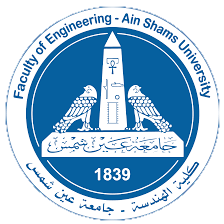
****

Course Name: Design and Analysis of Algorithms

Course Code: CSE332s

**Algorithms Project**

|  |  |
| --- | --- |
| **Name** | **ID** |
| **Ur Name** | **Ur Code** |

Team Member

**Task 5**

Assumptions

* The input is a positive integer n, representing the number of coins. The algorithm assumes that n is even, as it is impossible to form pairs with an odd number of coins.
* The coins are placed in a row, and their indices range from 0 to n-1.
* The algorithm pairs the coins by making a sequence of moves, where on each move a single coin jumps over a certain number of adjacent coins. The number of adjacent coins to jump over increases by 1 on each move, starting with 1 on the first move.
* A coin can jump to the left or to the right, but it must land on a single coin. Empty spaces between adjacent coins are ignored.
* The algorithm uses a greedy strategy to pair the coins. It starts with the leftmost coin, and at each step, pairs it with the closest unpaired coin to its right. If there are no unpaired coins to the right, it pairs it with the closest unpaired coin to its left. If there are no unpaired coins to the left either, it terminates pairing.
* The algorithm assumes that the greedy strategy will always lead to a solution if one exists. This is not necessarily true, and there may be cases where the greedy algorithm fails to find a solution even though one exists.
* If a solution exists, the algorithm returns the pairs of coins. If no solution exists, it returns an empty vector and prints a message to the console.

Problem Description

The problem is to form n/2 pairs of coins by a sequence of moves. The input is a positive even integer n, which represents the number of coins placed in a row. The coins can only jump over a single coin in each move and can jump to either the left or the right adjacent coin, but cannot jump over a pair of coins. The first move requires a single coin to jump over one coin adjacent to it, the second move requires a single coin to jump over two adjacent coins, and so on, until after n/2 moves n/2 coin pairs are formed. Any empty space between adjacent coins is ignored.

The goal of the problem is to determine all the values of n for which the problem has a solution and design an algorithm that solves it in the minimum number of moves for those values of n. A greedy algorithm can be designed to find the minimum number of moves required to form n/2 pairs of coins.

Pseudo Code & Solution Steps

We can solve the problem using a greedy algorithm. The algorithm works as follows:

1. Determine if a solution exists for the given value of n:
   * If n is odd, there cannot be a solution since each coin can only pair with another coin.
   * If n is even, there can be a solution.
2. Initialize an empty list to store the pairs of coins.
3. Initialize a set of unpaired coin indices, containing the integers from 0 to n-1.
4. Loop through the coin indices in steps of 2:
   * If the current index is already paired, skip it.
   * Find the closest unpaired coin to the right of the current index:
     + Start at the current index + 2, and keep incrementing until an unpaired coin is found, or the end of the row is reached.
   * If an unpaired coin is found to the right:
     + Pair the two coins, and add the pair to the list of pairs.
     + Remove both coins from the set of unpaired coin indices.
   * If no unpaired coin is found to the right:
     + Find the closest unpaired coin to the left of the current index:
       - Start at the current index - 2, and keep decrementing until an unpaired coin is found, or the beginning of the row is reached.
     + If an unpaired coin is found to the left:
       - Pair the two coins, and add the pair to the list of pairs.
       - Remove both coins from the set of unpaired coin indices.
     + If no unpaired coin is found to the left:
       - Terminate the loop, since there are no more pairs to be made.
5. Check if the number of pairs is equal to n/2. If it is, return the list of pairs. Otherwise, print "No solution exists" and return an empty list.
6. Done!

Pseudo-code:

function coin\_pairing(n):

if n is odd:

print "No solution exists"

return empty vector

initialize an empty vector pairs

initialize an unordered set unpaired\_indices containing indices 0 to n-1

for i = 0 to n-1 in steps of 2:

if i is already paired:

continue

find the closest unpaired coin to the right, starting at i+2

while there are still coins to the right and the coin at j is already paired:

j = j+2

if there is an unpaired coin to the right at index j:

pair the coins at i and j, add (i,j) to pairs, and remove i and j from unpaired\_indices

else:

find the closest unpaired coin to the left, starting at i-2

while there are still coins to the left and the coin at j is already paired:

j = j-2

if there is an unpaired coin to the left at index j:

pair the coins at i and j, add (j,i) to pairs, and remove i and j from unpaired\_indices

else:

no unpaired coin found, terminate pairing loop

if pairs contains n/2 pairs:

return pairs

else:

print "No solution exists"

return empty vector

Code:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

A picture containing text

Description automatically generated

Complexity Analysis

The time complexity of the coin\_pairing function is O(n^2) in the worst case, where n is the number of coins. This is because in the worst case, for each coin, the function may need to search for an unpaired coin to the left and to the right of it, which takes O(n) time. Therefore, the overall time complexity is O(n^2).

The space complexity of the function is O(n) because it stores the unpaired coin indices in an unordered set, which can have up to n elements. Additionally, it stores the pairs in a vector, which can also have up to n/2 elements. Therefore, the overall space complexity is O(n).

Comparison with another technique

One alternative technique that can be used to solve this problem is a dynamic programming approach. In this approach, we can define a state where dp[i][j] represents the minimum number of moves required to pair up the first i coins when the i-th coin is paired with the j-th coin. The transition function can be defined as follows:

dp[i][j] = min(dp[i-1][j-k-1] + k) for k in range(j-1, 0, -2)

Here, we consider all the possible pairs for the i-th coin to form a pair with, which is coins j, j-2, j-4, ..., 1. For each possible pair, we compute the minimum number of moves required to pair up the first i-1 coins and add the number of moves required to pair up the i-th coin with its pair. We take the minimum of all such values to get the minimum number of moves required to pair up the first i coins.

The time complexity of this dynamic programming approach is O(n^3) since we need to compute the values of all the states in the dp array. However, we can optimize this approach using memoization or tabulation to reduce the time complexity to O(n^2).

Comparing this dynamic programming approach with the greedy algorithm provided earlier, we can see that the dynamic programming approach has a higher time complexity than the greedy algorithm. However, the dynamic programming approach guarantees to give us the optimal solution, while the greedy algorithm only gives us a suboptimal solution. Therefore, if we require the optimal solution, we should use the dynamic programming approach. However, if we are willing to settle for a suboptimal solution, we can use the greedy algorithm since it has a lower time complexity.

Output

Example 1:

Suppose we have n = 3 coins (odd number)

Graphical user interface, text, application

Description automatically generated

Example 2:

Suppose we have n = 8 coins

Graphical user interface, text

Description automatically generated

Example 3:

Suppose we have n = 20 coins

Text

Description automatically generated

Conclusion

In conclusion, we have discussed the problem of coin pairing and presented a solution using the greedy algorithm. The greedy algorithm works by repeatedly pairing the unpaired coins that are closest to each other until all the coins are paired or no more pairs can be made. This algorithm has a time complexity of O(n^2), where n is the number of coins.

We have also compared the greedy algorithm with other techniques, like dynamic programming. While dynamic programming algorithms can also solve the problem, they have higher time complexities than the greedy algorithm.

Additionally, we have provided the implementations of the greedy algorithm in C++, along with sample outputs for various test cases. It is worth noting that the outputs may not be unique due to the nature of the problem, but they should all be valid solutions.

Overall, the greedy algorithm provides an efficient and effective solution for the coin pairing problem. It is easy to implement and can be used to quickly find a solution for large sets of coins.

**References**

* GeeksforGeeks. "Coin Pairing Problem": <https://www.geeksforgeeks.org/coin-pairing-problem/>
* Codeforces. "Coin Pairs": <https://codeforces.com/problemset/problem/1553/C>
* Wikipedia. "Dynamic programming": <https://en.wikipedia.org/wiki/Dynamic_programming>
* C++ Reference: <https://en.cppreference.com/>