CSE431 HW2

Siqi Cheng, 50388579

Problem 1:

(1):

```
      1
     / \
    2   3
   /   / \
  4   5   6
```

(2):

```
Function DFS(T, v, parent, size):
    size[v] ← 1
    For each neighbor w of v not equal to parent:
        DFS(T, w, v, size)
        size[v] += size[w]
Function BalancedCut(T):
    n ← len(T)             // Get the number of nodes in the tree
    size ← array of size n filled with zeros
    DFS(T, 0, -1, size)    // Start DFS from node 0

    For each node v and its neighbor w in T:
       diff ← abs(n - 2*size[w])
       If diff is minimum so far, store (v, w) as balanced_edge
return balanced_edge
```

DFS calculates the size of all subtrees correctly. The algorithm checks every connection (edge) and calculates the difference in node counts if we cut the tree at this edge. Algorithm will stop because each tree node is visited once. The algorithm finds the edge that, when cut, results in the smallest difference in node counts. The algorithm keeps track of which connection, when cut, would lead to the smallest difference in node numbers between the two resulting parts. That's the balanced cut the algorithm is looking for, and the edge is the balanced cut we're looking for.

Problem 2:

(1):

```
v1--v2
  |   |
v6--v3
  |   |
v5--v4
```

Edges: {(v1, v2), (v2, v3), (v3, v4), (v4, v5), (v5, v6), (v6, v1), (v1, v3), (v3, v5), (v5, v1)}

The set of joints could be: {(v1, v2, v3), (v3, v4, v5), (v5, v6, v1)}

(2)

```
 v1--v2
  |   |
 v4--v3
```

Edges: {(v1, v2), (v2, v3), (v3, v4), (v4, v1)}

In this case, there is no way to form joints that include each vertex exactly once. If we select any three vertices to form a joint, the remaining vertex would not be able to form a joint.

(3)

```
def dfs(node, parent, graph, count):
    count[node] = 1
    for child in graph[node]:
        if child != parent:
            dfs(child, node, graph, count)
            count[node] += count[child]
    return count[node] % 3 == 0


def joint_match(n, edges):
    if n % 3 != 0:
        return False


    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    count = [0] * n
    return dfs(0, -1, graph, count)
```

The algorithm is based on DFS, if a tree can be joint-matched, then it must have several vertices that are divisible by 3. Hence, if n % 3! = 0, the tree can't be joint matched, and return False. Then use DFS to compute the size of the subtree rooted at each node.

count -> count[node] = 1 + count[child_1] + count[child_2] + ... + count[child_k].

If count[node] % 3 == 0 for a node, there are enough vertices to form a certain number of joints in the subtree rooted at this node. Then reset this subtree (set count[node] = 0) because it has been matched. At the end, count[root] must be = 0, so return the result: count[root] % 3 == 0.

Problem 3

(1):

Calculate distance = H[i+1] - H[i] for each platform i, where H[i] is the height of platform i.

Sort the platforms by distance in ascending order.

```
 Initialize n = len(H), numJumps = 0  currentPlatform = 0
  while currentPlatform < n - 1:
    max_range = H[currentPlatform] + max_jump
    farthestPlatform = currentPlatform

    for i in range(currentPlatform + 1, n):
      if H[i] > max_range:
        break
      if H[i] <= max_range:
        farthestPlatform = i

    currentPlatform = farthestPlatform
    numJumps += 1
  return numJumps
```

Proving the GCP: By definition of GCP, if optimum and it is one achieved by our mapping, then it is optimum,

By contradiction, if optimum and it is not achieved by our mapping, then it is not optimum. assume that there exists an optimal solution S that doesn't follow our algorithm's mapping, meaning there's at least one point where it doesn't jump to the farthest reachable platform. Let F be the farthest platform that could have been reached from a given platform, and let N be the platform that was chosen in the optimal solution S instead of F, where N is closer than F.

When we replace the jump to N with a jump to F in S, the total number of jumps decreases, and the last platform remains reachable (since both N and F are reachable from the current platform).

Hence, we can obtain a solution with fewer jumps than 0, which contradicts the assumption that S is an optimal solution. Therefore, the optimal solution must involve always jumping to the farthest reachable platform, validating our algorithm's mapping and GCP.

Proving the OSP: By definition: it shows that if a solution is optimal, then the induced subproblem is also optimal. We want to demonstrate that if we have an optimal solution I to the problem of P[i], then I \ {i} is the optimal solution to P[i-1].

