P

1.) To prove DH is P-complete we must show that DH ∈ P and $P \leq_M^P$-reducible to DH.

DH is an element of P because it can be decided in finite time. To do this we will make a TM T that decides DH

T: get length of input x
all input strings = {S_1, S_2, S_3, S_4 ... S_K}
i = 1
*{Run DH on S_i
if DH halt within n steps accept
else reject
i++
goto *
}

Since this TM will decide DH within a finite amount of time DH ∈ P, Since it visits each element once it is done in linear time.

Now we need to reduce P to DH.

First we will reduce P from the class of languages in polynomial to to the Class of languges in linear time. We will now reduce this to the set of languges in linear time that accept in n Steps.

$\{\langle M, x, 1^n \rangle$ M is a DTM that accepts on input x within n steps$\}$

Since we know A_{TM} reduces to HALT this must reduce to DH.

Since DH $\in$ P & P reduces to DH. DH is P-complete

2) In order to show MOD $\in$ P it would suffice to create a program that computes it in polynomial time. I will be using C++

```
int   mod(int n, int m, int r){
    if(n≤0 || m≤0 || r<0 || n<m){
      cout << "Error: invalid args" << endl;
      return -1; // Failure/Reject state
    }
    int sum = n;
    while(sum>0){
    sum= sum - m;
    }
    if(sum<0){
      sum = sum+m;
    }
    if(sum == r){
      return 1; // Success/Accept State
    }
    else{
      return -1; // Failure/Reject state
    }
}
```

Since we can write a program that solves MOD in Polynomial time MOD $\in$ P.

· NP

1) To show that the decision version of the napsack problem is in NP it will suffice to show that we can build a turing machine that solves it in Polynomial Time. To do this we construct the following TM:

{ Given an array of length k we will construct a new array containing all possible combinations
$\{(a_i), (a_i, a_j), \ldots\}$

We will now check each element of this array to check for an element that satisfies the following criteria:

1) total value of element $\geq k$
2) total weight of element $\leq L$

If 1 is found accept, else reject }

Since the first part of this program can be solved in k! time and the second part in linear time, this problem can be solved in Polynomial time. This means that the decision version of the napsack problem is in NP

2) If the decision version of the knapsack problem can be solved in polynomial time then so can the maximization version since we can simply replace the linear search with a search for only elements whose weight is less than L and add those to a new array. We will then bubble sort this new array to get the max element.

NP-Complete

1) To show that A is NP-complete we will construct a TM that solves A in polynomial time.

TM:{ get length of x

generate all input strings of length x

$\{s_1, s_2, s_3, \ldots s_k\}$

$i = 1$

*{ Run M on s_i

if M accepts x within $|x|+n$ steps

accept

else

reject

goto *

}

Since we can solve A in polynomial time (linear time), A is NP complete.

2) To show that A_0 is NP-complete we will construct a TM that solves A_0 in polynomial time.

TM: {

Run M on 0

if M accepts 0 within n steps

accept

else

reject

}

Since we can solve A_0 in polynomial time $(O(1))$ A_0 is NP-Complete.