

Lecture #39: Finding a Substring

- **Problem:** Where does pattern string P (of length m) occur in text string T (of length n)?

- In interesting cases $n \gg m$.

- Obvious solution:

```
for (int i = 0; i <= n - m; i += 1)
    if (P.equals T[i .. i+m-1])
        return i;
return -1;
```

- How long does this take?
- Why does it take so long? A: Poor use of information leads to large amounts of redundant checking.

Knuth-Morris-Pratt

- **Observation:** If we have matched substring $P[0..j-1]$, $j > 0$, to text $T[i..i+j-1]$, but $P[j] \neq T[i+j]$, then there is no point in checking whether P matches $T[i+1..i+m]$ unless first $P[0..j-2]$ matches $P[1..j-1] = T[i+1..i+j-1]$.
- Likewise, for $j > 1$, no point in checking P against $T[i+2..i+2+m]$ unless first $P[0..j-3]$ matches $P[2..j-1]$, etc.
- This determination depends *only on P*. Therefore, we can *precompute* what substring checks we can skip.
- Leads to this program (Warning: pseudo-Java!):

```
if (m == 0) return 0; // Empty string is everywhere.
i = j = 0;
while (i-j < n-m+1) {
    if      (P[j] == T[i] && j == m-1) { return i - m + 1; }
    else if (P[j] == T[i])             { i += 1; j += 1; }
    else if (j == 0)                   { i += 1; }
    else { j = largest k<j such that P[0..k-1].equals (P[m-k..j-1]); }
}
return -1;
```

Knuth-Morris-Pratt, Analysis

- The quantity “largest $k < j$ such that...” depends only on P and j , so we can precompute an array of values for it indexed by j , called the failure table.
- At each iteration, neither i nor $i - j$ decrease.
- Furthermore, either i increases by 1 or $i - j$ increases by at least one.
- Since the maximum value of i and $i - j$ is n , the goes around at most $2n$ times.
- That is, once you do some precomputation on P , the actual matching takes $O(n)$ time, regardless of m , the length of P .
- Book shows modification of the basic program that allows you to compute the failure table in $O(m)$ time, for a total time of $O(m + n)$.

New Topic: 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 search.

Obvious Program

- Recursion makes it easy, again:

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

- Unfortunately, the time cost for this, as a function of N , the length of V , is $C(0) = 1$, $C(N) = 2C(N - 1)$ —exponential, in other words.

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];
}
```

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];
}
```

Example from Book: Longest Common Subsequence

- **Problem:** Find length of the longest string that is a subsequence of each of two other strings.
- Similarity testing, for example.
- Obvious recursive algorithm:

```
/** Length of longest common subsequence of S0[0..k0-1]
 * and S1[0..k1-1] (pseudo Java) */
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 (exercise to reader).