In [1]:
```
# a python implementation and solution of the breakthrough problem
import numpy as np

# we start by implementing a binary tree

# a simple binary tree node
class Node:
    def __init__(self, n):
        self.parent = None
        self.left = None
        self.right = None
        self.n = n

        # attributes indicating the arc to the respective child node i
s free/blocked
        self.block_left = False
        self.block_right = False

    # return true if the node is the root node
    def is_root(self):
        return True if self.parent is None else False

    def is_leaf(self):
        return True if self.left is None and self.right is None else F
alse


class Tree:
    # function to initialize a binary tree
    # n_stages denotes the depth of the binary tree
    #(e.g. 3 stages means 7 nodes in total)

    def __init__(self, k):
        self.n_stages = k
        self.tree = [] # initialize an array to store the binary tree
        current_node = 1 # keep track of the number of nodes assigned

        for k in range(self.n_stages):
            n_nodes = 2 ** k # 2^k nodes at depth k
            if n_nodes == 1:
                self.tree.append(Node(current_node)) # this is the roo
t node
                current_node += 1
            else:
                for _ in range(n_nodes):
                    node = Node(current_node)
                    self.tree.append(node)
```

```
                    # update parent node
                    # # the idea is a child node's parent node is alwa
ys floor(current_node / 2)
                    parent = self.tree[current_node // 2 - 1]
                    node.parent = parent

                    if current_node % 2 == 0:
                        parent.left = node
                    else:
                        parent.right = node

                    current_node += 1


    # utility function used to print the tree
    def info(self, pr=False):
        info = []
        for node in self.tree:
            pl = []  # a list of info to print()
            pl.append(f"node: {node.n}")
            pl.append(f"lf_node: {node.left.n if node.left is not None
else None}")
            pl.append(f"re_node: {node.right.n if node.right is not No
ne else None}")
            pl.append(f"parent: {node.parent.n if node.parent is not N
one else None}")
            #pl.append(f"root: {node.is_root()}")
            pl.append(f"leaf: {node.is_leaf()}")
            pl.append(f"lf_block: {node.block_left}")
            pl.append(f"rt_block: {node.block_right}")
            info.append(pl)

        if pr: # print info
            for i in info:
                print(i)
        return info


    # this function creates a random path (solution) from the root to
a leaf node
    def random_path(self):
        directions = ['left','right']
        current_node = self.tree[0] # root node
        self.path = [current_node.n] # initialize the path from the ro
ot
        self.actions = [] # use this array to store the actions to tak
e to go to the leaf node

        while current_node.is_leaf() is False:
            choice = np.random.choice(directions)
```

```
            current_node = current_node.left if choice == 'left' else
current_node.right
            self.path.append(current_node.n)
            self.actions.append(choice)

        return self.path, self.actions


    # this function trys to traverse back from any node to the root no
de
    # returns true if the node can be reached from root and vice versa
    def check_free_path(self, num_node):
        node = self.tree[num_node-1]
        parent = node.parent
        while parent is not None:
            if node is parent.left and parent.block_left:
                return False
            if node is parent.right and parent.block_right:
                return False
            node = parent
            parent = node.parent

        return True


    # this function creates the breakthrough problem
    # by randomly blocking arcs (with a certain probability)
    # so that only the assigned path is allowed to be traversed
    def create_puzzle(self, p=0.5):
        # first block all arcs in the tree
        for node in self.tree:
            if not node.is_leaf():
                node.block_left = True
                node.block_right = True

        # clear all blocks in the random path to create the solution
        assert(self.path is not None)
        assert(self.actions is not None)

        for i, num_node in enumerate(self.path[:-1]):
            # for the other arc
            actions = ['block', 'free']
            action = np.random.choice(actions, p=[p, 1.0-p])

            if self.actions[i] == 'left':
                self.tree[num_node-1].block_left = False
                # the other arc has chance p to be freed
                if action == 'free':
                    self.tree[num_node-1].block_right = False
            else: # do the same
```

```
                self.tree[num_node-1].block_right = False
                if action == 'free':
                    self.tree[num_node-1].block_left = False

        # the rest of the blocked arcs have chance = p to be freed
        actions = ['block', 'free']
        for node in self.tree:
            if not node.n in self.path:
                for direction in ['left', 'right']:
                    action = np.random.choice(actions, p=[p, 1.0-p])
                    if action == 'free' and direction == 'left':
                        node.block_left = False
                    elif action == 'free' and direction == 'right':
                        node.block_right = False

        # check if there's any accidentally reopened path from leaves to root
        # if so block the arc to the respective leaf node
        checklist = [node.n for node in self.tree[len(self.tree)//2:] if node.n != self.path[-1]]

        while checklist:
            checklist = [n for n in checklist if self.check_free_path(n)]

            for i in checklist:
                node = self.tree[i-1]
                parent = node.parent
                if node is parent.left:
                    parent.block_left = True
                if node is parent.right:
                    parent.block_right = True
```

