Project - Task 5

CSE332s: Design and Analysis of Algorithms

1. Problem Description

“There are n coins placed in a row. The goal is to form n/2 pairs of them by a sequence of moves. On the first move a single coin has to jump over one coin adjacent to it, on the second move a single coin has to jump over two adjacent coins, on the third move a single coin has to jump over three adjacent coins, and so on, until after n/2 moves n/2 coin pairs are formed. (On each move, a coin can jump right or left but it has to land on a single coin. Jumping over a coin pair counts as jumping over two coins. Any empty space between adjacent coins is ignored.) Determine all the values of n which the problem has a solution and design an algorithm that solves it in the minimum number of moves for those n’s. Design greedy algorithm to find minimum number of moves.”

2. Detailed Assumptions

We assumed that in both solutions either the greedy or the brute force, the algorithm will start from the very first coin in the left most of the row and work its way to the right direction. As it’s arbitrary to start from any coin and work its way in any direction and there’s no constraints or cases that will go against this assumption, so there’s no problem to assume that we will start from that coin and in that direction.

3. Detailed Solution

3.1. Key-Observation

If we try by hand to solve this problem, we will find that some values of n work fine and other doesn't, some of the values that works are 4, 12 and 28. We need to observe the pattern or the mathematical formula, and after that use it to deduce the next values, then finally test those values to see if the pattern or the mathematical formula was right or wrong. After looking for some formulas and pattern, there's only one simple pattern or formula that satisfies the problem constraints, and it is as follows:

If we assume that the value of n is equal to the sum of two big piles of mathematically calculated number of fours, then you could put the 4, 12 and 28 as follows: 4 = (0) + (4), 12 = (4) + (4+4), 28 = (4+4+4) + (4+4+4+4) then you we can deduce a mathematical formula based on that, and it is as follows:

n = ((2^(i)-1) + (2^ (i))) * 4 = (2^(i+3)) - 4 where i is a non-negative integer.

Then for i=0 -> n = 4, for i=1 -> n=12, for i=2 -> n=28, for i=3 -> n=60, etc.…

If you try n=60, it will work. So, the deduced formula is right, hence that was one of the requirements, what are the values of n that satisfies the problem constraints? **(2^(i+3)) – 4 where i is a non-negative integer.**

3.2. Greedy Technique

3.2.1. Key Idea

Simply start from the left most of the row, set the number of moves to zero and start moving the first coin according to the constraints of the problem to form a pair in the left direction, then next ***choose the nearest coin to the empty space that it was left behind (greedy step)***, increment the number of moves then repeat the steps again until you form n/2 coin pairs in just ***n/2 moves which is logically the minimum amount of moves required to form n/2 coin pairs.***

3.2.2. Pseudocode

3.2.2.1. High-Level

1. Set the current move to 0.
2. Start from the leftmost coin.
3. Iterate through the row until you reach the coin before last.
   3.1. For each single coin you find while iterating, start to the column to the left of it.
      3.1.1. Increase the current move by one.
      3.1.2. Set the count of jumped over coins for this specific coin to 0.
      3.1.3. If the current jumped over coins count equals the current move, then put the single coin on the current column to form a pair, then break out of this loop to go to the next nearest single coin to the previous one that we selected.
      3.1.4. Else if the current jumped over coins count doesn't equal the current move, then increase the jumped over coins by the number of coins in the current column and go to the next left column.

