CS61B Lecture 36

Wednesday, April 22, 2020

Dynamic Programming -- Maximizing Tree

This is something we covered in 61A, and we will touch on this briefly with respect to the concepts we learned in 61B.

Here's a puzzle from Dan Garcia:

- Start with a list with an even number of non-negative integers.
- Each player in turn takes either the leftmost number or the rightmost.
- Idea is to get the largest possible sum.

Example: starting with (6, 12, 0, 8), you (as first player) should take the 8. Whatever the second player takes, you also get the 12, for a total of 20. Assuming your opponent plays perfectly (i.e., to get as much as possible), how can you maximize your sum? We can solve this with exhaustive game-tree search.

```
int bestSum(int[] V) {
    int total, i, N = V.length;
    for (i = 0, total = 0; i < N; i += 1) {
        total += V[i];
    }
    return bestSum(V, 0, N-1, total);
}

/** The largest sum obtainable by the first player in the choosing
 *  game on the list V[LEFT .. RIGHT], assuming that TOTAL is the
 *  sum of all the elements in V[LEFT .. RIGHT]. */
int bestSum(int[] V, int left, int right, int total) {
    if (left > right) { return 0; }
    else {
        int L = total - bestSum(V, left+1, right, total-V[left]);
        int R = total - bestSum(V, left, right-1, total-V[right]);
        return Math.max(L, R);
    }
}
```

The time cost is

$$C(N) \in \Theta(2^N)$$

The problem is that we are recomputing intermediate results many times. The solution is to memoize intermediate results: here, we pass in an N x N array, where N is the length of V, initialized to -1.

```
int bestSum(int[] V, int left, int right, int total, int[][] memo) {
    if (left > right) { return 0; }
    else if (memo[left][right] == -1) {
        int L = total - bestSum(V, left+1, right, total-V[left], memo);
        int R = total - bestSum(V, left, right-1, total-V[right], memo);
        memo[left][right] = Math.max(L, R);
    }
    return memo[left][right];
}
```

Iterative Version

There is an iterative approach to this, which isn't really necessary unless it is of utmost importance to save space:

```
int bestSum(int[] V) {
    int[][] memo = new int[V.length][V.length];
    int[][] total = new int[V.length][V.length];
    for (int i = 0; i < V.length; i += 1) {
        memo[i][i] = total[i][i] = V[i];
    }
    for (int k = 1; k < V.length; k += 1) {
        for (int i = 0; i < V.length-k-1; i += 1) {
            total[i][i+k] = V[i] + total[i+1][i+k];
            int L = total[i][i+k] - memo[i+1][i+k];
            int R = total[i][i+k] - memo[i][i+k-1];
            memo[i][i+k] = Math.max(L, R);
        }
    }
    return memo[0][V.length-1];
}
```

The problem with this is that we need to figure out ahead of time the order in which the memoized version will fill in memo, and write an explicit loop.

Longest Common Subsequence

Find length of the longest string that is a subsequence of each of two other strings. For example, the longest common subsequence of “sally sells sea shells by the seashore” and “sarah sold salt sellers at the salt mines” is “sa sl sa sells the sae” (length 23).

Here is an obvious recursive algorithm:

```
/** Length of longest common subsequence of S0[0..k0-1]
* and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
    if (k0 == 0 || k1 == 0) { return 0; }
    if (S0[k0-1] == S1[k1-1]) { return 1 + lls(S0, k0-1, S1, k1-1); }
    else { return Math.max(lls(S0, k0-1, S1, k1), lls(S0, k0, S1, k1-1); }
}
```

This is exponential growth, but it's obvious memoizable:

```
/** Length of longest common subsequence of S0[0..k0-1]
```

```
* and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
int[][] memo = new int[k0+1][k1+1];
    for (int[] row : memo) { Arrays.fill(row, -1); }
    return lls(S0, k0, S1, k1, memo);
}

private static int lls(String S0, int k0, String S1, int k1, int[][] memo) {
    if (k0 == 0 || k1 == 0) { return 0; }
    if (memo[k0][k1] == -1) {
        if (S0[k0-1] == S1[k1-1]) {
            memo[k0][k1] = 1 + lls(S0, k0-1, S1, k1-1, memo);
        } else {
            memo[k0][k1] = Math.max(lls(S0, k0-1, S1, k1, memo), lls(S0, k0, S1,
k1-1, memo));
        }
    }
    return memo[k0][k1];
}
```

This brings the time complexity down to

$$\Theta(k_0 \bullet k_1)$$

where `k0` and `k1` are the lengths of `String`s `s0` and `s1`.

Enumeration Types in Java

If we wanted a type that the user can produce whose sole purpose has several different, discrete values. In the purest form, the only necessary operations are `==` and `!=`. The only property of a value of the type that it differs from all others. In older versions of Java, we used named integer constants.

```
interface Pieces {
    static final int BLACK_PIECE = 0,
                     BLACK_KING = 1,
                     WHITE_PIECE = 2,
                     WHITE_KING = 3,
                     EMPTY = 4;
}
```

C and C++ provide enumeration types as shorthand, with syntax like this:

```
enum Piece { BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY };
```

But since all these values are basically `int`s, accidents can happen.

New versions in Java allows syntax like that of C or C++, but with more guarantees, defining `Piece` to be a new reference type, or a special kind of class type.

```
public enum Piece {
    BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY
}
```

The names `BLACK_PIECE` and such are final enumeration constants, or enumerals, of type `Piece`.

They are automatically initialized, and are the only values of the enumeration types that exist (it is illegal to use `new` to create an enum value). They are static members of the class, so we must call them using the class name, e.g. `Piece.BLACK_PIECE`, or import them with a static import.

We can safely use `==`, and also `switch` statements:

```
boolean isKing(Piece p) {
    switch (p) {
    case BLACK_KING:
    case WHITE_KING:
        return true;
    default:
        return false;
    }
}
```

The order of declaration of enumeration constants is significant: `.ordinal()` gives the position (indexed from 0) of an enumeration value. Thus, `Piece.BLACK_KING.ordinal()` is 1.

The array `Piece.values()` gives all the possible values of the type. Thus, you can write:

```
for (Piece p : Piece.values()) {
    System.out.println("Piece value #%d is %s%n", p.ordinal(), p);
}
```

The static function `Piece.valueOf()` converts a String into a value of type Piece. So, `Piece.valueOf("EMPTY") == EMPTY`.

Fancy Enum Types

Enums are classes, so we can define all the extra fields, methods, and constructors that we want. Constructors are only used in creating enumeration constants. The constructor arguments follow the constant name.

```
enum Piece {
    BLACK_PIECE(BLACK, false, "b"), BLACK_KING(BLACK, true, "B"),
    WHITE_PIECE(WHITE, false, "w"), WHITE_KING(WHITE, true, "W"),
    EMPTY(null, false, " ");

    private final Side color;
    private final boolean isKing;
    private final String textName;
    Piece(Side color, boolean isKing, String textName) {
        this.color = color;
        this.isKing = isKing;
        this.textName = textName;
    }

    Side color() { return color; }
    boolean isKing() { return isKing; }
    String textName() { return textName; }
}
```