Ninety percent of science fiction is crud.
But then, ninety percent of everything is crud,
and it's the ten percent that isn't crud that is important.

— [Theodore] Sturgeon's Law (1953)

⋆6   Advanced Dynamic Programming Tricks

Dynamic programming is a powerful technique for efficiently solving recursive problems, but it's hardly the end of the story. In many cases, once we have a basic dynamic programming algorithm in place, we can make further improvements to bring down the running time or the space usage. We saw one example in the Fibonacci number algorithm. Buried inside the naïve iterative Fibonacci algorithm is a recursive problem—computing a power of a matrix—that can be solved more efficiently by dynamic programming techniques—in this case, repeated squaring.

6.1   Saving Space: Divide and Conquer

Just as we did for the Fibonacci recurrence, we can reduce the space complexity of our edit distance algorithm from $O(mn)$ to $O(m+n)$ by only storing the current and previous rows of the memoization table. This 'sliding window' technique provides an easy space improvement for most (but *not* all) dynamic programming algorithm.

Unfortunately, this technique seems to be useful only if we are interested in the *cost* of the optimal edit sequence, not if we want the optimal edit sequence itself. By throwing away most of the table, we apparently lose the ability to walk backward through the table to recover the optimal sequence.

Fortunately for memory-misers, in 1975 Dan Hirshberg discovered a simple divide-and-conquer strategy that allows us to compute the optimal edit sequence in $O(mn)$ time, using just $O(m+n)$ space. The trick is to record not just the edit distance for each pair of prefixes, but also a single position in the middle of the optimal editing sequence for that prefix. Specifically, any optimal editing sequence that transforms $A[1..m]$ into $B[1..n]$ can be split into two smaller editing sequences, one transforming $A[1..m/2]$ into $B[1..h]$ for some integer h, the other transforming $A[m/2+1..m]$ into $B[h+1..n]$.

To compute this breakpoint h, we define a second function $Half(i,j)$ such that some optimal edit sequence from $A[1..i]$ into $B[1..j]$ contains an optimal edit sequence from $A[1..m/2]$ to $B[1..Half(i,j)]$. We can define this function recursively as follows:

$$Half(i,j) = \begin{cases} \infty & \text{if } i < m/2 \\ j & \text{if } i = m/2 \\ Half(i-1,j) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i-1,j)+1 \\ Half(i,j-1) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i,j-1)+1 \\ Half(i-1,j-1) & \text{otherwise} \end{cases}$$

(Because there there may be more than one optimal edit sequence, this is not the only correct definition.) A simple inductive argument implies that $Half(m,n)$ is indeed the correct value of h. We can easily modify our earlier algorithm so that it computes $Half(m,n)$ at the same time as the edit distance $Edit(m,n)$, all in $O(mn)$ time, using only $O(m)$ space.

Edit		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Half		A	L	G	O	R	I	T	H	M
	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
A	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
T	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
R	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
U	0	1	2	3	4	5	6	7	8	9
I	0	1	2	3	4	5	5	5	5	5
S	0	1	2	3	4	5	5	5	5	5
T	0	1	2	3	4	5	5	5	5	5
I	0	1	2	3	4	5	5	5	5	5
C	0	1	2	3	4	5	5	5	5	5

Finally, to compute the optimal editing sequence that transforms A into B, we recursively compute the optimal sequences transforming $A[1..m/2]$ into $B[1..Half(m, n)]$ and transforming $A[m/2 + 1..m]$ into $B[Half(m, n) + 1..n]$. The recursion bottoms out when one string has only constant length, in which case we can determine the optimal editing sequence in linear time using our old dynamic programming algorithm. The running time of the resulting algorithm satisfies the following recurrence:

$$T(m, n) = \begin{cases} O(n) & \text{if } m \leq 1 \\ O(m) & \text{if } n \leq 1 \\ O(mn) + T(m/2, h) + T(m/2, n - h) & \text{otherwise} \end{cases}$$

It's easy to prove inductively that $T(m, n) = O(mn)$, no matter what the value of h is. Specifically, the entire algorithm's running time is at most twice the time for the initial dynamic programming phase.

$$\begin{aligned} T(m, n) &\leq \alpha mn + T(m/2, h) + T(m/2, n - h) \\ &\leq \alpha mn + 2\alpha mh/2 + 2\alpha m(n - h)/2 \qquad \text{[inductive hypothesis]} \\ &= 2\alpha mn \end{aligned}$$

A similar inductive argument implies that the algorithm uses only $O(n + m)$ space.

