1. **Traveling Salesperson Problem (TSP)** -- Find an optimal (i.e., minimum length) tour when at least one tour exists. A tour (or Hamiltonian circuit) is a path from a vertex to a vertex back to itself that passes through each of the other vertices exactly once. (Since a tour visits every vertex, it does not matter where you start, so we will generally start at $v_0$.)

   a) What are the length of the following tour?
   \[ [v_0, v_3, v_4, v_1, v_2, v_0] = 23 \]

   b) List another tour starting at $v_0$ and its length.
   \[ v_0, v_3, v_2, v_1, v_4, v_0 = 16 \]

   c) For a graph with "n" vertices ($v_0, v_1, v_2, \ldots, v_{n-1}$), one possible approach to solving TSP would be to brute-force generate all possible tours to find the minimum length tour. "Complete" the following decision tree to determine the number of possible tours.

   \[
   \begin{array}{c}
   v_0 \\
   v_1 \\
   v_2 \\
   \vdots \\
   v_{n-1} \\
   v_n \\
   \vdots \\
   v_0
   \end{array}
   \]

   \[
   (n-1) \times (n-2) \times (n-3) \times \cdots \times 1 = (n-1)!
   \]

   Unfortunately, TSP is an "NP-hard" problem, i.e., no known polynomial-time algorithm.
2. Handling "Hard" Problems: For many optimization problems (e.g., TSP, knapsack, job-scheduling), the best known algorithms have run-time's that grow exponentially ($O(2^n)$ or worse). Thus, you could wait centuries for the solution of all but the smallest problems!

Ways to handle these "hard" problems:

- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the $2^n$ worse solutions, e.g., Backtracking (section 4.6) and Best-first-search-branch-and-bound
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., $O(n^3)$) solution. e.g., TSP problem satisfying the triangle inequality, Fractional Knapsack problem
- Use an approximation algorithm to find a good, but not necessarily optimal solution

**Backtracking** general idea: (Recall the coin-change problem from lectures 10 and 13)

- Search the "state-space tree" using depth-first search to find a suboptimal solution quickly
- Use the best solution found so far to prune partial solutions that are not "promising," i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time. However, in the worst case, the algorithm is still exponential.

My simple backtracking solution for the coin-change problem **without pruning**:

```python
def recMC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in coinValueList:
            if i <= change:
                numCoins = 1 + recMC(change - i, coinValueList)
                if numCoins < minCoins:
                    minCoins = numCoins
    return minCoins
```

Results of running this code:

<table>
<thead>
<tr>
<th>Change Amount: 63 Coin types: [1, 5, 10, 25]</th>
<th>Run-time: 0.036 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Amount: 63 Coin types: [25, 10, 5, 1]</td>
<td>Run-time: 0.003 seconds</td>
</tr>
</tbody>
</table>

Fewest number of coins 6
Number of Backtracking Nodes: 4831

Consider the output of running the backtracking code **with pruning** twice with a change amount of 63 cents.

<table>
<thead>
<tr>
<th>Change Amount: 63 Coin types: [1, 5, 10, 25]</th>
<th>Run-time: 0.036 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Amount: 63 Coin types: [25, 10, 5, 1]</td>
<td>Run-time: 0.003 seconds</td>
</tr>
</tbody>
</table>

Fewest number of coins 6
Number of Backtracking Nodes: 310

a) With the coin types sorted in ascending order what is the first solution found?

```
all pennies
```

b) How useful is the solution found in (a) for pruning?

```
not very
```

c) With the coin types sorted in descending order what is the first solution found?

```
greedy solution
```

d) How useful is the solution found in (c) for pruning?

```
very good
```
e) For the coin-change problem, backtracking is not the best problem-solving technique. What technique was better?
dyn, programming to avoid recalculating the same problem

3. a) For the TSP problem, why is backtracking the best problem-solving technique? many times

For TSP, dyn, programming solutions are $O(2^n)$

b) To prune a node in the search-tree, we need to be certain that it cannot lead to the best solution. How can we calculate a “bound” on the best solution possible from a node (e.g., say node with partial tour: $[v_0, v_4, v_1]$)?
Lecture 28

Some place reasonable. Complete the backtracking space using when pruning.

1. To prune a level in the search-tree, we need to be certain that it cannot lead to the best solution.

2. If a node has edges leaving the remaining nodes going to other nodes, then a node not leading to the best solution can be pruned.

Data Structures (CS 1520)
(b) Complete the best-first search with branch-and-bound state-space tree with pruning. Indicate the order of nodes expanded.

Put unexpanded nodes in BHeap. So finding smallest-bound node is fast.

(a) What type of data structure would we use to find the most promising node to expand next?

- A priority queue
- A hash table
- A linked list
- A binary heap

When we expand the most promising (best) node first by visiting its children:
- It expands the most promising following node which might be that node, i.e., how "promising" following node might be.
- It calculates a "bound" estimate for each node that indicates the "best" possible solution that could be obtained from any node in the subtree rooted at.
- It does not limit to any particular search pattern in the state-space tree like backtracking's left-most branch to right-most branch.

3. An alternative to backtracking is the best-first search with branch-and-bound approach.

Name: