It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde, "A Few Maxims for the Instruction Of The Over-Educated" (1894)

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)

Dis-moi ce que tu manges, je te dirai ce que tu es.

— Jean Anthelme Brillat-Savarin, *Physiologie du Gout* (1825)

D

Advanced Dynamic Programming

[Read Chapter 3 first.]

Status: beta, except section D.2

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.

D.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, it is possible to compute the optimal edit sequence in $O(mn)$ time, using just $O(m+n)$ space, by combining dynamic programming with divide and conquer. I'll describe two such algorithms, the first proposed in 1975 Dan Hirschberg, and the second proposed by Rezaul Alam Chowdhury and Vijaya Ramachandran in 2005.

Variants of both of these divide-and-conquer strategies can be applied to almost any dynamic programming problem that looks for an optimal *path* in the dependency dag. The resulting algorithms return the actual path in the same time *and space* bounds as computing the *cost* of that optimal path. Consequently, when we develop dynamic programming algorithms to compute optimal structures, it almost always suffices to focus on the simpler problem of computing the *cost* of the optimal structure, rather than the optimal structure itself.

Hirschberg's algorithm

Hirschberg's main insight was to compute not only the edit distance for each pair of prefixes, but also a single position in the middle of the optimal edit sequence for those prefixes. Any optimal edit sequence that transforms $A[1..m]$ into $B[1..n]$ can be split into two smaller edit sequences, one transforming $A[1..m/2]$ into $B[1..h]$ and the other transforming $A[m/2+1..m]$ into $B[h+1..n]$, for some index h between 1 and n.

To compute this breakpoint index h, we define a second function $\mathit{Half}(i,j)$ 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 $B[1..\mathit{Half}(i,j)]$. We can define this function recursively as follows:

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

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

Finally, to compute the optimal sequence of edit operations that transforms A into B, we recursively compute the optimal sequences transforming $A[1..m/2]$ into $B[1..\mathit{Half}(m,n)]$ and transforming $A[m/2+1..m]$ into $B[\mathit{Half}(m,n)+1..n]$. The recursion bottoms out when one string has only constant length, in which case we can

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

Figure D.1. Memoization tables for *Edit* (showing an optimal edit path) and *Half*.

determine the optimal edit sequence in linear time using our old dynamic programming algorithm. The running time of the resulting algorithm satisfies the following recurrence:

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

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, or about four times the running time of the textbook dynamic programming algorithm to compute edit distance alone.

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

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

Chowdhury and Ramachandran's algorithm

Choudhury and Ramachandran pushed Hirschberg's divide-and-conquer idea further, developing another algorithm with the same $O(mn)$ time and $O(n+m)$ space bounds, but which in practice appears to be at least twice as fast on relatively small instances, and even faster on large instances because of improved cache performance.[1] Instead of

[1] In 2004, Joon-Sang Park, Michael Penner, and Viktor K. Prasanna proposed a similar divide-and-conquer strategy for improving the cache performance of the Floyd-Warshall all-pairs-shortest-path algorithm. Already in 1987, Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin described an efficient *parallel* algorithm for computing edit distance, which uses a similar divide-and-conquer strategy as Choudhury and Ramachandran's EditBoundary.

repeatedly sweeping through the entire memoization array to determine the recursive subproblems, their algorithm intuitively consists of two divide-and-conquer phases: a *forward* phase to compute optimal edit distances, followed by a *backward* phase to calculate the actual edit sequence. To simplify presentation, I'll assume that both strings have the same length ($m = n$) and that length is a power of 2; neither of these assumptions is significant.

First I'll describe the forward algorithm ***EDITBOUNDARY***, which is itself an efficient replacement for the usual dynamic programming algorithm. A ***block*** is a square subarray of the memoization array of the form $Edit[i..i+w, j..j+w]$, described by the index i of the top row, the index j of the left column, and the integer w (which is always a power of 2).

The input to EDITBOUNDARY consists of a triple (i, j, w) that describes an *active* block, along with two arrays $T[0..w]$ and $L[0..w]$ that contain the values in the top row (i) and leftmost column (j) of the active block. The output of EDITBOUNDARY consists of two arrays $B[0..w]$ and $R[0..w]$ that contain the values of the bottom row ($i+w$) and rightmost column ($j+w$) of the active block. The algorithm has two cases:

- If $r = 0$, we compute the only unknown output value $Edit[i, j]$ in $O(1)$ time using the edit distance recurrence.
- Otherwise, we split the active block into *quadrants* with half as many rows and columns. We recursively process the upper left quadrant, then the upper-right quadrant, then the lower left quadrant, and finally the lower right quadrant, using the outputs from earlier quadrants as inputs to later quadrants.

Our top-level function call is EDITBOUNDARY$(0, 0, n, T[0..n], L[0..n])$, where $T[i] = L[i] = i$ for every index i, as required by the base cases of the edit-distance recurrence.

```
EDITBOUNDARY(i, j, w, T[0..w], L[0..w]):
  if w = 1
      B[0] ← L[1];  R[0] ← T[1]
      compute B[1] using the edit distance recurrence
      R[1] ← B[1]
      return R[0..1], B[0..1]
  else
      T_11 ← T[0..w/2];  T_12 ← T[w/2..w]
      L_11 ← L[0..w/2];  L_21 ← L[w/2..w]
      T_21, L_12 ← EDITBOUNDARY(i,       j,       w/2, T_11, L_11)
      T_22, R_12 ← EDITBOUNDARY(i,       j+w/2,   w/2, T_12, L_12)
      B_21, L_22 ← EDITBOUNDARY(i+w/2,   j,       w/2, T_21, L_21)
      B_22, R_22 ← EDITBOUNDARY(i+w/2,   j+w/2,   w/2, T_22, L_22)
      return B_21 · B_22, R_12 · R_22    ⟨⟨Concatenation⟩⟩
```

The running time of this algorithm obeys the recurrence $T(n) = 4T(n/2) + O(1)$, and its space usage obeys the recurrence $S(n) = O(n) + S(n/2)$. We conclude that the algorithm runs in $\boldsymbol{O(n^2)}$ ***time*** and uses $\boldsymbol{O(n)}$ ***space***.

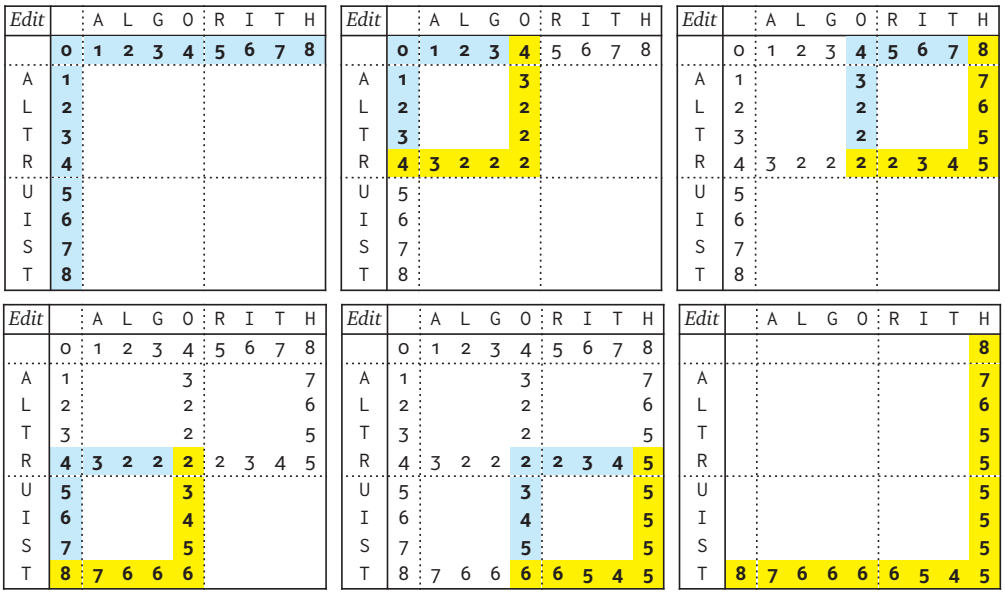


Figure D.2. Chowdhury and Ramachandran's edit distance algorithm: input, four recursive calls, and output.

