4
Dynamic Programming

Please read the 15-451 lecture notes on dynamic programming for the basic concepts, of top-down dynamic programming (or memoization), and bottom-up dynamic programming. (It also talks about dynamic programming on trees, etc.) These notes here are focused on the issues of reducing space usage for these DPs.

4.1 Longest Common Subseqeuence

Here is the naive bottom-up dynamic program to find the longest common subseqeuence (LCS) of two strings S and T. Define M to be a table with $m+1$ rows and $n+1$ columns, where $M(i,j)$ computes the length of the longest common subsequence of the prefixes $S_{1:i}$ and $T_{1:j}$.

Algorithm 6: LCS-value(S, T)

$M(0,\star) = M(\star,0) = 0$
for $i = 1$ ***to*** m **do**
  for $j = 1$ ***to*** n **do**
    if $S_i = T_j$ **then**
      $M(i,j) \leftarrow 1 + M(i-1, j-1)$
    else
      $M(i,j) \leftarrow \max(M(i-1,j), M(i,j-1))$
return $M(m,n)$

Theorem 4.1. *Algorithm 6 computes the length of the longest common subsequence of two strings of length m, n in $O(mn)$ time and space.*

```
[0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 1 1 1 1 1 1 1 1]
[0 1 1 1 1 1 1 2 2 2 2]
[0 1 1 1 1 1 1 2 3 3 3]
[0 1 1 2 2 2 2 2 3 3 3]
[0 1 2 2 3 3 3 3 3 4 4]
[0 1 2 2 3 3 3 4 4 4 4]
[0 1 2 2 3 3 4 4 4 5 5]
[0 1 2 2 3 3 4 4 4 5 6]
```

Figure 4.1: The LCS of ACCTACAG and CATATACCAG.

4.1.1 Finding the LCS Itself

Having run Algorithm 6 to fill in the table, we can find the LCS itself in $O(m+n)$ time by just "following the decisions" when filling the table.

Algorithm 7: LCS-Search(S, T)

$i \leftarrow m, j \leftarrow n$
while $i > 0$ ***or*** $j > 0$ **do**
  if $S_i = T_j$ **then**
    output S_i
    $i \leftarrow i - 1, j \leftarrow j - 1$
  else
    if $M(i, j) = M(i - 1, j)$ **then** $i \leftarrow i - 1$ **else** $j \leftarrow j - 1$

(Exercise: One of the strings T has been accidentally deleted, but you still have the string S, and the table $M(\cdot, \cdot)$. Show how to output the LCS in $O(m + n)$ time)

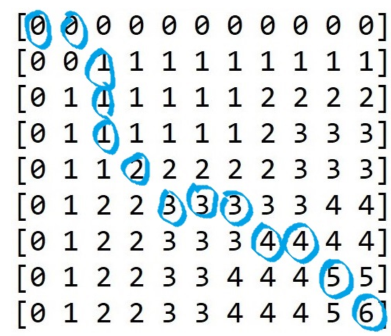

Figure 4.2: The LCS of ACCTACAG and CATATACCAG is ATACAG.

4.2 *Space-Efficiency*

The above bottom-up algorithm for the LCS problem always takes $O(mn)$ time and space. A very recent result shows that the quadratic runtime is necessary in general, but we can reduce the space usage. The crucial observations are simple: (a) we care only about the value of $M(m, n)$, and (b) the update rule for a cell $M(i, j)$ depends only on $M(i - 1, j - 1)$, $M(i - 1, j)$ and $M(i, j - 1)$, which belong to the same row or previous row as the current cell (i, j) being filled in. Hence we can fill the table row-by-row, "keeping in mind" only rows $i - 1$ and i when filling in row i. Formally, we define the table M to have only 2 rows and $n + 1$ columns, and change the algorithm as follows:

Algorithm 8: Low-Space LCS(S, T)

$M(0, \star) = M(\star, 0) = 0$
for $i = 1$ ***to*** m **do**
  for $j = 1$ ***to*** n **do**
    if $S_i = T_j$ **then**
      $M(i \bmod 2, j) \leftarrow 1 + M(i - 1 \bmod 2, j - 1)$
    else
      $M(i \bmod 2, j) \leftarrow$
       $\max(M(i - 1 \bmod 2, j), M(i \bmod 2, j - 1))$
return $M(m \bmod 2, n)$

Theorem 4.2. *Algorithm 8 computes the length of the longest common subsequence of two strings of length m, n in $O(mn)$ time and $O(\min(m, n))$ space*

4.3 (Optional) Finding the LCS in Linear Space

How can we find the actual LCS using $O(m+n)$ space: clearly the search algorithm given in Algorithm 7 will no longer work, since we don't have the entire table. Hence we need to be smarter: the lovely idea here can be called "guess the mid-point".

This idea is essentially that used by Savitch for his classical result relating log-space computation to non-deterministic log-space.

The main observation is this: there exists a value q such that

$$LCS(S_{1:m}, T_{1:n}) = LCS(S_{1:m/2}, T_{1,q}) + LCS(S_{m/2+1,m}, T_{q+1,n}). \quad (4.1)$$

I visualize this as follows: when we follow the optimal solution up from $M(m,n)$ to $M(0,0)$, this optimal solution must cross row $m/2$ at some point—this point $(m/2, q)$ must provide this partition. Add a picture here.

Now using Algorithm 8 on $S_{1:m/2}$ and T, and on the *reversed* strings $S_{m/2+1,m}$ and T, we can find the index q that achieves the equality (4.1). Now we can recurse on the two halves

---

Algorithm 9: Low-Space LCS-Search(S, T)

9.1 run Algorithm 8 on $S_{1:m/2}$ and T, and on reversed $S_{m/2+1,m}$ and T

9.2 find q that satisfies equality (4.1)

9.3 recurse on $S_{1:m/2}, T_{1:q}$, and on $S_{m/2+1,m}, T_{q+1,n}$.

Theorem 4.3. *Algorithm 9 runs in time $O(mn)$ and space $O(m+n)$.*

Proof. For the runtime, note that the first line of the algorithm runs in $O(mn)$ time, using Theorem 4.2. Now a linear-time scan can find the value q that minimizes the sum $LCS(S_{1:m/2}, T_{1,q}) + LCS(S_{m/2+1,m}, T_{q+1,n})$. Now for the inductive proof, assume that the rumtime of the recursive calls is at most $c(m/2)q + c(m/2)(n-q) = c(m/2)n$. Summing this all up, we get at most cmn. □