CSC373S Lecture 7

- The way I like to think about greedy algorithms is to first try to define a "semantic" array that indicates what we are trying to compute. In the case of weighted interval selection, we are trying to compute $OPT'(j)$= the optimal value that can be obtained by a non conflicting subset of the intervals $I_1, \ldots I_j$. I am using OPT' to distinguish it from the recursively defined array OPT. Formally, one needs to prove that $OPT'(j) = OPT(j)$ for all j. We usually do this somewhat informally (often waiving our hands) but we really should have a proof that the two arrays are equivalent. Of course, the array OPT' is defined in such a way that it 1) will be easy to obtain the desired answer from one or more entries of the array and 2) every entry of the array can be easily computed from "previous" entries of the array. That is, when we define the array OPT' we usually have some recursively defined way of computing it (i.e. OPT)

- The slides show that a naive implementation of OPT will be very inefficient (exponential time) but that the use of *memoization* provides an efficient recursive implementation. And there is a very efficient iterative implementation. One usually uses the iterative implementation but usually the conception of the algorithm is in terms of the recursive definition.

- Finally, we address the question as to how to compute an optimal solution and not just the value of an optimal solution. The slides give a recursive method for constructing an optimal solution once the optimal value has be determined. Another method (which may not be as efficient but is conceptually simpler) is to construct the solution while computing the optimal value. Namely, when one chooses between not including (resp. including) I_j in computing $OPT(j)$, we can add a line of code $SOL(j) := SOL(j-1)$ (resp. $SOL(j) := SOL(j-1) \cup \{I_j\}$) according to which choice afford the higher value.

- Another problem that has a very similar DP solution is the Longest Increasing Subsequence LIS problem (see Jepson notes).

  Given a string $x_1, x_2, \ldots x_n$ with x_i in some totally ordered set (say the integers), find the longest subsequence $x_{j(1)} < x_{j(2)} \ldots < x_{j(p)}$ with $1 \leq j(1) < j(2) \ldots < j(p)$.

  An appropriate semantic array array for this problem is $Q'(k)$ for $1 \leq k \leq n$ where $Q'(k)$ = the length of the longest increasing sequence ending at the k^{th} symbol x_k. The desired optimum value is then $\max_k Q'(k)$. The recursive computational array is defined by $Q(k) = 1 + \max_j[Q(j) : 1 \leq j < k \text{ and } x_j < x_k]$ for $1 \leq k \leq n$. Note that this implicitly defines $Q(1) = 1$

  One should then prove by induction that $Q(k) = Q'(k)$ for $1 \leq k \leq n$. And as in the weighetd interval selection problem, knowing the array Q allows one to compute an optimal solution corresponding to the optimum value.

- Both the weighted interval selection and LIS DPs share the fact that the correctness is established by induction on the "latest" item that can be part of the solution.

We consider another example of a DP that shares this DP style, namely DPs for the *knapsack problem* (and also called the {0,1} knapsack problem) defined as follows:

We are given n objects $\{(w_i, v_i), \ldots (w_n, v_n)\}$ and a knapsack weight bound W. We think of w_i as the weight or size of the i^{th} object and v_i as its value. (Here we will assume all parameters are positive integers.) A feasible solution is a set of (indices of) objects $T \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in T} w_i \leq W$. The goal is to find a feasible solution T so as to maximize $\sum_{i \in T} v_i$. That is, we are trying to maximize the value of objects that can be placed in the knapsack.
This is a problem that is easy to motivate. It also has some algorithmic significance. It is a NP hard optimization problem but one that is only "weakly NP hard" in the sense that if either all the weights w_i are "small" or all the values v_i are small then the problem can be solved in polynomial time. To be more precise if these integer parameters are encoded in unary then the encoding has polynomial length. Moreover, we can obtain a $(1 + \epsilon)$ approximation for any $\epsilon > 0$ in time linear in n and $\frac{1}{\epsilon}$.

- It is easy to formulate a reasonable semantic array and a recursive array which does not correctly compute what is desired. Namely, suppose we define the semantic array S' : $S'(w)$ = maximum value obtainable by a feasible solution T whose total weight is at most w; i.e. $\sum_{i \in T} w_i \leq w$. Then we could define a recursive computational array S: $S(0) = 0$ and $S(w) = \max_{k: w_k \leq w} v_k + S(w - w_k)$.

  Now waving hands we could claim that we have solved the problem but something is wrong here! That is, just writing down the recursive array doesn't mean that it is correct and hence that the approach of defining the computational array S' as we did is a viable approach.

  Moral: One really has to prove the equivalnce of the semantic and computational arrays for correctness. Often this seems obvious but it is also easy to make mistakes.

  What goes wrong here is that this recursive computational array allows the knapsack to have more than one copy of an object and the definition of the problem (and hence the name {0,1} knapsack problem) requires that an object can be used at most once.

- Here is one way to solve the {0,1} knapsack problem. We define the semantic array $S'(i, w)$ = largest profit obtainable by a feasible solution $T \subseteq \{1, \ldots i\} : \sum_{i \in T} w_i \leq w$. Now we can define an appropriate computational array $S(i, w)$ 'which can be proved correct by induction on i. Namely, we define :
  $S(i, w) = 0$ if $i = 0$ for all w and $S(i, w) = \max\{A, B\}$ for $i > 0$ where $A = S(i-1, w)$ and $B = v_i + S(i - 1, w - w_i)$ if $w_i \leq w$ and 0 otherwise.

  As already suggested , one proves $S'(i, w) = S(i, w)$ by induction on i.

- Lets consider the time complexity (assuming memorization is used to efficiently implement the recursion or an iterative implementation is used. There are nW

entries of S that have to be computed and assuming all previous entries have been computed, computing any $S(i, w)$ takes $O(1)$ time for a total time complexity of $O(nW)$. This is called a pseudo polynomial time algorithm since W might be large (i.e if the w_i and W are n bits then the encoding of the inout uses $O(n^2)$ bits and the complexity is $O(n2^n)$. But if all w_i are small (or if W is small), say $O(logn)$ bits, then the alg is running in polynomial time.

- There is an alternative DP for knapscak which is polynomial time when all values are small. This is in turn can be used to provide arbitrarily good approximations for every knapsack instance. Now instead of determining the maximum value for a given weight w, we want to determine the minimum weight to obtain a given value v.

  That is, we define a semantic array $M'(i, v)$ = the minimum weight w such that there is a solution $T \subseteq \{1, \ldots, i\}$ and $\sum_{j \in T} v_j \geq v$. The desired optimum value is the maximum v such that $M'(n, v) \leq W$. We now have to indicate how to compute each $M'(i, v)$ by providing an appropriate recursive array M.

  To initialize, we define $M(0, v) = \infty$ for all $v > 0$ which indicates that there is no solution if we do not have any items and define $M(0, v) = 0$ for all $v < 0$. For $i > 0$, we have:
  $M(i, v) = \min\{A, B\}$ where $A = M(i - 1, v)$ and $B = w_i + M(i - 1, v - v_i)$.

  The benefit of this method is that the complexity is now $O(nV)$ where $V = \max_i\{v_i\}$.