

Lecture #39

- **Today:** Dynamic programming and memoization.

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:

```
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.

```
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 =
            V[left] + total - bestSum (V, left+1, right, total-V[left],memo);
        int R =
            V[right] + 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:

```
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 = V[i] + total[i+1][i+k] - memo[i+1][i+k];
            int R = V[i+k] + total[i][i+k-1] - 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?

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:

```
/** 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? $\Theta(k_0 \cdot k_1)$