HINTS TO CSCE 340 STUDY GUIDE 4 PART 1

DR. GODFREY MUGANDA

Do not look up solutions for the study guide on the Internet: that defeats the purpose of you trying to figure it out yourself. Come for help if you need it!

1. Backtracking Algorithms

1. Assume that you have a directed graph specified by an adjacency list. You are given two vertices A and B.

Write an algorithm that prints *all* paths from the source vertex A to the destination vertex B. Consider only paths that go through each vertex at most once.

If you want to write a program implementing your algorithm, compare this problem to the *Firetruck* problem at the UVA online judge programming competition site.

Hints:

Look back in your notes at the backtracking solutions for generating *all* subsets of a set, and all permutations of a set. There is also an example program on the K-drive that solves generation of all subsets and all permutations of a set.

Start with a partial solution that is a list consisting of just the source vertex A.

1.1 How do you recognize a complete solution? When a complete solution is recognized, print it out and return.

1.2 If a partial solution p is not complete, consider every vertex v in the graph as a candidate for extending p. How do you define compatibility for extension?

2. Devise a backtracking algorithm for solving the 0-1 Knapsack problem (without repetition) on page 167 of the class textbook.

Extensive Hints:

If the items are 0, 2, ..., $n-1$, A solution to the knapsack problem is just a subset S of {0, ..., $n-1$ } where

(1) the sum of all weights $w[i]$ for $i \in S$ is less than the capacity C of the knapsack, that is
$$\sum_{i \in S} w[i] \leq C,$$
(2) the sum of the values $v[i]$ for all $i \in S$ is as high as possible.

To simplify the algorithm, assume that we have global data structures

```
int [] w;           // w[i] is weight of item i.
int [ ]v;           // v[i] is value of item i.
int initCapacity;   // initial capacity of the knapsack.

LinkedList<Integer> bestKnapsack = new LinkedList<>();
                // will hold the indices of the times
                // included in the knapsack for the best value.
int bestValue;  // The value of items in best knapsack.
```

Because a knapsack is a set rather than a list, our backtracking framework will proceed in stages $s = 0, 1, ..., n-1$, where at stage s of a partial solution p we have already passed through stages 0,.., $s-1$ and are working on determining whether s should be included or excluded from p.

The parameters are:

```
void extend( LinkedList<Integer> p,  // items in current knapsackl
             int  s,                 // current stage.
             int currentValue,       // total value of current knapsack.
             int remainCapacity      // capacity remaining in knapsack.
           )
```

The goal of extend is to extend p is to find all knapsacks that extend p, and to record the knapsack with the best value in the two global variables

```
LinkedList<Integer> bestKnapsack;
int bestValue;
```

Compatibility: An item s can only be added to the p if it fits in the knapsack, meaning that
$$w[s] \leq \text{remainCapacity}.$$

The initial call to `extend()` in `main()` is:

```
// Main
LinkedList<integer> p = new LinkedList<>();
LinkedList<integer> bestKnapsack = new LinkedList<>();
int bestValue = 0;

int s = 0;
extend (
           p, // initial empty knapsack
           0, // stage
           0, // currentValue
           initCapacity,   // capacity remaining in knapsack
       );
System.out.println (bestKnapsack);
```

Here are the main points in writing the `extend()` method.

(1) if $s = n$, the partial solution cannot be extended so return.
(2) if s does not fit in the remaining capacity, the partial solution cannot be extended so return.
(3) extend without adding s to p.
(4) extend with s added to p as follows:
   (a) Add s to p.
   (b) Set a local variable
   ```
   int value = currentValue + v[s];
   ```
   This is the new value of the augmented knapsack. If this new value is greater than the global best value, replace both the global best value and the global best knapsack:
   ```
   if (value > bestValue)
   {
      bestValue = value;
      bestKnapsack.clear();
      bestKnapsack.addAll(p);
   }
   ```
   Once this is done, make the recursive call to `extend()`. Make sure you get the parameters right!
   (c) Do not forget to undo the addition of s to p when the recursive call returns!

When you return back to `main()`, the global best value and best knapsack variables will have the solution you want.

Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E. A subset $C \subset V$ is called a *vertex cover* of G if every edge of G has at least one of its end points in C.

3. Assume that a graph is specified by an adjacency matrix M, and that a set of vertices C is represented by a Java linked list. Write a Java method that returns **true** if C is a vertex cover and **false** otherwise.

HINT: The method signature looks like this

```
boolean isCover(LinkedList<Integer> C, boolean[ ][ ] M)
```

4. Given an integer k where $1 \leq k \leq |V|$, devise a backtracking algorithm for computing a vertex cover of size at most k is one exists. Follow the backtracking framework we have been using in this course by writing a method

```
boolean extend(C, k, s,  M)
```

where C is a linked list representing a *partial solution* at stage s and M is the adjacency matrix.

2. Dynamic Programming

Do problems 6.1, 6.2, 6.3, 6.6, and 6.25 in the class textbook.

Do not feel discouraged if you can't figure all of them out. Just try to solve as many of them as you can. As you study for the quiz, the most important thing is to determine the recurrence and the boundary conditions.

Even if you cannot figure out a problem, you will be in a better position to understand the solution when you come for help.

Hints to these Dynamic Programming problems will come later, for now focus on dynamic programming.