In [2]:
```
# now we are implementing an optimal solution for the problem

def backward_recursion(tree):
    # this is a tweaked version of Tree.check_free_path()
    # instead this function returns the number of steps for finding su
ch path
    # and all the steps in the path
    n_nodes = len(tree.tree)
    step_count = 0

    start_node = n_nodes // 2
    end_node = n_nodes

    solutions = {k: [] for k in range(n_nodes+1)} # store solutions fo
r each node in the tree
    for node in tree.tree[start_node:end_node]:
        solutions[node.n].append(node.n)

    current_stage = tree.n_stages - 1 # set current stage to the secon
d last stage N-1 in the tree (depth N-1)

    while current_stage >= 1:
        end_node = start_node
        start_node = start_node // 2
        nodes = tree.tree[start_node:end_node]
        # for every node the current stage
        for node in nodes:
            if node.block_left == False and solutions[node.left.n]:
                step_count += 1 # +1 count for looking left
                solutions[node.n].append(node.n)
                solutions[node.n].append(solutions[node.left.n])
            if node.block_right == False and solutions[node.right.n]:
                step_count += 1 # +1 count for looking right
                solutions[node.n].append(node.n)
                solutions[node.n].append(solutions[node.right.n])

        #candidates[current_stage] = new_candidates
        #candidate_nodes = new_candidates
        current_stage -= 1

    return solutions, step_count
```

In [23]:
```python
# now we look into a heuristic solution:
# the basic idea is to start from the root node
# explore the two child nodes and select the right child if both child
nodes are free
# otherwise explore via availble arc
# keep exploring till a leaf node is reached, if ever possible
def heuristic(tree, start_node=None):
    step_count = 0
    if start_node is None:
        node = tree.tree[0]
    else:
        node = start_node
    path = [node.n]

    while not node.is_leaf():
        if not node.block_right:
            step_count += 1
            node = node.right
            path.append(node.n)
        elif not node.block_left:
            step_count += 1
            node = node.left
            path.append(node.n)
        else:
            break

    return path, step_count
```

In [29]:

```
# the rollout algorithm
# if both arcs are available, explore both using the heuristic
# else go with the available arc
def rollout(tree):
    step_count = 0
    node = tree.tree[0] # start from the root note
    path = [node.n]

    while not node.is_leaf():
        # consider four cases:
        if not node.block_right and not node.block_left:
            r_path, r_count = heuristic(tree, start_node=node.right)
            l_path, l_count = heuristic(tree, start_node=node.left)
            step_count += r_count + l_count

            if tree.tree[r_path[-1]-1].is_leaf():
                path.extend(r_path)
                return path, step_count
            elif tree.tree[l_path[-1]-1].is_leaf():
                path.extend(l_path)
                return path, step_count
            else:
                step_count += 1
                node = node.right
                path.append(node.n)

        elif not node.block_right:
            step_count += 1
            node = node.right
            path.append(node.n)

        elif not node.block_left:
            step_count += 1
            node = node.left
            path.append(node.n)

        else:
            break

    return path, step_count
```

In [46]:
```
# test
depth = 15
t1 = Tree(depth) # initialize a tree of depth 10
path, actions = t1.random_path() # create a random solution
print("created path: ", path)
print("directions (root to leaf): ", actions)
t1.create_puzzle() # create a puzzle. the probability of blocking an a
rc is p=0.5 by default
#info = t1.info(pr=True)
print("-" * 100)

solutions, count = backward_recursion(t1)
print("solution found by the backward recursion: ", solutions[1])
print("number of looks in the backward recursion: ", count)
print("-" * 100)

h_path, h_count = heuristic(t1)
print("solution found by the heuristic algorithm: ", h_path)
print("number of looks in the heuristic algorithm: ", h_count)
print("-" * 100)

roll_path, roll_count = rollout(t1)
print("solution found by the rollout algorithm: ", roll_path)
print("number of looks in the rollout algorithm: ", roll_count)
```

```
created path:  [1, 2, 5, 10, 21, 43, 86, 172, 345, 691, 1383, 2766,
5533, 11067, 22135]
directions (root to leaf):  ['left', 'right', 'left', 'right', 'righ
t', 'left', 'left', 'right', 'right', 'right', 'left', 'right', 'rig
ht', 'right']
----------------------------------------------------------------------
------------------------------
solution found by the backward recursion:  [1, [2, [5, [10, [21, [43
, [86, [172, [345, [691, [1383, [2766, [5533, [11067, [22135]]]]]]]]
]]]]]]
number of looks in the backward recursion:  13400
----------------------------------------------------------------------
------------------------------
solution found by the heuristic algorithm:  [1, 2, 5, 11, 22, 45, 90
, 180, 360, 721, 1442, 2885, 5770]
number of looks in the heuristic algorithm:  12
----------------------------------------------------------------------
------------------------------
solution found by the rollout algorithm:  [1, 2, 5, 10, 21, 43, 86,
172, 345, 691, 1383, 2766, 5533, 11067, 22135]
number of looks in the rollout algorithm:  22
```