3.2.2.1. In Details

```
numberOfMoves <- 0
For i = 0 to n-1 {
    // skip empty spaces or coin pairs until you find the nearest coin (greedy step)
    if(rowOfCoins[i] != 1)
      continue

    numberOfMoves <- numberOfMoves + 1
    jumpedOverCoins <- 0

    // making a coin move to form a pair of coins
    For j=i+1 to n-1 {
      if (jumpedOverCoins == numberOfMoves) {
        rowOfCoins[i]  <- 0;
        rowOfCoins[j] <- rowOfCoins[j] + 1
        break
      }
      // counting the number of jumped over coins
      jumpedOverCoins <- jumpedOverCoins + rowOfCoins[j];
    }
}
```

3.3. Brute-Force Technique

3.3.1. Key Idea

Again simply start from the left most of the row, set the number of moves and start moving the first coin according to the constraints of the problem to form a pair in the right direction of the row, and if no place was found satisfying the constraints to form a coin pair then search in the left direction for a place, if also no place was found satisfying the constraints then we go back one step to the previous pair that was formed, unform it then try to find another different place for that previous singe coin to form a pair then go to the next single coin that we couldn't find a place for it to form a pair in the first place, then we try to find a place for it, ***eventually we will repeat the steps and make our way brute forcing it with those steps*** until we make n/2 coin pairs in just ***n/2 moves which is logically the minimum amount of moves required to form n/2 coin pairs.***

3.3.2. Pseudocode

3.3.2.1. High-Level

1. Start by calling the function to run at the left most position and operate in the right direction of the row.
2. Check if the position passed is valid, if not then return false.
3. Set the number of moves to 1 and make a flag that determines if we are done or not and set it to false.
4. Set the count of jumped over coins to 0.
5. Start iterating from the column to the to the left or right of the passed position as it was specified in the function call, and do the following in each iteration:
    5.1. If the current jumped over coins count equals the current move, then put the single coin at the specified position on the current column to form a coin pair, increase the current move number by one, set the flag to true, then finally break out of this loop to go to the next nearest single coin to the previous one that we selected.
    5.2. Else if the current jumped over coins are greater than the current move then breaks out of the loop.
    5.3. Else if the current jumped over coins count are less than the current move, then increase the jumped over coins by the number of coins in the current column and go to the next left or right column as the direction was specified at the beginning of the loop in the function call.
6. Check if there's still any single coins left and if so, set the flag to false.
7. If the flag is true, then return true.
8. Determine the next position of the next single coin that will be moved to form a coin pair.
9. Call the function again with the new position calculated in the right direction (go to step 2) and set the returned value out of the function to the flag.
10. If the flag is true, then return true.
11. Call the function again with the new position calculated in the left direction (go to step 2) and set the returned value out of the function to the flag.
12. Return the flag value.

3.3.2.1. In Details

```
algorithmUsingBruteForce (position, currentMoveNumber, direction) {

    if ((position < 0) || (position > n))
      return false

    numberOfMoves ← currentMoveNumber
    jumpedOverCoins ← 0
    doneFlag ← false

    // check in the right direction the possibility to form a pair
    if (direction == "right")) {
      for i ← position+1 to n-1 {
        // move the coin to the column if the condition is satisfied
        // or break if exceeded
        If (jumpedOverCoins >= numberOfMoves) {
          if (jumpedOverCoins == numberOfMoves) {
            rowOfCoins[position] ← 0
            rowOfCoins[i] ← rowOfCoins[i] + 1
            currentMoveNumber ← currentMoveNumber + 1
            doneFlag ← true
          }
          break
        }
        // or jump over the column and count the jumped over coins
        else if (jumpedOverCoins < numberOfMoves)
          jumpedOverCoins ← jumpedOverCoins + rowOfCoins[i]
      }
    }
```

```
// check in the left direction the possibility to form a pair
else if (direction == "left")) {
   for  i ← position-1 to 0 {
      // move the coin to the column if the condition is satisfied
      // or break if exceeded
      If (jumpedOverCoins >= numberOfMoves) {
         if (jumpedOverCoins == numberOfMoves) {
            rowOfCoins[position] ← 0
            rowOfCoins[i] ← rowOfCoins[i] + 1
            currentMoveNumber ← currentMoveNumber + 1
            doneFlag ← true
         }
         break
      }

      // or jump over the column and count the jumped over coins
      else if (jumpedOverCoins < numberOfMoves)
         jumpedOverCoins ← jumpedOverCoins + rowOfCoins[i]
   }
}

// check if there're still single coins
For i in rowOfCoins {
   if (i == 1) {
      doneFlag ← false
      break
   }
}
```

```
    // check if all are the coins are pairs
    // if so, then we are done
    if (doneFlag == true)
       return true

    // next single coin to be moved to form a pair
    int nextPosition ← position;
    for i ←position+1 to n-1{
       if (rowOfCoins[i] == 1){
          nextPosition ← i
          break
       }
    }

    doneFlag ← algorithmUsingBruteForce (nextPosition, currentMoveNumber, "right")

    // still, there're single coins?
    If (doneFlag == true)
       return true

    doneFlag ← algorithmUsingBruteForce (nextPosition, currentMoveNumber, "left")

    return doneFlag
}
```

