Lab exam 3

Graphs

- Types of graphs:
  - Directed graphs
  - Undirected graphs
  - Weighted graphs
- Graph representations:
  - Adjacency matrix - a matrix where if there is a 1 at position `(i,j)` that means there is an edge from i to j. For undirected graphs this matrix is symmetric.
  - Incidence matrix - a matrix where each row represents a vertex and each column represents an edge. The value at position `(i,j)` indicates that vertex i is incident to edge j.
  - Edge list - a simple list of tuples where each tuple represents an edge in the graph.
  - Adjacency/Successor list - a dictionary where keys are vertices and values are lists of neighbouring vertices.
- DFS for graphs (adjacency matrix)

```
def dfs(a, v):
    visited = []
    def traverse(a, v):
        visited.append(v)
        for j,k in enumerate(a[v]):
            if k == 0: continue
            if j not in visited: traverse(a, j)
    traverse(a, v)
    print(visited)
```

- BFS for graphs (adjacency matrix)

```
def bfs(a, v):
    visited = [v]
    temp = [j for j,k in enumerate(a[v]) if k == 1]
    while len(temp) > 0:
        visited.append(temp[0])
        for j,k in enumerate(a[temp[0]]):
            if k == 1 and j not in visited:
                temp.append(j)
        temp.pop(0)
    print(visited)
```

Backtracking algorithms

- Backtracking algorithms are algorithms for searching some kind of a solution space with a specific set of constraints. They use DFS to go through a state space tree, which represents the whole solution space. If some branch of solutions doesn’t meet the constraints then the DFS goes back and explores another branch.
- It often has an exponential time complexity, but it’s still more efficient then brute force.
- It is an exact algorithm, if one of the solutions is better than all, then it will return that solution.
- It requires a modest amount of memory.

Hamiltonian Cycle

- An algorithms for checking if there exists a Hamiltonian cycle in a graph with adjacency matrix representation.
- A Hamiltonian cycle is a path where each node is visited exactly once and it returns to the destination.

```
def hamiltonian(graph):
    path = [0]
    def dfs(path):
        if len(path) == len(graph) and graph[path[-1]][path[0]]
            return True, path
        for i in range(len(graph)):
            if i in path: continue
            if graph[path[-1]][i] != 1: continue
            new_path = path.copy()
            new_path.append(i)
            result = dfs(new_path)
            if result[0]: return True, result[1]
        return False, []

    return dfs(path)
```

N-Queens

- We have an n by n chess board and we need to place n queens on it so that no two queens attack each other. Queens move vertically, horizontally or diagonally.
- The output is in a form of a matrix that has 1’s where the queens are supposed to be and 0’s elsewhere.

```
import copy
def n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    def dfs(board, col, queens):
        if queens == n:
            return True, board
        for i in range(n):
            if any(board[i]): continue
            diagonal = False
            x,y = i, col
```

```
            # We only care about the left half of a diagonal
            while 0 <= x and 0 <= y:
                if board[x][y] == 1:
                    diagonal = True
                    break
                x -= 1
                y -= 1
            if diagonal: continue
            diagonal = False
            x,y = i, col
            while x < n and 0 <= y:
                if board[x][y] == 1:
                    diagonal = True
                    break
                x += 1
                y -= 1
            if diagonal: continue
            new_board = copy.deepcopy(board)
            new_board[i][col] = 1
            result = dfs(new_board, col+1, queens+1)
            if result[0]: return result
        return False, []

    return dfs(board, 0, 0)
```

Sudoku solver

- We are given a 9×9 grid filled with numbers from 1 to 9, or empty cells (represented by 0's). The goal is to fill the whole grid so that each number appears exactly once in every row, column or 3×3 box (there are 9 such boxes).

```
import copy
def get_box(grid, i, j):
    a = i // 3
```

```
    b = j // 3
    result = set()
    for x in range(a*3, (a+1)*3):
        for y in range(b*3, (b+1)*3):
            result.add(grid[x][y])
    return result

def sudoku(grid):
    def dfs(grid, row, column):
        if all([all(r) for r in grid]):
            return True, grid
        if grid[row][column] != 0:
            return dfs(grid, (row*9+column+1)//9, (row*9+column-
        for num in range(1,10):
            if num in grid[row]: continue
            if num in [grid[i][column] for i in range(9)]: conti
            if num in get_box(grid, row, column): continue
            new_grid = copy.deepcopy(grid)
            new_grid[row][column] = num
            result = dfs(new_grid, (row*9+column+1)//9, (row*9+c
            if result[0]: return result
        return False, []

    return dfs(grid, 0, 0)
```

Knapsack problem

- Given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- You are given two lists with equal lengths and an integer representing the limit.

```
def knapsack(weights, values, limit):
    def dfs(w, v, n):
        if n == len(weights):
```

```
            return v
        if weights[n] + w > limit:
            return dfs(w,v,n+1)
        else:
            return max(
                dfs(w + weights[n], v + values[n], n+1),
                dfs(w, v, n+1)
            )
    return dfs(0,0,0)
```

Combinatorial sum

- Given a list of integers and a target sum, write a function to find all unique combinations of the integers that sum up to the target. Each number in the list may be used once. Return a list of lists containing the combinations.

Letter combinations of a Phone Number

- Given a string containing digits from 2-9, return all possible letter combinations that the number could represent (simply return all combinations for each number choosing one letter out of three given in the mapping).

Dynamic Programming

- Dynamic programming is just using solving a problem which normally would use recursion and storing intermediate results somewhere.
- There are two subcategories of dynamic programming:
    - Memoization: This is a top-down approach. We store intermediate results in an array of a hash map and if a function is called we check if we already have a solution to that problem, if not then we compute it recursively and store it in the data structure. Usually faster if we call this function multiple of times, it remembers the intermediate results after the call.
    - Tabulation: Bottom-up approach. When a function is called we compute each intermediate step from the first to the last and store intermediate

results in a table. This approach doesn’t use recursion, but can solve a problem, which usually involves recursion much faster.

Knapsack

```
def knapsack(weights, values, limit):
    dp = [[0 for _ in range(limit+1)] for _ in range(len(weights
    for i in range(1, len(weights)+1):
        for j in range(1, limit+1):
            if weights[i-1] > j:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = max(
                    dp[i-1][j],
                    dp[i-1][j-weights[i-1]] + values[i-1]
                )
    return dp[len(weights)][limit]
```

Unbounded Knapsack

```
def knapsack_unbounded(weights, values, limit):
    dp = [0 for _ in range(limit + 1)]

    for i in range(1, limit + 1):
        for j in range(len(weights)):
            if (weights[j] <= i):
                dp[i] = max(dp[i], dp[i - weights[j]] + values[

    return dp[limit]
```