In [48]:
```
# test 2
depth = 15
t1 = Tree(depth) # initialize a tree of depth 10
path, actions = t1.random_path() # create a random solution
print("created path: ", path)
print("directions (root to leaf): ", actions)
t1.create_puzzle() # create a puzzle. the probability of blocking an a
rc is p=0.5 by default
#info = t1.info(pr=True)
print("-" * 100)

solutions, count = backward_recursion(t1)
print("solution found by the backward recursion: ", solutions[1])
print("number of looks in the backward recursion: ", count)
print("-" * 100)

h_path, h_count = heuristic(t1)
print("solution found by the heuristic algorithm: ", h_path)
print("number of looks in the heuristic algorithm: ", h_count)
print("-" * 100)

roll_path, roll_count = rollout(t1)
print("solution found by the rollout algorithm: ", roll_path)
print("number of looks in the rollout algorithm: ", roll_count)
```

```
created path:  [1, 3, 7, 14, 28, 57, 115, 230, 461, 923, 1847, 3695,
7390, 14781, 29562]
directions (root to leaf):  ['right', 'right', 'left', 'left', 'righ
t', 'right', 'left', 'right', 'right', 'right', 'right', 'left', 'ri
ght', 'left']
--------------------------------------------------------------------
--------------------------------
solution found by the backward recursion:  [1, [3, [7, [14, [28, [57
, [115, [230, [461, [923, [1847, [3695, [7390, [14781, [29562]]]]]]]
]]]]]]]
number of looks in the backward recursion:  13309
--------------------------------------------------------------------
--------------------------------
solution found by the heuristic algorithm:  [1, 3, 7, 15, 30]
number of looks in the heuristic algorithm:  4
--------------------------------------------------------------------
--------------------------------
solution found by the rollout algorithm:  [1, 3, 7, 14, 28, 57, 115,
230, 461, 923, 1847, 3695, 7390, 14781, 29562]
number of looks in the rollout algorithm:  14
```

In [53]:
```
# test 3
depth = 10
t1 = Tree(depth) # initialize a tree of depth 10
path, actions = t1.random_path() # create a random solution
print("created path: ", path)
print("directions (root to leaf): ", actions)
t1.create_puzzle() # create a puzzle. the probability of blocking an a
rc is p=0.5 by default
#info = t1.info(pr=True)
print("-" * 100)

solutions, count = backward_recursion(t1)
print("solution found by the backward recursion: ", solutions[1])
print("number of looks in the backward recursion: ", count)
print("-" * 100)

h_path, h_count = heuristic(t1)
print("solution found by the heuristic algorithm: ", h_path)
print("number of looks in the heuristic algorithm: ", h_count)
print("-" * 100)

roll_path, roll_count = rollout(t1)
print("solution found by the rollout algorithm: ", roll_path)
print("number of looks in the rollout algorithm: ", roll_count)
```

```
created path:  [1, 2, 5, 10, 20, 41, 82, 164, 329, 659]
directions (root to leaf):  ['left', 'right', 'left', 'left', 'right
', 'left', 'left', 'right', 'right']
----------------------------------------------------------------------
------------------------------
solution found by the backward recursion:  [1, [2, [5, [10, [20, [41
, [82, [164, [329, [659]]]]]]]]]]
number of looks in the backward recursion:  425
----------------------------------------------------------------------
------------------------------
solution found by the heuristic algorithm:  [1, 2, 5, 10, 20, 41, 83
, 167]
number of looks in the heuristic algorithm:  7
----------------------------------------------------------------------
------------------------------
solution found by the rollout algorithm:  [1, 2, 5, 10, 20, 41, 82,
164, 329, 659]
number of looks in the rollout algorithm:  17
```

In [54]:
```
# test 4
depth = 10
t1 = Tree(depth) # initialize a tree of depth 10
path, actions = t1.random_path() # create a random solution
print("created path: ", path)
print("directions (root to leaf): ", actions)
t1.create_puzzle(p=0.8) # create a puzzle. the probability of blocking
an arc is p=0.5 by default
#info = t1.info(pr=True)
print("-" * 100)

solutions, count = backward_recursion(t1)
print("solution found by the backward recursion: ", solutions[1])
print("number of looks in the backward recursion: ", count)
print("-" * 100)

h_path, h_count = heuristic(t1)
print("solution found by the heuristic algorithm: ", h_path)
print("number of looks in the heuristic algorithm: ", h_count)
print("-" * 100)

roll_path, roll_count = rollout(t1)
print("solution found by the rollout algorithm: ", roll_path)
print("number of looks in the rollout algorithm: ", roll_count)
```

```
created path:  [1, 2, 4, 9, 18, 37, 74, 148, 296, 592]
directions (root to leaf):  ['left', 'left', 'right', 'left', 'right
', 'left', 'left', 'left', 'left']
----------------------------------------------------------------------
------------------------------
solution found by the backward recursion:  [1, [2, [4, [9, [18, [37,
[74, [148, [296, [592]]]]]]]]]]
number of looks in the backward recursion:  154
----------------------------------------------------------------------
------------------------------
solution found by the heuristic algorithm:  [1, 3]
number of looks in the heuristic algorithm:  1
----------------------------------------------------------------------
------------------------------
solution found by the rollout algorithm:  [1, 2, 4, 9, 18, 37, 74, 1
48, 296, 592]
number of looks in the rollout algorithm:  8
```

In [55]:

```
# test 5
depth = 10
t1 = Tree(depth) # initialize a tree of depth 10
path, actions = t1.random_path() # create a random solution
print("created path: ", path)
print("directions (root to leaf): ", actions)
t1.create_puzzle(p=0.8) # create a puzzle. the probability of blocking
an arc is p=0.5 by default
#info = t1.info(pr=True)
print("-" * 100)

solutions, count = backward_recursion(t1)
print("solution found by the backward recursion: ", solutions[1])
print("number of looks in the backward recursion: ", count)
print("-" * 100)

h_path, h_count = heuristic(t1)
print("solution found by the heuristic algorithm: ", h_path)
print("number of looks in the heuristic algorithm: ", h_count)
print("-" * 100)

roll_path, roll_count = rollout(t1)
print("solution found by the rollout algorithm: ", roll_path)
print("number of looks in the rollout algorithm: ", roll_count)
```

```
created path:  [1, 2, 5, 10, 21, 43, 87, 174, 348, 696]
directions (root to leaf):  ['left', 'right', 'left', 'right', 'righ
t', 'right', 'left', 'left', 'left']
--------------------------------------------------------------------
--------------------------------
solution found by the backward recursion:  [1, [2, [5, [10, [21, [43
, [87, [174, [348, [696]]]]]]]]]]
number of looks in the backward recursion:  149
--------------------------------------------------------------------
--------------------------------
solution found by the heuristic algorithm:  [1, 2, 5, 11]
number of looks in the heuristic algorithm:  3
--------------------------------------------------------------------
--------------------------------
solution found by the rollout algorithm:  [1, 2, 5, 10, 21, 43, 87,
174, 348, 696]
number of looks in the rollout algorithm:  8
```

In [67]:

```
# test 5
depth = 10
t1 = Tree(depth) # initialize a tree of depth 10
path, actions = t1.random_path() # create a random solution
print("created path: ", path)
print("directions (root to leaf): ", actions)
t1.create_puzzle(p=0.3) # create a puzzle. the probability of blocking an arc is p=0.5 by default
#info = t1.info(pr=True)
print("-" * 100)

solutions, count = backward_recursion(t1)
print("solution found by the backward recursion: ", solutions[1])
print("number of looks in the backward recursion: ", count)
print("-" * 100)

h_path, h_count = heuristic(t1)
print("solution found by the heuristic algorithm: ", h_path)
print("number of looks in the heuristic algorithm: ", h_count)
print("-" * 100)

roll_path, roll_count = rollout(t1)
print("solution found by the rollout algorithm: ", roll_path)
print("number of looks in the rollout algorithm: ", roll_count)
```

```
created path:  [1, 2, 5, 11, 22, 44, 89, 178, 357, 714]
directions (root to leaf):  ['left', 'right', 'right', 'left', 'left
', 'right', 'left', 'right', 'left']
--------------------------------------------------------------------
--------------------------------
solution found by the backward recursion:  [1, [2, [5, [11, [22, [44
, [89, [178, [357, [714]]]]]]]]]]
number of looks in the backward recursion:  483
--------------------------------------------------------------------
--------------------------------
solution found by the heuristic algorithm:  [1, 3, 7, 15, 31, 63, 12
7, 255]
number of looks in the heuristic algorithm:  7
--------------------------------------------------------------------
--------------------------------
solution found by the rollout algorithm:  [1, 3, 7, 15, 31, 63, 127,
255]
number of looks in the rollout algorithm:  50
```

In [ ]: