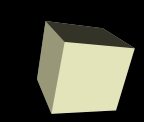
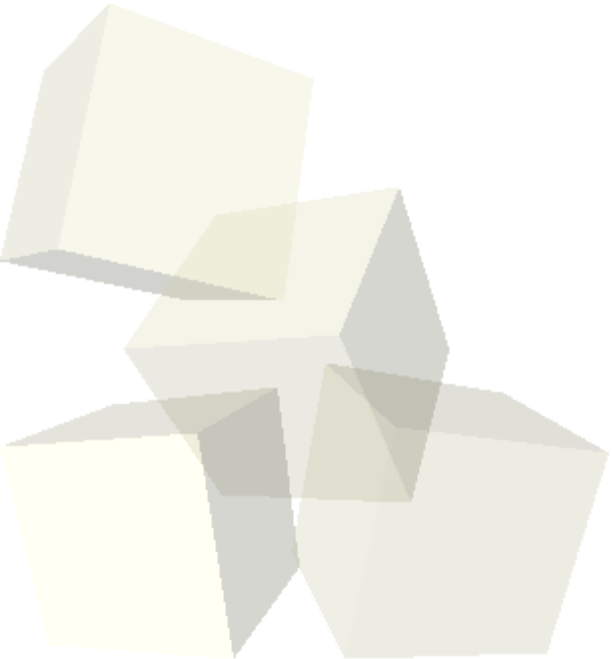


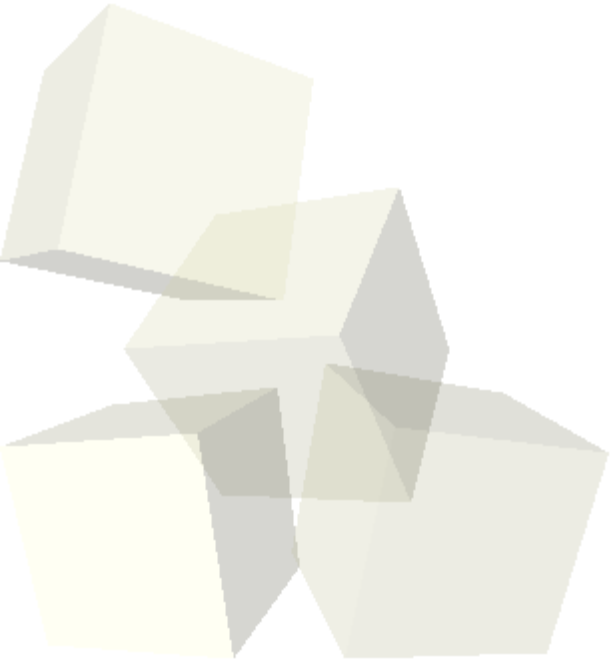
Memoization

2-16-2006



Opening Discussion

- What did we talk about last class?
- Do you have any questions about the assignment?



Duplicated Work

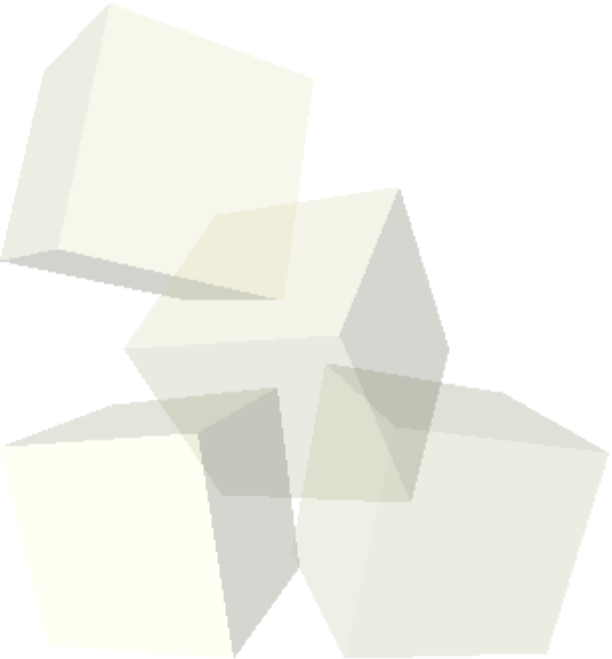
- We have looked at the technique of searching an entire solution space to find an optimal solution to a problem. We have also gone to the other extreme where we pick only one possible option at each step in solving a problem.
- The latter doesn't work for most problems and the former winds up doing a lot more work than is needed in many problems. Today we will begin looking at ways to get around this.

Memoization

- The simplest way to reduce duplicated work is to memoize our full recursive searches.
- The idea of memoization is that we will pass in an array (or possibly some other structure) that stores solutions that we have found so far. This is a bit different than “smart breadcrumbs” though the two are often lumped together.
- When thinking about memoizing our thought process should be that when you enter the recursion you check to see if you have solved this before. If so return that value. Otherwise do the recursion and set the value to the solution that you find.

