Week 6 Lecture 17

Theory

What's in this lecture?

- Dynamic Programming
  - Faster algorithms using memoization

Problem Solving

- Often times, an algorithm may be phrased as a function of recursive solutions to smaller sub-problems
- If the *optimal* solution to the problem is always composed of an *optimal* solution of sub-problems, then Dynamic Programming may be used

Dynamic Programming

- In dynamic programming, we keep track of (or *memoize*) the results of sub-problem solutions to avoid re-computing those results
- If the size of sub-problems is dramatically smaller than the original problem (such as with mergesort), we often use the term “divide and conquer” instead

Fibonnaci

```
function fib(n) {
  if (n <= 1) {
    return 1;
  }
  // no memoization: always compute
  return fib(n - 1) + fib(n - 2);
}
```

Memoized Fibonnaci

```
var fibmap = { “0” : 1,“1”: 1 };
function mfib(n) {
  if (!fibmap[n]) {
    fibmap[n] = mfib(n - 1) + mfib(n - 2);
  }
  return fibmap[n];
}
```

Tangent: Shortest Path

- Consider the problem of finding the shortest road distance between two cities on a map, where roads connect each city
- This problem displays optimal substructure: that is, the shortest path will be composed of shortest paths between other cities
- Rationale: if there was a shorter path between 2 cities along the way, we could substitute it into the result and find a shorter path

Knapsack Problem

- Given a fixed-size knapsack, choose a subset of items which have the most value
- Input is an array of items, such as: [{weight:4,value:5},{weight:2,value:3}]
- If capacity < 4 and capacity > 2, choose the second item
- If capacity >= 4, choose the first
- With more choices, is more complicated...

Knapsack Algorithm

- For each item, we try to find the maximum value of the weight we can carry *with* or *without* that item
- If the weight of the current item is greater than available capacity, we need not consider that item
- This is usually called the 0-1 knapsack problem, since each item may be carried zero or one times

Knapsack Algorithm

```
// initialization: 2D array of n items and
// capacity c
function make_array(n, c) {
  var results = new Array();
  for (var i = 0; i <= n; i++) {
    results[i] = new Array();
    for (var j = 0; j <= c; j++) {
      results[i][j] = 0;
    }
  }
  return results;
}
```

Knapsack Algorithm

```
function knapsack(capacity, items) {
  var n = items.length;
  var m = make_array(n, capacity);
  for (var i = 1; i <= n; i++) {
    var c = items[i-1];
    for (var w = 0; w <= capacity; w++) {
      if (w < c.weight) {
        m[i][w] = m[i - 1][w];
      } else {
        m[i][w] = Math.max(m[i-1][w],
                          m[i-1][w - c.weight] + c.value);
      }
    }
  }
  return m[n][capacity];
}
```

Exercises

- Research the "integer knapsack problem", and modify the knapsack code to solve the case where each item may be "copied" as many times as necessary; (See Change-Making Problem for more details)
- Research the "weighted interval scheduling problem", find the algorithm and implement it in JavaScript