By contradiction, Let I be the optimal solution with J jumps to problem P[i], and let X be the final platform reached. Let I' = I \ {i} be a solution to P[i-1] with J' = J - 1 jumps and the final platform reached being X'. Assume I' is not optimal for P[i-1] and that there is another solution I'' to P[i-1] that requires fewer jumps J'' < J'. Then, I'', U {i} is a solution to P[i] with fewer jumps J'' + 1 < J' + 1 = J. This means I is not an optimal solution, contradicting our assumption.

Therefore, I' must be optimal for P[i-1], confirming our OSP.

(2)

```
def min_bounces(H, S):
    n = len(H)
    m = len(S)
    dp = [float('inf')] * n
    dp[0] = 0
    for i in range(1, n):
        if H[i] in S:
            last_spring_idx = max([j for j in range(i) if H[j] in S and H[j] < H[i]], default=-1)
            if last_spring_idx != -1:
                dp[i] = dp[last_spring_idx]
        else:
            last_spring_idx = max([j for j in range(i) if H[j] in S and H[j] < S[i]], default=-1)
            if last_spring_idx != -1:
                dp[i] = dp[last_spring_idx] + 1
    return dp[n-1]
```

Proving the GCP: By definition of GCP, if optimum and it is one achieved by our mapping, then it is optimum,

Assume there exists an optimal solution O that doesn't follow the greedy approach of always choosing the last spring platform before a regular platform or the last spring platform before another spring platform.

Let i be the first platform in O where the greedy approach deviates, and let p be the platform chosen in O instead of the last spring platform before i. Replacing p with the last spring platform before i yields a new solution O'. However, O' has an equal or fewer number of bounces and reaches the same final platform as O. Thus, O' contradicts the assumption that O is optimal.

Proving the OSP: By definition: it shows that if a solution is optimal, then the induced subproblem is also optimal. We want to demonstrate that if we have an optimal solution I to the problem of P[i], then I \ {i} is the optimal solution to P[i-1].

By contradiction, Let I be the optimal solution with J jumps to problem P[i], and let X be the final platform reached. Let I' = I \ {i} be a solution to P[i-1] with J' = J - 1 jumps and the final platform reached being X'. Assume I' is not optimal for P[i-1] and that there is another solution I'' to P[i-1] that requires fewer jumps J'' < J'. Then, I'', U {i} is a solution to P[i] with fewer jumps J'' + 1 < J' + 1 = J. This means I is not an optimal solution, contradicting our assumption.

Therefore, I' must be optimal for P[i-1], confirming our OSP.

Problem 4

(1)

Total number of counts is :5

100101, 100101, 100101, 100101, 100101

(2)

The algorithm recursively divides the binary string into two halves until the base case is reached, then it combines the results of the left and right halves, checking for any occurrences that cross the dividing line.

It has a time complexity of O(n log n), where n is the length of the binary string, this is because the algorithm recursively divides the string in half until it reaches the base. Each level of the tree operates on a total of n characters, and there are log n levels in the tree.

Problem 5

(1):

1 0 1

0 1 0

1 0 1

(2):

1 1 0

0 1 1

1 0 0

(3):

```
 function issubmatrix(arr, top, bottom, left, right) {
     if (right-left == 1 and bottom-top == 1):
         return (arr[top][left] == 1 and arr[top][right] == 0 and arr[bottom][left] == 0 and
arr[bottom][right] == 1)
     // Divide the matrix in four quadrants and recursively check each
     midRow = (top+bottom)/2
     midCol = (left+right)/2
     topLeftSub = issubmatrix(arr, top, midRow, left, midCol)
     topRightSub = issubmatrix(arr, top, midRow, midCol+1, right)
     bottomLeftSub = issubmatrix(arr, midRow+1, bottom, left, midCol)
     bottomRightSub = issubmatrix(arr, midRow+1, bottom, midCol+1, right)
     return topLeftSub or topRightSub or bottomLeftSub or bottomRightSub
 }
```

The algorithm checks all 2x2 submatrices in the given matrix for the diagonal submatrix. If it finds one, returns True, If all of them return False, we know that the original matrix does not contain a diagonal submatrix, so we return False. so it must find a diagonal submatrix if there is one.

The recursion depth is logarithmic in the size of the matrix, denoted as O(log(n)), where n is the dimension of the matrix.