Perfect-Information Blackjack

Let's remember what we covered in lecture. Suppose that we want to play Blackjack against a dealer (with no other players). Suppose, in addition, that we have x-ray vision that allows us to see the entire deck $(c_0, c_1, ..., c_{n-1})$. As in a casino, the dealer will use a fixed strategy that we know ahead of time (stand-on-17), and that we area allowed to make \$1 bets (so with each round, we can either win \$1, lose \$1, or tie and make no profits and no losses). How do we maximize our winnings in this game? When should we hit or stand?

Let's use dynamic programming to try to come up with the best sequence of moves for us to make so that we will be able to earn as much as we can. Our approach will be as follows. We want to guess when to hit and when to stand for some suffix of cards starting at index i.

Let i be the number of cards we've already played, and $\texttt{BJ}(i)$ the best play using the remaining cards $(c_i, c_{i+1}, ..., c_{n-1})$. These $\texttt{BJ}(i)$s will be our subproblems. How many of them are there? Since i spans from 0 to n, we have $O(n)$ subproblems.

What are our guesses? We have to guess how many times the player hits (requests another card). There are $O(n)$ choices for this guess. Here's a detailed recurrence:

```
1  BJ(i):
2          if n-1 < 4: return 0, since there are not enough cards
3          for  p in range(2, n-i-2): # number of cards taken
4                  # player's cards by deal order (player, then dealer, then player)
5                  player = sum(c_i, c_{i+2}, c_{i+4:i+p+2})
6                  if player > 21: # bust
7                          options.append(-1 + BJ(i+p+2))
8                          break
9                  for d in range(2, n-i-p):
10                         dealer = sum(c_{i+1}, c_{i+3}, c_{i+p+2:i+p+d})
11                         if dealer >= 17: break
12                 if dealer > 21: dealer = 0 # bust
13                 options.append(cmp(player, dealer) + BJ(i+p+d))
14         return max(options)
```

Finding the player and dealer sums each takes $O(n)$. Looping over p takes $O(n)$. Therefore, the whole thing takes $O(n^2)$ time.

We can also construct a DAG for this problem as we have in the past for other problems. The vertices will represent the subproblems and the edges will represent connections between subproblems. In this case, the edges will point to subproblems with a greater i value (as seen in the recurrence where we append to options). The edge weights will be the outcome (-1, 0, or 1). We will then try to find the longest path (to maximize our winnings).

Parent Pointers

To reconstruct the answer, any time we memoize the result to a subproblem, we also need to save some additional piece of information that would help us reconstruct the solution. There are various ways to do this. For example, instead of memoizing the result to a subproblem, we can memoize a pair (2-tuple) of the result and the parent pointer. As a result, whenever we use a memoized value, we have to take the 0th element (since the memoized value is now a pair of the answer to our desired subproblem and a parent pointer). This means that the answer to the original problem will also be a `(result, parent pointer)` pair. We can now loop until the parent pointer is `None`, beginning with the parent pointer that was just returned to us, and navigate down these parent pointers to reconstruct the result.

Longest Increasing Subsequence

The longest increasing subsequence problem is a problem where we want to find a longest subsequence of a given sequence in which the subsequence elements are in sorted order. Note that a subsequence can skip elements in the original sequence. We can solve this using dynamic programming as well.

Wikipedia talks about the Van der Corput sequence. This is a sequence that, for any real number in $[0, 1]$, contains a subsequence that converges to that number. It's constructed by reversing the base n representation of the sequence of natural numbers (1, 2, 3, ...). For example, in base 10, the sequence begins with 0.1, 0.2, 0.3, ..., 0.9, 0.01, 0.11, 0.21, ...

The binary Van der Corput sequence begins with
0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15, ...

Two longest-increasing subsequences in this sequence are
0, 2, 6, 9, 13, 15.
0, 4, 6, 9, 11, 15.

To solve this using dynamic programming, we want to go through the sequence in order, keeping track of the longest increasing subsequence found so far. Denote the element of the sequence $a_0, a_1, ..., a_{n-1}$. We will define b_i as the length of the longest subsequence ending with a_i. Now we can build each b_i as follows. Find all j such that $j < i$ and $a_j \leq a_i$, and collect them in a set S. Then, our desired value of b_i is $\max\{b_j | j \in S\} + 1$ if S is nonempty and 1 if S is empty. After we have found all n values of b, we simply choose the largest. We can use parent pointers to actually construct the subsequence of the elements of a itself.