Hirschberg's divide-and-conquer trick can be applied to almost any dynamic programming problem to obtain an algorithm to construct an optimal *structure* (in this case, the cheapest edit sequence) within the same space and time bounds as computing the *cost* of that optimal structure (in this case, edit distance). For this reason, we will almost always ask you for algorithms to compute the cost of some optimal structure, not the optimal structure itself.

6.2   Saving Time: Sparseness

In many applications of dynamic programming, we are faced with instances where almost every recursive subproblem will be resolved exactly the same way. We call such instances *sparse*. For example, we might want to compute the edit distance between two strings that have few characters in common, which means there are few "free" substitutions anywhere in the table. Most of the table has exactly the same structure. If we can reconstruct the entire table from just a few key entries, then why compute the entire table?

To better illustrate how to exploit sparseness, let's consider a simplification of the edit distance problem, in which substitutions are not allowed (or equivalently, where a substitution counts as two operations instead of one). Now our goal is to maximize the number of "free" substitutions, or equivalently, to find the *longest common subsequence* of the two input strings.

Let $LCS(i, j)$ denote the length of the longest common subsequence of two fixed strings $A[1 .. m]$ and $B[1 .. n]$. This function can be defined recursively as follows:

$$LCS(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i - 1, j - 1) & \text{if } A[i] = B[j] \\ \max\{LCS(i, j - 1),\ LCS(i - 1, j)\} & \text{otherwise} \end{cases}$$

This recursive definition directly translates into an $O(mn)$-time dynamic programming algorithm.

Call an index pair (i, j) a *match point* if $A[i] = B[j]$. In some sense, match points are the only 'interesting' locations in the memoization table; given a list of the match points, we could easily reconstruct the entire table.

The *LCS* memoization table for ALGORITHMS and ALTRUISTIC; the brackets « and » are sentinel characters.

More importantly, we can compute the *LCS* function directly from the list of match points using the following recurrence:

$$LCS(i, j) = \begin{cases} 0 & \text{if } i = j = 0 \\ \max\{LCS(i', j') \mid A[i'] = B[j'] \text{ and } i' < i \text{ and } j' < j\} + 1 & \text{if } A[i] = B[j] \\ \max\{LCS(i', j') \mid A[i'] = B[j'] \text{ and } i' \leq i \text{ and } j' \leq j\} & \text{otherwise} \end{cases}$$

(Notice that the inequalities are strict in the second case, but not in the third.) To simplify boundary issues, we add unique sentinel characters $A[0] = B[0]$ and $A[m + 1] = B[n + 1]$ to both strings. This ensures that the sets on the right side of the recurrence equation are non-empty, and that we only have to consider match points to compute $LCS(m, n) = LCS(m + 1, n + 1) - 1$.

If there are K match points, we can actually compute them all in $O(m \log m + n \log n + K)$ time. Sort the characters in each input string, but remembering the original index of each character, and then essentially merge the two sorted arrays, as follows:

```
FINDMATCHES(A[1..m], B[1..n]):
    for i ← 1 to m:  I[i] ← i
    for j ← 1 to n:  J[j] ← j

    sort A and permute I to match
    sort B and permute J to match

    i ← 1;  j ← 1
    while i < m and j < n
        if A[i] < B[j]
            i ← i + 1
        else if A[i] > B[j]
            j ← j + 1
        else                    《Found a match!》
            ii ← i
            while A[ii] = A[i]
                jj ← j
                while B[jj] = B[j]
                    report (I[ii], J[jj])
                    jj ← jj + 1
                ii ← i + 1
            i ← ii;  j ← jj
```

To efficiently evaluate our modified recurrence, we once again turn to dynamic programming. We consider the match points in lexicographic order—the order they would be encountered in a standard row-major traversal of the $m \times n$ table—so that when we need to evaluate $LCS(i, j)$, all match points (i', j') with $i' < i$ and $j' < j$ have already been evaluated.

```
SPARSELCS(A[1..m], B[1..n]):
    Match[1..K] ← FINDMATCHES(A, B)
    Match[K + 1] ← (m + 1, n + 1)       《Add end sentinel》
    Sort M lexicographically
    for k ← 1 to K
        (i, j) ← Match[k]
        LCS[k] ← 1                      《From start sentinel》
        for ℓ ← 1 to k − 1
            (i', j') ← Match[ℓ]
            if i' < i and j' < j
                LCS[k] ← min{LCS[k], 1 + LCS[ℓ]}
    return LCS[K + 1] − 1
```

The overall running time of this algorithm is $O(m \log m + n \log n + K^2)$. So as long as $K = o(\sqrt{mn})$, this algorithm is actually faster than naïve dynamic programming.

