Common DP Problems

1 Knapsack Problem

1.1 Problem

Given n items, labeled item $0 \ldots n-1$. Let v_j denote the value of item j and w_j denote the weight of item j. Suppose we have a bag that can hold up to weight W. Which items should we put into the bag to maximize the sum of the values of the items in the bag?

1.2 Solution

In this fairly standard DP problem, the state and recurrence relation is not hard to see. We can describe the state by the tuple (item we are currently considering, how much more weight can the bag hold). The recurrence relation essentially deciding whether to put the item we are currently considering into the bag or not:

$$f(i, w) = \min\{f(i+1, w), f(i+1, w - w_i)\}$$

2 Subset Sum

2.1 Problem

Given a set $S = \{x_1, \ldots, x_n\}$ of positive integers, does there exists a subset $T \subset S$ such that the sum of all integers in T is t?

2.2 Solution

We will construct an iterative DP solution. Let `memo[i][s]` denote whether we can find a subset of $\{x_1, \ldots, x_i\}$ whose sum is equal to s. The base case is: `memo[0][0] = true` and for $1 \leq i \leq n$, `memo[0][i] = false`. The recurrence relation is then

`memo[i][s] = memo[i-1][s] || memo[i-1][s - x_n]`

Now note that the first dimension of the `memo[][]` is actually redundant, since `memo[i][*]` only depends on `memo[i-1][*]`. Recall from the last lecture that this means we can reduce one of the dimension to yield the following code:

```
bool subset_sum(int t) {
  int memo[t+1];

  // initialize to the base case
  memset(memo, false, sizeof(memo));
  memo[0] = true;

  // iterative dp
  for (int i = 1; i <= n; ++i) {
    for (int j = t; j >= x[i]; --j) {
```

```
        memo[j] |= memo[j - x[i]];
      }
    }

    return memo[t];
}
```

2.3 Final Remarks

In the above code, we exploited the knowledge that S only contains positive integer, so the memo array only need the range $[0 \dots t]$. The more general version of the subset sum does not limit S contain only positive integers. In this case, the range of the memo array must encompass the range $[N \dots P]$ where N is the sum of all negative integers in S and P is the sum of all positive integers in S. Of course, in actual implementation, we must shift this since we can't use negative number to index into an array (alternatively, you can use a map, which is slower).

Also, note that in the above code, we actually know whether there exists a subset whose sum is any number in the range $[0 \dots t]$. So if S is not changed in the problem and only t is varied, we can actually pre-calculate the memo array first. This way, we can return the answer in constant time per query.

3 Longest Increasing Subsequence

3.1 Problem

Given a sequence of integers $x_1, x_2, \dots, x_n$, what is the length of the longest (strictly) increasing subsequence? A sequence is increasing if $\forall i < j, x_i < x_j$.

3.2 Naive Solution

Let $f(i)$ denote the longest increasing subsequence of the sequence $x_1, x_2, \dots, x_i$ ending with x_i. Then the recurrence relation is specified by $\max_j \{1 + f(j)\}$, where j is iterated over all $1 \leq j < i$ and $x_j \leq x_i$. This has the runtime of $O(n^2)$.

3.3 A Smarter Solution

We construct an iterative solution. We iterate through the sequence and for each element x_i, we keep track of an array `memo[k]` which denotes the index $j \leq i$ such that there exists an increasing subsequence of length k ending with the integer x_j. If there are multiple such indices, we will choose the index such that x_j is the smallest. If there is still a tie, then we choose the smaller j. Note that by this construction, if $k < l$, then $x_{memo[k]} < x_{memo[l]}$. The base case is `memo[1] = 0`.

When we are considering x_{i+1}, then the length of the longest increasing subsequence of ending in x_{i+1} is l where $x_{memo[l-1]} < x_{i+1} \leq x_{memo[l]}$ (ie l is the largest index such that $x_{memo[l-1]} < x_{i+1}$). How do we find l efficiently? The answer is binary search! The algorithm is best illustrated by an example. Suppose we are considering the sequence [2, 1, 3, 3, 5, 4].

i	x[i]	l	memo[1]	memo[2]	memo[3]	memo[4]	memo[5]	memo[6]	L
1	1	1	1	∞	∞	∞	∞	∞	1
2	3	2	1	2	∞	∞	∞	∞	2
3	3	3	1	2	∞	∞	∞	∞	2
4	5	3	1	2	4	∞	∞	∞	3
5	4	3	1	2	5	∞	∞	∞	3

Here is the code for LIS:

```
// need to define a custom comparator
bool mycmp(const int& a, const int& b) {
  return x[a] < x[b];
}

int LIS() {
  int L = 1;
  int memo[64]; // make sure the array is large enough
  memset(memo, 0x3f, sizeof(memo));
  memo[1] = 0;

  // note that the first element is x[0]
  for (i = 1; i < n; ++i) {
    int l = upper_bound(memo, memo + L + 1, i, mycmp) - memo;
    if (l > L && x[memo[l-1]] < x[i])
      memo[l] = i, L = l;
    else if (l <= L && (l == 1 || x[memo[l-1]]))
      memo[l] = i;
  }
  return L;
}
```