Part I Strong NP-Completeness and Pseudo-Polynomial Time Algorithms

Informally, a decision problem π is strongly NP-complete if the language corresponding to a "unary" encoding of it, which is reasonable except that each integer k is recorded using k +1 symbols instead of O(log k) symbols, is NP-complete (to be very precise we would want to specify more exactly the encoding we were considering).
A pseudo-polynomial time algorithm for π runs in time which is polynomial in the size of the input under such an encoding. Such an algorithm shows that the language corresponding to a unary encoding of π is in P.
We showed that TSP is strongly NP-complete by reducing Hamilton Cycle to a special encoding of TSP. Since the only numbers in the TSP instance are 1,2,and n, we see this is a polynomial reduction to a unary encoding, as even under the unary encoding we use at most n symbols for any of our integers, and hence at most polynomial time to write down any such integer. In the same vein IP-FEASIBILITY is strongly NP-complete because the reduction from SAT to IP-FEASIBILITY we consider is still polynomial if we reduce to a unary encoding. CLIQUE is also strongly NP-Complete as the only integer we use, k, is at most |V|.
In contrast, we presented two pseudo-polynomial time algorithms for Knapsack provided all weights are integer.

In the first we filled out HIGH_VAL[1..n,0..W] where HIGH_VAL[i,j] records the highest value of a subset of the first i items whose weight sums to j, or contains a -1 if there is no such subset. We proceed as follows:

HIGH_VAL[1,0]:=0, HIGH_VAL[1,w_1]: =v_1
FOR j := 1 TO W, IF j ≠ w_1 THEN HIGH_VAL[1,j]=-1
FOR i:=2 TO n
  FOR j:=0 TO W
    HIGH_VAL[i,j]:=HIGH_VAL[i-1,j]
  FOR j:=w_i TO W
    IF HIGH_VAL[i-1,j-w_i]≥ 0 AND v_i +HIGH_VAL[i-1,j-w_i]>HIGH_VAL[i,j] THEN
      HIGH_VAL[i,j]:= v_i +HIGH_VAL[i-1,j-w_i]
SOLUTION=0
FOR j:=1 to W IF HIGH_VAL[n, j]>SOLUTION THEN SOLUTION:= HIGH_VAL[n, j]

In the second we filled out L_W[1..n,0..V] where V is the sum of the values of the n items. L_W[i,j] records the lowest weight of a subset of the first i items of weight<W

whose value sums to j, or contains a W+1 if there is no such subset. We proceed as follows:

```
L_W[1,0]:=0, L_W[1,v1]: =w1
FOR j := 1  TO V, IF j ≠ v1 THEN L_W[1,j]= W+1
FOR i:=2 TO n
  FOR j:=0 TO V
     L_W[i,j]:=L_W[i-1,j]
  FOR j:=vi  TO V
      IF  wi +L_W[i-1,j-vi]<L_W[i,j]  THEN
          L_W[i,j]:= wi +L_W[i-1,j-vi]
SOLUTION=0
FOR j:=1 to V IF L_W[n, j]<W+1  THEN SOLUTION:= j.
```

The first of these algorithms run in O(nW) time, the second in O(nV) time.

Part 2 Dealing With NP-Complete Problems.

Determining that a problem is NP-Complete does not make it go away. Much of the rest of the course is devoted to studying how to handle NP-complete problems algorithmically. Approaches we investigate will include:

(1) algorithms which run in polynomial time and give approximately optimal solutions,

(2) algorithms which while they do not run in polynomial time on all inputs, run in polynomial time on average, and

(3) algorithms which run in polynomial time for a restricted class of inputs.

Pseudo-polynomial time algorithms are an example of the third kind of algorithm, they run in polynomial time on those inputs for which the value of the integers in the instance are bounded by a polynomial in the (classical not unary) input size.

For some strongly NP-Complete problems, we can get polynomial time algorithms by imposing even more draconian restrictions on the input. Thus, for example, while

Clique is strongly NP-complete, if we insist that the size k of the clique we are looking for is a constant then a brute force algorithm which checks for each set S of k vertices whether or not all the edges between these vertices are present solves the problem in $O(n^k)$ time. Thus the decision problem k-Clique in which an instance is a graph and we are asked if it contains a clique of size k, can be solved in polynomial time. In contrast 3-Colourability is NP-complete even though this is a restriction of the general Colourability problem to the 3 colour case.

Other problems can be solved quickly if we restrict our attention to specially structured inputs. For example, in a bipartite graph the size of a minimum cover is the same as the size of a maximum matching so we can compute this in polynomial time, even though in general Vertex Cover is NP-complete. Also, if we retrict our attention to IPs which satisfy

(*) every optimal solution to the fractional relaxation is integer valued

then we can solve them in polynomial time by solving their fractional relaxations. Finally we have seen a polynomial time algorithm for the restriction of SAT to instances where each clause has 2 literals.

We then turned to approximation algorithms for optimization problems. We observed that we can greedily construct in linear time a maximal matching and use its endpoints as a vertex cover which must be within a factor of 2 of the optimal. We then discussed an approximation algorithm for Knapsack, which consisted of rounding the values so that they were integers between 1 and cn for some constant c and then applying our pseudo-polynomial algorithm.

We talked about a second approximation algorithm for Knapsack for which yu are not responsible.