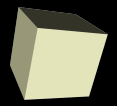
Memoizing Fibonacci Numbers

- As a simple example, we can memoize a recursive function to calculate Fibonacci numbers. You all know that this is a prime candidate for the process as it is not only simple, it duplicates all types of work.



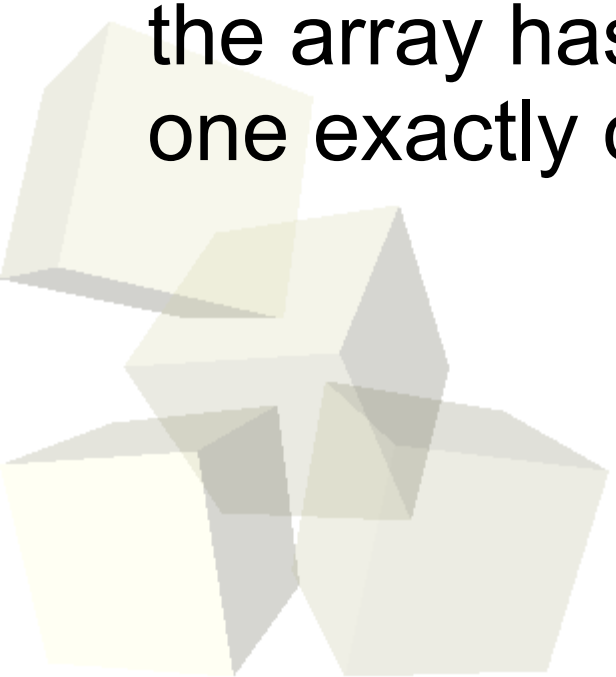
Weighted Interval Scheduling

- We showed that standard interval scheduling is a greedy problem when we want to maximize how many events we schedule. If we place weights, w(n), on the events greedy no longer works.
- To help with solving this efficiently we will sort the intervals by finish time and keep an array p(n) which is the index of the last interval that doesn't overlap with interval n.
- Given this, we can define the optimal solution on the first n intervals as the max of w(n)+opt(p(n)) and opt(n-1). Basically, we either take this option or we don't.



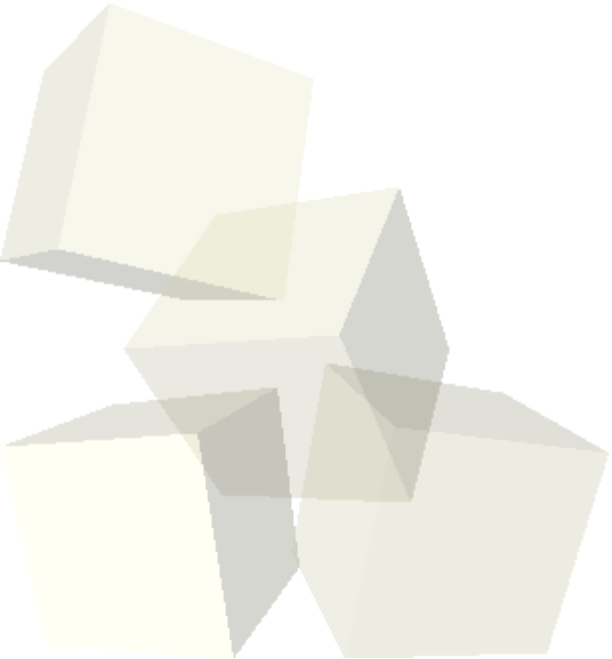
Memoizing It

- The problem is that this recursive solution will calculate the optimal value for the first few elements many times. We can keep a simple array that will store the solutions once they are found and do lookups on that array to get a significant speed boost.
- How fast is it now? This will make it O(n) because the array has n elements and we calculate each one exactly once.



Memoizing 0/1 Knapsack

- Let's go ahead and write a full recursive version of the 0/1 knapsack problem now.
- How do we memoize this problem? What do we keep track of in our array?

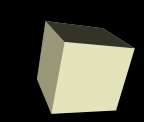


When to Memoize?

- In general we want to memoize any time when we see that our recursive solution is calculating the same thing more than once. This is common when a problem exhibits optimal substructure.
- As we will see next time, we can use dynamic programming instead of memoization when we have optimal substructure. DP will be slightly more efficient but memoization might be more intuitive.
- Memoization can also be used in situations where DP can't because subproblems aren't solved in a nice order or because we don't have true optimal substructure.

Memoizing TSP

- This latter case is exemplified by the traveling salesman problem. It does not have optimal substructure. However, the standard recursive solution to this does repeat work.
- It is tempting to do this with “smart breadcrumbs”, but that isn't correct. It could return non-optimal results. We need to use memoization so we store the length of completing the rest of the cycle from a given state?
- What is our state and how does that index into an array?



Reminders

- I should have your test already, but if I don't you should probably fix that.
- Read up on dynamic programming for next class and remember that assignment #3 is due on Tuesday.