1. Fibonacci Problem

In [13]:
```
def fibonacci(text):

  #split text into individual integers
  a1, a2, n = map(int, text.split())

  # initialise result variable
  result = 0

  # loop and add until nth position in fibonacci
  for i in range(n-2):

    if i == 0:
      result = a1 + a2

    else:
      a2 = result
      result = result + a1

    a1 = a2

  return result
```

In [15]:
```
print(fibonacci("1 2 3"))
print(fibonacci("7 12 5"))
```

```
3
50
```

2. LRU Cache

In [24]:
```
# the data structure is linked list, where the nodes are stored in a dictionary
class Node:
  def __init__(self, key=None, value=None):
    self.key = key
    self.value = value
    self.prev = None
    self.next = None

class LRUCache:
  def __init__(self, capacity: int):
    self.capacity = capacity
    self.cache = {}
    self.head = Node() #dummy
    self.tail = Node() #dummy
    self.head.next = self.tail
    self.tail.prev = self.head

  def remove(self, node):
    prev_node = node.prev
    next_node = node.next
    prev_node.next = next_node
    next_node.prev = prev_node

  def insert(self, node):
```

```
    # insert after dummy head
    next_node = self.head.next
    self.head.next = node
    node.prev = self.head
    node.next = next_node
    next_node.prev = node

  def get(self, key):
    if key in self.cache:
      node = self.cache[key]
      self.remove(node)
      self.insert(node)  # move to front
      return node.value
    else:
      return -1

  def put(self, key: int, value: int):

    # if key exists, update value and move up priority
    if key in self.cache:
      node = self.cache[key]
      node.value = value
      self.remove(node)
      self.insert(node)
      print("Existing key {} updated with value {}".format(key, value))

    # if new key
    else:
      # check if cache is full
      if len(self.cache) == self.capacity:
          # remove least priority (tail node)
          last_node = self.tail.prev
          self.remove(last_node)
          del self.cache[last_node.key]
          print("Cache is full, removing key {}".format(last_node.key))

      # Insert new node
      new_node = Node(key, value)
      self.cache[key] = new_node
      self.insert(new_node)
      print("Inserting key {} with value {}".format(key, value))
```

In [25]:
```
def LRU(capacity, n, operations):
  cache = LRUCache(capacity)
  for operation in operations:
    if len(operation) == 2:
      key, value = operation
      cache.put(key, value)

    else:
      key = operation[0]
      print("The value of key {} is {}".format(key, cache.get(key)))
```

In [26]:
```
c = 3
n = 11
ops = [[1,1], [2,2], [3,3], [4,5], [3], [1], [4], [2,3], [1], [3], [2]]

LRU(c,n,ops)
```

```
Inserting key 1 with value 1
Inserting key 2 with value 2
Inserting key 3 with value 3
Cache is full, removing key 1
Inserting key 4 with value 5
The value of key 3 is 3
The value of key 1 is -1
The value of key 4 is 5
Existing key 2 updated with value 3
The value of key 1 is -1
The value of key 3 is 3
The value of key 2 is 3
```

3. Route Planning

In [2]:
```
from collections import deque
```

In [56]:
```
def is_valid_path(x, y, map, visited):
  # check if the coods are in the map, is a road and is not visited yet
  rows = len(map)
  cols = len(map[0])

  return 0 <= x < rows and 0 <= y < cols and map[x][y] in ('r', 'e') and visited

def find_shortest_path(map):

  DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

  start = None
  end = None

  # find start and end coords
  for x in range(len(map)):
    for y in range(len(map[x])):
      if map[x][y] == "s":
        start = (x,y)

      if map[x][y] == "e":
        end = (x,y)

  # initialise queue with starting coords
  queue = deque([start])
  # initialise visited table
  visited = [[False for i in range(len(map[0]))] for j in range(len(map))]
  print(visited)
  visited[start[0]][start[1]] = True
  # initialise parent dictionary to store route information (can be improved usi
  parents = {}

  while queue:
    x, y = queue.popleft()

    if (x, y) == end:
      break

    for dx, dy in DIRECTIONS:
      mx, my = dx + x, dy + y
      if is_valid_path(mx, my, map, visited):
        queue.append((mx,my))
```

```
        visited[mx][my] = True
        parents[(mx, my)] = (x, y)

  print((x,y))
  if (x, y) != end:
    return "No path found"

  path = []
  curr = end
  while curr != start:
    path.append(curr)
    curr = parents[curr]

  return path[::-1]
```

In [58]:
```
map = [
    ['s', 'o', 'o', 'o'],
    ['r', 'r', 'r', 'e'],
    ['r', 'o', 'o', 'r'],
    ['r', 'r', 'r', 'r']
]

map2 = [
    ['s', 'o', 'o', 'o'],
    ['r', 'r', 'r', 'r'],
    ['r', 'o', 'o', 'r'],
    ['r', 'r', 'r', 'e']
]

print(find_shortest_path(map2))
```

```
[[False, False, False, False], [False, False, False, False], [False, False, Fals
e, False], [False, False, False, False]]
(3, 3)
[(1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3)]
```