6.3 Saving Time: Monotonicity

> The SMAWK matrix-searching algorithm is a better example here; the problem is more general, the algorithm is simple, and the proof is self-contained. Next time!

Recall the optimal binary search tree problem from the previous lecture. Given an array $F[1..n]$ of access frequencies for n items, the problem it to compute the binary search tree that minimizes the cost of all accesses. A relatively straightforward dynamic programming algorithm solves this problem in $O(n^3)$ time.

As for longest common subsequence problem, the algorithm can be improved by exploiting some structure in the memoization table. In this case, however, the relevant structure isn't in the table of costs,

but rather in the table used to reconstruct the actual optimal tree. Let $OptRoot[i, j]$ denote the index of the root of the optimal search tree for the frequencies $F[i .. j]$; this is always an integer between i and j. Donald Knuth proved the following nice monotonicity property for optimal subtrees: If we move either end of the subarray, the optimal root moves in the same direction or not at all. More formally:

$$OptRoot[i, j-1] \leq OptRoot[i, j] \leq OptRoot[i+1, j] \text{ for all } i \text{ and } j.$$

This (nontrivial!) observation leads to the following more efficient algorithm:

```
FASTEROPTIMALSEARCHTREE(f[1..n]):
    INITF(f[1..n])
    for i ← 1 downto n
        OptCost[i, i − 1] ← 0
        OptRoot[i, i − 1] ← i
    for d ← 0 to n
        for i ← 1 to n
            COMPUTECOSTANDROOT(i, i + d)
    return OptCost[1, n]
```

```
COMPUTECOSTANDROOT(i, j):
    OptCost[i, j] ← ∞
    for r ← OptRoot[i, j − 1] to OptRoot[i + 1, j]
        tmp ← OptCost[i, r − 1] + OptCost[r + 1, j]
        if OptCost[i, j] > tmp
            OptCost[i, j] ← tmp
            OptRoot[i, j] ← r
    OptCost[i, j] ← OptCost[i, j] + F[i, j]
```

It's not hard to see that the loop index r increases monotonically from 1 to n during each iteration of the *outermost* for loop of FASTEROPTIMALSEARCHTREE. Consequently, the total cost of all calls to COMPUTECOSTANDROOT is only $O(n^2)$.

If we formulate the problem slightly differently, this algorithm can be improved even further. Suppose we require the optimum *external* binary tree, where the keys $A[1..n]$ are all stored at the leaves, and intermediate pivot values are stored at the internal nodes. An algorithm discovered by Ching Hu and Alan Tucker[1] computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

Exercises

1. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K^2)$ time, where K is the number of match points. *[Hint: Use the FINDMATCHES algorithm on page 3 as a subroutine.]*

2. (a) Describe an algorithm to compute the *longest increasing subsequence* of a string $X[1..n]$ in $O(n \log n)$ time.

    (b) Using your solution to part (a) as a subroutine, describe an algorithm to compute the longest common subsequence of two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points.

3. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points. *[Hint: Combine your answers for problems 1 and 2(b).]*

---

[1]T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Applied Math.* 21:514–532, 1971. For a slightly simpler algorithm with the same running time, see A. M. Garsia and M. L. Wachs, A new algorithms for minimal binary search trees, *SIAM J. Comput.* 6:622–642, 1977. The original correctness proofs for both algorithms are rather intricate; for simpler proofs, see Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science*, 180:309-324, 1997.

4. Let T be an arbitrary rooted tree, where each vertex is labeled with a positive integer. A subset S of the nodes of T is *heap-ordered* if it satisfies two properties:

   - S contains a node that is an ancestor of every other node in S.
   - For any node v in S, the label of v is larger than the labels of any ancestor of v in S.

A heap-ordered subset of nodes in a tree.

   (a) Describe an algorithm to find the largest heap-ordered subset S of nodes in T that has the heap property in $O(n^2)$ time.

   (b) Modify your algorithm from part (a) so that it runs in $O(n \log n)$ time when T is a linked list. *[Hint: This special case is equivalent to a problem you've seen before.]*

   *(c) Describe an algorithm to find the largest subset S of nodes in T that has the heap property, in $O(n \log n)$ time. *[Hint: Find an algorithm to merge two sorted lists of lengths k and ℓ in $O(\log \binom{k+\ell}{k})$ time.]*