4. Complexity Analysis

4.1. Greedy Technique Complexity Analysis

In general, you loop over n in the left direction skipping every empty space and every coin pair that was formed, until you find a single coin, then you start to iterate in the same left direction until you find the right spot to put that single coin so you form a coin pair, then repeat.

For the first coin, you will jump over one coin, for the second you will jump over two coins, etc.…

As an approximation we could say that the whole time is upper bounded by:

[1 + 2 + 3 + 4 + … + n/2] + n/2 where each number represents a coin that we iterated that number of times to find the right spot for it, except the last (n/2) it represents the skipped empty spaces and coin pairs that we looped through, but we didn't select any of them.

So, [1 + 2 + 3 + 4 + … + n/2] + n/2 = ((n/2) * (n/2 + 1))/2 + (n/2) = ((n^2) / 8) + (3n/4) which is O(n^2) in all cases.

4.2. Brute Force Technique Complexity Analysis

It is expected that the brute force technique will have a very high running time, higher than the greedy technique by a lot as it goes through a lot of paths most of them are wrong until it reaches the right path that leads to the solution, but the greedy chooses the greediest path from the start which is likely to be the solution or an approximate to it.

But in our case, and due to the assumption that we start in both algorithms from the left most coin and work our way in the right direction, the brute force technique will reach the right path from the first try as if it was the greedy technique, and then it will stop.

So, the brute force technique will be upper bounded by approximately the same running time as the greedy technique which is O(n^2) and that is in all cases.

4.3. Comparison Between the two techniques

As the two techniques has two different methods of finding the solution. But like discussed above, the two techniques will result in two different algorithms each will solve the problem and find the solution in approximately similar running time in all cases due to the problem nature and the assumption and implementations that were made.

5. Code in Java

5.1. Algorithm Using Greedy Technique

```java
////////////////////////////////////////////////////////////////////////////////
// number of coins
// n = 2^(i+3)-4  where i is non-negative integer [0,...]
// i.e. (i=0, n=4), (i=1, n=12), (i=2, n=28), ...
int n = 28;

// there's a row of coins
// 0 in a column -> empty column
// 1 in a column -> one coin
// 2 in a column -> pair of coins
int[] rowOfCoins = new int[n];

// number of moves that are going to be done to form the pairs
int numberOfMoves = 0;

// number of coins that a coin will jump over in a certain move
// it should be equal to the number of the move being made
private int jumpedOverCoins;
////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////
// Constructor
public JumpingCoins(){

    // each column in the row has just one coin at first
    for (int i = 0; i < n; i++)
        rowOfCoins[i]=1;

}
////////////////////////////////////////////////////////////////////////////////
```

```
66      ////////////////////////////////////////////////////////////////////////////////
67      // Solution using greedy technique
68      public void algorithmUsingGreedy(){
69
70          for(int i = 0; i < n-1; i++){
71
72              // skip any coin pairs (2) or empty spaces (0)
73              // skip until you find the nearest coin (greedy step)
74              if(rowOfCoins[i] != 1)
75                  continue;
76
77              // the current move number
78              numberOfMoves++;
79
80              jumpedOverCoins = 0;
81
82              // making a coin move to form a pair of coins
83              for (int j = i+1; j<n; j++){
84
85                  // move the coin to the column if the condition is satisfied
86                  if(jumpedOverCoins == numberOfMoves){
87                      rowOfCoins[i] = 0;
88                      rowOfCoins[j] += 1;
89                      break;
90                  }
91
92                  // or jump over the column and count the jumped over coins
93                  jumpedOverCoins += rowOfCoins[j];
94
95              }
96
97          }
98
99      }
100     ////////////////////////////////////////////////////////////////////////////////
```