4. Graphical Editor

In [1]:
```
from pprint import pprint
```

In [26]:
```
def is_valid_path(x, y, map, visited, color):
  # check if the coods are in the map, is a road and is not visited yet
  rows = len(map)
  cols = len(map[0])

  return 0 <= x < rows and 0 <= y < cols and map[x][y] == color and visited[x][y

def is_valid_coords(x1,y1,map, x2 = 1, y2 = 1):
  return 0 <= x1 < len(map[0]) and 0 <= y1 < len(map) and 0 <= x2 < len(map[0])


def editor():
  DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

  print("Please initialize grid by typing command: 'I M N' where M and N are the
  initialize_text = input("Enter command: ")

  split_text = initialize_text.split()
```

```
if len(split_text) != 3 or split_text[0] != "I":
  print("Incorrect initialization command")
  return None

width = int(split_text[1])
height = int(split_text[2])

grid = [[0 for i in range(width)] for j in range(height)]
pprint(grid)

while True:
  text = input("Enter command: ")

  visited = [[False for k in range(len(grid[0]))] for l in range(len(grid))]

  if text == "X":
    return None

  if text == "":
    print("Please input command")

  else:
    split_text = text.split()

    if split_text[0] == "L" and len(split_text) == 4:
      x = int(split_text[1]) - 1
      y = int(split_text[2]) - 1
      if is_valid_coords(x ,y, grid) == False:
        print("Index out of range")
      else:
        grid[y][x] = split_text[3]

    elif split_text[0] == "V" and len(split_text) == 5:
      x = int(split_text[1]) - 1
      y1 = int(split_text[2]) - 1
      y2 = int(split_text[3]) - 1
      if is_valid_coords(x,y1, grid, y2 = y2) == False:
        print("Index out of range")
      else:
        for i in range(y1, y2 + 1):
          grid[i][x] = split_text[4]

    elif split_text[0] == "H" and len(split_text) == 5:
      x1 = int(split_text[1]) - 1
      x2 = int(split_text[2]) - 1
      y = int(split_text[3]) - 1
      if is_valid_coords(x1,y, grid, x2 = x2) == False:
        print("Index out of range")
      else:
        for i in range(x1, x2 + 1):
          grid[y][i] = split_text[4]

    elif split_text[0] == "K" and len(split_text) == 6:
      x1 = int(split_text[1]) - 1
      x2 = int(split_text[2]) - 1
      y1 = int(split_text[3]) - 1
      y2 = int(split_text[4]) - 1
      if is_valid_coords(x1,y1, grid, x2 = x2, y2 = y2) == False:
        print("Index out of range")
      else:
```

```
          for i in range(x1, x2 + 1):
            for j in range(y1, y2 + 1):
              grid[i][j] = split_text[5]

      elif split_text[0] == "F" and len(split_text) == 4:
        x = int(split_text[1]) - 1
        y = int(split_text[2]) - 1

        if is_valid_coords(x ,y, grid) == False:
          print("Index out of range")
        else:
          c = split_text[3]
          prev_c = grid[x][y]
          visited[x][y] = True

          queue = deque([(x,y)])

          while queue:
            x, y = queue.popleft()
            grid[x][y] = c

            for dx, dy in DIRECTIONS:
              mx, my = dx + x, dy + y
              if is_valid_path(mx, my, grid, visited, prev_c):
                queue.append((mx,my))
                visited[mx][my] = True

      elif split_text[0] == "S" and len(split_text) == 2:
        print(split_text[1])
        pprint(grid)

      else:
        print("Invalid command!")
```

In [25]:
```
editor()
```

```
Please initialize grid by typing command: 'I M N' where M and N are the respectiv
e width and height
[[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]]
one.bmp
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
Index out of range
Please input command
one.bmp
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 0, 0, 0, 'K'],
 [0, 0, 0, 0, 'K']]
Please input command
Index out of range
one.bmp
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 0, 0, 0, 'K'],
 [0, 0, 0, 0, 'K']]
Please input command
one.bmp
[['C', 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 0, 0, 0, 'K'],
 [0, 0, 0, 0, 'K']]
one.bmp
[['C', 0, 0, 0, 0],
 [0, 0, 'Z', 'Z', 0],
 [0, 'W', 0, 0, 0],
 [0, 'W', 0, 0, 0],
 [0, 0, 0, 0, 'K'],
 [0, 0, 0, 0, 'K']]
one.bmp
[['C', 'J', 'J', 'J', 'J'],
 ['J', 'J', 'Z', 'Z', 'J'],
 ['J', 'W', 'J', 'J', 'J'],
 ['J', 'W', 'J', 'J', 'J'],
 ['J', 'J', 'J', 'J', 'K'],
 ['J', 'J', 'J', 'J', 'K']]
```