CS61B Lecture #36

Today:

• Dynamic Programming

• A Brief Side Trip: Enumeration types.
Dynamic Programming

• A puzzle (D. 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?

• Can solve this with exhaustive game-tree search.
Obvious Program

• Recursion makes it easy, again:

```java
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);
    }
}
```

• Time cost is $C(0) = 1$, $C(N) = 2C(N - 1)$; so $C(N) \in \Theta(2^N)$
Still Another Idea from CS61A

• The problem is that we are recomputing intermediate results many times.

• Solution: memoize the intermediate results. Here, we pass in an $N \times N$ array ($N = V$.length) of memoized results, initialized to -1.

```java
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];
}
```

• Now the number of recursive calls to bestSum must be $O(N^2)$, for $N$ = the length of $V$, an enormous improvement from $\Theta(2^N)$!
Iterative Version

- I prefer the recursive version, but the usual presentation of this idea—known as dynamic programming—is iterative:

```java
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];
}
```

- That is, we figure out ahead of time the order in which the memoized version will fill in `memo`, and write an explicit loop.

- Save the time needed to check whether result exists.

- But I say, why bother unless it’s necessary to save space?
Longest Common Subsequence

- **Problem:** Find length of the longest string that is a subsequence of each of two other strings.

- **Example:** 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)

- Similarity testing, for example.

- **Obvious recursive algorithm:**

```java
/** 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));
}
```

- Exponential, but obviously memoizable.
Memoized Longest Common Subsequence

/** 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];
}

Q: How fast will the memoized version be?
Memoized Longest Common Subsequence

/** 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];
}

Q: How fast will the memoized version be? \( \Theta(k_0 \cdot k_1) \)
Side Trip into Java: Enumeration Types

- Problem: Need a type to represent something that has a few, named, discrete values.
- In the purest form, the only necessary operations are `==` and `!=`; the only property of a value of the type is that it differs from all others.
- In older versions of Java, used named integer constants:

```java
interface Pieces {
    int BLACK PIECE = 0, // Fields in interfaces are static final.
    BLACK KING = 1,
    WHITE PIECE = 2,
    WHITE KING = 3,
    EMPTY = 4;
}
```

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

```java
enum Piece { BLACK PIECE, BLACK KING, WHITE PIECE, WHITE KING, EMPTY };
```

- But since all these values are basically ints, accidents can happen.
Enum Types in Java

- New version of Java allows syntax like that of C or C++, but with more guarantees:

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

- Defines `Piece` as a new reference type, a special kind of class type.

- The names `BLACK_PIECE`, etc., are static, final *enumeration constants* (or *enumerals*) of type `PIECE`.

- They are automatically initialized, and are the only values of the enumeration type that exist (illegal to use `new` to create an enum value.)

- Can safely use `==`, and also `switch` statements:

  ```java
  boolean isKing(Piece p) {
      switch (p) {
          case BLACK_KING: case WHITE_KING: return true;
          default: return false;
      }
  }
  ```
Making Enumerals Available Elsewhere

- Enumerals like `BLACK_PIECE` are static members of a class, not classes.
- Therefore, unlike C or C++, their declarations are not automatically visible outside the enumeration class definition.
- So, in other classes, must write `Piece.BLACK_PIECE`, which can get annoying.
- However, with version 1.5, Java has `static imports`: to import all static definitions of class `checkers.Piece` (including enumerals), you write

  ```java
  import static checkers.Piece.*;
  ```

  among the import clauses.
- Alas, `cannot` use this for enum classes in the anonymous package.
Operations on Enum Types

- **Order of declaration of enumeration constants significant**: `.ordinal()` gives the position (numbering 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:

  ```java
  for (Piece p : Piece.values())
      System.out.printf("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. You can define all the extra fields, methods, and constructors you want.
- Constructors are used only in creating enumeration constants. The constructor arguments follow the constant name:

```java
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, " ");
}
```

```java
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; }
```