CS330 – Dynamic Programming

Purpose: Given an optimization problem, we want to efficiently find the optimal solution.

Knowledge: This activity will help you become familiar with the following content knowledge:

- How to recursively solve optimization problems
- How to do a bottom-up solution of optimization problems for efficiency

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 60 minutes.

1. We want to find choices that maximize or minimize some quantity. An example is cutting a rod. We are given a rod of length n and a table of prices p_i for $i = 1, ..., n$ where p_i is the price of a rod of length i. The goal is to determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

   Try to solve this for $n = 5$ and $p_1 = 1, p_2 = 5, p_3 = 8, p_4 = 9, p_5 = 10$. Observe that a greedy approach does not work.

2. Notice that with the rod cutting, if the optimal solution has a cut of length i, and we took away that length i, what remains must be an optimal solution for a rod of length $n - i$.

   Why must that be true? Hint: Think about what it would mean if we had a better solution for $n - i$ and how that would give a contradiction to the supposition that we had an optimal solution for length i.

3. We can recursively define a solution by figuring out where to make the first cut to maximize our possibilities with what is leftover. We want to maximize over the possibilities with the first cut:

   $p_1 + R(n-1)$,
   $p_2 + R(n-2)$,
   ...
   $p_{n-1} + R(1)$,
   p_n
   Complete the recursive algorithm which does this:

```
R(n,p[1..n]){
  if (n==1){

  }else{
     max = 0;
     for (i = 1, i<=n; o++){
         x = p[i] + R(n-i);



     }
     return max;
 }
}
```

4. Using the algorithm above, if we called R(4), how many times would each of R(3), R(2), and R(1) be called?

5. This recursive approach is not very efficient since it makes the same recursive calls many times. A better approach is a to solve the problems in a bottom-up fashion. We solve the smallest subproblems first and combine solutions of small subproblems to solve larger ones:

```
R(n,p[1..n]){
  r[0] = 0;
  for (j=1; j<=n; j++){
     max = 0;
     for (i=1; i<=j; i++) {
         x = p[i] + r[j-i];
         if (x>max) max = x;
     }
     r[j] = max;
  }
  return r[n]
}
```

   Iteratively fill in the r array by calculating r[1], r[2], r[3], ... for $n = 5$ and $p_1 = 1, p_2 = 5, p_3 = 8, p_4 = 9, p_5 = 10$.

6. This bottom-up approach doesn't wait until a subproblem is encountered but instead solves the smallest subproblems first and combines the smaller subproblems to solve the larger ones. What is the order of growth of this algorithm?

7. Modify the code so it outputs the cuts rather than just the maximum profit. You can do this by creating another array s and store in s[j] the i value that achieves the max for that subproblem.

8. We will look some problems which can be solved with dynamic programming to give you some practice.

   Problem: Given an array A[1..n] of real numbers, find the numbers j and k so that the sum from j to k, A[j..k] is maximal.

   Step 1: Let S[k] represent the maximal subsequence that ends in position k. Write a recursive definition for S[k+1].

9. Problem: Given an array A[1..n] of real numbers, find the numbers j and k so that the sum from j to k, A[j..k] is maximal.

$$S[0] = 0$$
$$S[k+1] = max\{S[k] + A[k+1], A[k+1]\}$$

Step 2: Write bottom-up code to compute S[k] and use another array T[k] to store the starting index.

10. Problem: Given a graph with vertices labeled 1..n, find the shortest path between all possible pairs of vertices.

Step 1: Let $D[i, j, m]$ represent the shortest path from vertex i to vertex j with at most m edges. Let $w[i, j]$ be the weight of the edge from vertex i to vertex j. Write a recursive definition for $D[i, j, m + 1]$.

11. Problem: Given a graph with vertices labeled 1..n, find the shortest path between all possible pairs of vertices.

$$D[i, j, 0] = 0$$
$$D[i, j, m + 1] = \min\{D[i, j, m], \min_{0<k\leq n}\{D[i, k, m] + w[k, j]\}\}$$

Step 2: Write bottom-up code to compute $D[i, j, n]$ for all pairs of i and j. What is the order of growth?

12. Problem: Given a graph with vertices labeled 1..n, find the shortest path between all possible pairs of vertices.

Step 1: Let $D[i, j, m]$ represent the shortest path from vertex i to vertex j **which onlu uses paths through vertices 1 through m**. Let $w[i, j]$ be the weight of the edge from vertex i to vertex j. Write a recursive definition for $D[i, j, m + 1]$.

13. Problem: Given a graph with vertices labeled 1..n, find the shortest path between all possible pairs of vertices.

$$D[i, j, 0] = w[i, j]$$
$$D[i, j, m + 1] = \min\{D[i, j, m], D[i, m + 1, m] + D[m + 1, j, m]\}$$

Step 2: Write bottom-up code to compute $D[i, j, n]$ for all pairs of i and j. What is the order of growth?

Complete the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies.

Procedure for Dynamic Programming problems

We want to find the choices that maximize or minimize some quantity. Here's how we could approach this with dynamic programming

1. First write a recursive solution for the max (or min) quantity.
   (a) For each initial choice you can make, compute the cost of making that choice plus the (recursively calculated) best way of making all other choices.
   (b) Return the maximum (or minimum) of all the possible choices.
2. Then, it is usually more efficient to write a non-recursive version:
   (a) Compute "base cases" and write answers into an array.
   (b) Have a loop compute higher values in terms of already computed values in the array
   (c) The code should look very similar to the recursive version.
3. If needed, modify code to keep track of the choices made as well as the quantities they yield.

Assignment 1:
We will start by focusing on writing a top-down recursive solution first.

Suppose you have a knapsack that can hold a weight of up to W. You are robbing a store, and in front of you are items $1 \ldots n$ with values v_i and weights w_i. You want to pick the items to put in your knapsack so that you maximize the total value of what's in the knapsack.

1. A greedy approach is to take the most valuable item that will fit in your knapsack first, then the next most valuable item, etc. Find an example where this doesn't yield an optimal solution.

2. Another form of the greedy approach is to first take the most valuable item by weight. That is, you compute the price per pound of each object, then take the item with highest price-per-pound, etc. Suppose $W = 50$ and you have three items with $v_1 = 60, v_2 = 100, v_3 = 120$ and $w_1 = 10, w_2 = 20, w_3 = 30$. Does this greedy approach work for this example?

3. Let's consider whether the last item, n,should go in the knapsack or not. Clearly it won't go in if its weight is too large. If it does go into the knapsack, what recursive subproblem will we need to solve?

4. If item n does not go into the knapsack, what recursive subproblem do we need to solve?

5. How would we test whether n goes into the knapsack or not?

6. Complete a *recursive* function `knapsackRecursive(int W, int[] wt, int[] val,int n)` in the code provided. This function should compute the maximal possible total value that can fit into a knapsack of capacity W when the items have values $val[i]$ and weights $wt[i]$

Criteria for Success: You have written counterexamples for the greedy approaches and working code for the recursive solution.

Assignment 2:
Now we will develop a bottom-up solution.

Write a bottom-up solution for the knapsack problem. We will create an array such that `K[i][w]` will contain the optimal solution for when we have i items and we can carry weight w. We will use nested loops to fill in the values in this array. Instead of doing a recursive call as before, we will simply use a value that we previously assigned in the array. Complete the new version and test it out.

Criteria for Success: You have working code for the bottom-up solution.

Assignment 3:
We will now develop a full solution.

Modify your previous solution to keep track of the optimal choices being used. You will want to create another array to store that information.

Criteria for Success: You have code which provides the optimal choices for the knapsack problem.