5.2 Algorithm Using Brute-Force Technique

```
8   ////////////////////////////////////////////////////////////////////////////////
9   // number of coins
10  // n = 2^(i+3)-4  where i is non-negative integer [0,...]
11  // i.e. (i=0, n=4), (i=1, n=12), (i=2, n=28), ...
12  int n = 28;
13
14  // there's a row of coins
15  // 0 in a column -> empty column
16  // 1 in a column -> one coin
17  // 2 in a column -> pair of coins
18  int[] rowOfCoins = new int[n];
19
20  // number of moves that are going to be done to form the pairs
21  int numberOfMoves = 0;
22
23  // number of coins that a coin will jump over in a certain move
24  // it should be equal to the number of the move being made
25  private int jumpedOverCoins;
26  ////////////////////////////////////////////////////////////////////////////////
27
28
29
30  ////////////////////////////////////////////////////////////////////////////////
31  // Constructor
32  public JumpingCoins(){
33
34      // each column in the row has just one coin at first
35      for (int i = 0; i < n; i++)
36          rowOfCoins[i]=1;
37
38  }
39  ////////////////////////////////////////////////////////////////////////////////
```

```
104         ////////////////////////////////////////////////////////////////////////////////
105         // Solution using brute force technique
106         public boolean algorithmUsingBruteForce(int position, int currentMoveNumber, String direction){
107
108             if ((position < 0) || (position > n)){
109                 return false;
110             }
111
112             numberOfMoves = currentMoveNumber;
113             jumpedOverCoins = 0;
114             boolean doneFlag = false;
115
116             // check in the right direction the possibility to form a pair
117             if (direction.equals(anObject: "right")){
118
119                 for(int i = position+1; i < n; i++){
120
121                     // move the coin to the column if the condition is satisfied
122                     // or break if exceeded
123                     if(jumpedOverCoins >= numberOfMoves){
124                         if (jumpedOverCoins == numberOfMoves){
125                             rowOfCoins[position] = 0;
126                             rowOfCoins[i] += 1;
127                             currentMoveNumber++;
128                             doneFlag = true;
129                         }
130                         break;
131                     }
132
133                     // or jump over the column and count the jumped over coins
134                     else if(jumpedOverCoins < numberOfMoves)
135                         jumpedOverCoins += rowOfCoins[i];
136
137                 }
138
139             }
```

```
140         // check in the left direction the possibility to form a pair
141         else if (direction.equals(anObject: "left")){
142
143             for(int i = position-1; i >= 0; i--){
144
145                 // move the coin to the column if the condition is satisfied
146                 // or break if exceeded
147                 if(jumpedOverCoins >= numberOfMoves){
148                     if (jumpedOverCoins == numberOfMoves){
149                         rowOfCoins[position] = 0;
150                         rowOfCoins[i] += 1;
151                         currentMoveNumber++;
152                         doneFlag = true;
153                     }
154                     break;
155                 }
156
157                 // or jump over the column and count the jumped over coins
158                 else if(jumpedOverCoins < numberOfMoves)
159                     jumpedOverCoins += rowOfCoins[i];
160
161             }
162
163         }
164
165         // check if there're still single coins
166         for(int i : rowOfCoins){
167             if (i == 1){
168                 doneFlag = false;
169                 break;
170             }
171         }
172
173         // check if all are the coins are pairs
174         // if so, then we are done
175         if(doneFlag == true)
176             return true;
177
```

```
178             // next single coin to be moved to form a pair
179             int nextPosition = position;
180             for(int i = position+1; i < n; i++){
181                 if (rowOfCoins[i] == 1){
182                     nextPosition = i;
183                     break;
184                 }
185             }
186 
187             doneFlag = algorithmUsingBruteForce(position: nextPosition, currentMoveNumber, direction: "right");
188 
189             if(doneFlag == true)
190                 return true;
191 
192             doneFlag = algorithmUsingBruteForce(position: nextPosition, currentMoveNumber, direction: "left");
193 
194             return doneFlag;
195 
196         }
197         ////////////////////////////////////////////////////////////////////////////////
```

6. Output samples

```
// number of coins
// n = 2^(i+3)-4  where i is non-negative integer [0,...]
// i.e. (i=0, n=4), (i=1, n=12), (i=2, n=28), ...
int n = 4;

// there's a row of coins
// 0 in a column -> empty column
// 1 in a column -> one coin
// 2 in a column -> pair of coins
int[] rowOfCoins = new int[n];

// number of moves that are going to be done to form the pairs
int numberOfMoves = 0;
```

Where n = 4

```
1  1  1  1
Number of Moves = 0

Using Greedy Technique:

0  0  2  2
Number of Moves = 2

----------------------------------------

1  1  1  1
Number of Moves = 0

Using Brute Force Technique:

0  0  2  2
Number of Moves = 2
------------------------------------------------
BUILD SUCCESS
```

Where n = 12

```
1  1  1  1  1  1  1  1  1  1  1  1
Number of Moves = 0

Using Greedy Technique:

0  0  2  2  0  0  0  0  2  2  2  2
Number of Moves = 6

----------------------------------------

1  1  1  1  1  1  1  1  1  1  1  1
Number of Moves = 0

Using Brute Force Technique:

0  0  2  2  0  0  0  0  2  2  2  2
Number of Moves = 6
------------------------------------------------
BUILD SUCCESS
```

Where n = 28

```
1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
Number of Moves = 0

Using Greedy Technique:

0  0  2  2  0  0  0  0  2  2  2  2  0  0  0  0  0  0  0  0  2  2  2  2  2  2  2  2
Number of Moves = 14

-----------------------------------------

1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
Number of Moves = 0

Using Brute Force Technique:

0  0  2  2  0  0  0  0  2  2  2  2  0  0  0  0  0  0  0  0  2  2  2  2  2  2  2  2
Number of Moves = 14
------------------------------------------------------------------------
BUILD SUCCESS
```

Where n = 60

```
1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
Number of Moves = 0

Using Greedy Technique:

0  0  2  2  0  0  0  0  2  2  2  2  0  0  0  0  0  0  0  0  2  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
Number of Moves = 30

-----------------------------------------

1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
Number of Moves = 0

Using Brute Force Technique:

0  0  2  2  0  0  0  0  2  2  2  2  0  0  0  0  0  0  0  0  2  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
Number of Moves = 30
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Where n = 124

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Number of Moves = 0

Using Greedy Technique:

0 0 2 2 0 0 0 0 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Number of Moves = 62

-----------------------------------------

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Number of Moves = 0

Using Brute Force Technique:

0 0 2 2 0 0 0 0 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Number of Moves = 62
------------------------------------------------------------------------
BUILD SUCCESS
```

7. Conclusion

There might be in some cases, two different techniques that could produce two different algorithms each one of them has its own way of finding the solution to the problem but due to the problem nature, implementation, assumptions and so on, they could end up with the same time complexity, like in our case in the discussed problem in task 5.

In our case we saw the problem solved by two different techniques the first one is the main one that was required which is the greedy technique which produced a solution that was upper-bounded by O(n^2) in all cases, and the second technique was the brute-force technique which could have led to a much higher running time but as we said due to the assumptions, implementation and the problem nature that helped in our case, the algorithm based on the brute-force technique produced a solution that was upper-bounded by the same running time as the algorithm based on the greedy technique which is O(n^2).