A_1, A_2, A_3, …, A_N is a list of integers,
S is another integer.

IntKnapsack(N, S) = what sublist of A_1, A_2, A_3, …, A_N has sum S?

We can easily solve this, in polynomial time, with dynamic programming.

But suppose we are using real numbers (floats) instead of ints.

The general knapsack problem can't be solved by dynamic programming (because we can't build the table), or by any other known technique, except in exponential time.

If we could program with non-determinism, a solution would be easy, and polynomial time.

```
Knapsack(N, S) =
{ if (S == 0)
    return <>;
  if (N == 0 || S < 0)
    FAIL;
  CHOOSE
    return <A_N> ∪ Knapsack(N-1, S-A_N);
  OR
    return Knapsack(N-1, S); }
```

Knapsack is an NP problem.

The claimed solution is verifiable in polynomial time.

There is a related decision problem (boolean function):
Is there a sublist of A_1, A_2, A_3, …, A_N that has sum S?
call that BoolKnapsack(N, S).

BoolKnapsack solves Knapsack for us:

```
Knapsack(N, S) =
{ if (S == 0)
    return <>;
  if (N == 0 || S < 0)
    FAIL;
  if (BoolKnapsack(N-1, S-A_N))
    return <A_N> ∪ Knapsack(N-1, S-A_N);
  else
    return Knapsack(N-1, S); }
```

If we could solve BoolKnapsack it would be the oracle that lets us resolve the non-determinism.

It may be more convincing without recursion:

```
Knapsack(N, S) =
{ solution = <>;
  while (true)
  {  if (S == 0)
        SUCCESS;
     if (N == 0 || S < 0)
        FAIL;
     CHOOSE
     {  solution = solution ∪ <A_N>;
        S = S - A_N; }
     OR
        {}
     N = N - 1; } }
```

or

```
Knapsack(N, S) =
{ solution = <>;
  while (true)
  {  if (S == 0)
        SUCCESS;
     if (N == 0 || S < 0)
        FAIL;
     if (BoolKnapsack(N-1, S-A_N))
     {  solution = solution ∪ <A_N>;
        S = S - A_N; }
     N = N - 1; } }
```