[size].build()

    # to figure out greatest number of boxes that
    # mover is able to pick up.
    call findGreatest(boxList, size, myBoxes, index)
    iterate through myBoxes and find index which myBoxes[i] == answer
    perform naïve algorithm starting at index boxList[i]:
        for i to n:
                if boxList[i] >= boxList[i + 1]:
                        add value to boxOrder: boxOrder[] = boxList[i + 1]
                        set i equal to i + 1 for new box on top
    return boxOrder[] contains all boxes that mover was able to pick up
```