Choudhury and Ramachandran's more complex algorithm ***EditSequence*** actually computes an optimal path through the memoization array. This algorithm takes two additional input parameters: the indices (ti, tj) of a cell on the right or bottom boundary of the active block. The algorithm returns the indices (si, sj) of a cell on the left or top boundary of the active block, along with an optimal path through the block from (si, sj) to (ti, tj). To compute the optimal edit sequence for two strings, we call EditSequence$(0, 0, n, T[0..n], L[0..n], n, n)$, where again $T[i] = L[i] = i$ for every index i.

Each recursive call in EditSequence updates the indices ti and tj and returns a subpath Z_{ab}; the final path is the concatenation of these subpaths. The key observation is that a recursive calls to EditSequence is non-trivial if and only if the optimal edit path intersects the corresponding quadrant. Thus, the running time obeys the recurrence $T(n) = O(n^2) + 3T(n/2)$, and the space usage still satisfies the recurrence $S(n) = O(n) + S(n/2)$. We conclude that Chowdhury and Ramachandran's algorithm computes the optimal edit sequence in $\boldsymbol{O(n^2)}$ ***time*** and uses $\boldsymbol{O(n)}$ ***space***.

More careful analysis implies that EditSequence makes at most $2(n/2^r)$ recursive calls on blocks of size 2^r, namely, one call for each block that the optimal edit path intersects. Each call to EditSequence invokes EditBoundary on only three of the four quadrants of its active block. It follows that the running time of EditSequence is at most three times the running time of EditBoundary.

$$\sum_{r=0}^{\lg n}\left(2\frac{n}{2^r}\cdot\frac{3}{4}2^{2r}\right) = 3n\sum_{r=0}^{\lg n}2^{r-1} < 3n^2$$

```
EDITSEQUENCE(i, j, w, T[0..w], L[0..w], ti, tj):
  if w = 1
    ⟨⟨Base case: Brute force⟩⟩
    compute operation Z and indices si, sj using the edit distance recurrence
    return Z, si, sj
  else if (ti ≤ i or ti ≥ i + w or tj ≤ j or tj ≥ j + w or (ti < i + w and tj < j + w))
    ⟨⟨Trivial case: target indices are not on the outer block boundary⟩⟩
    return ε, ti, tj
  else
    ⟨⟨Forward phase: Set up inputs for recursive subproblems⟩⟩
    T11 ← T[0..w/2]; T12 ← T[w/2..w];
    L11 ← L[0..w/2]; L21 ← L[w/2..w]
    T21, L12 ← EDITBOUNDARY(i,       j,       w/2, T11, L11)
    T22, R12 ← EDITBOUNDARY(i,       j + w/2, w/2, T12, L12)
    B21, L22 ← EDITBOUNDARY(i + w/2, j,       w/2, T21, L21)
    ⟨⟨Backward phase: Recursively backtrack along the optimal edit path.⟩⟩
    ⟨⟨At most three of these recursive calls actually do anything.⟩⟩
    ⟨⟨Each nontrivial recursive call updates ti and tj.⟩⟩
    Z22, ti, tj ← EDITSEQUENCE(i + w/2, j + w/2, w/2, T22, L22, ti, tj)
    Z21, ti, tj ← EDITSEQUENCE(i + w/2, j,       w/2, T21, L21, ti, tj)
    Z12, ti, tj ← EDITSEQUENCE(i,       j + w/2, w/2, T12, L12, ti, tj)
    Z11, ti, tj ← EDITSEQUENCE(i,       j,       w/2, T11, L11, ti, tj)
    return Z11 · Z12 · Z21 · Z22, ti, tj
```

Figure D.3. Chowdhury and Ramachandran's algorithm for recursively computing an optimal edit sequence.

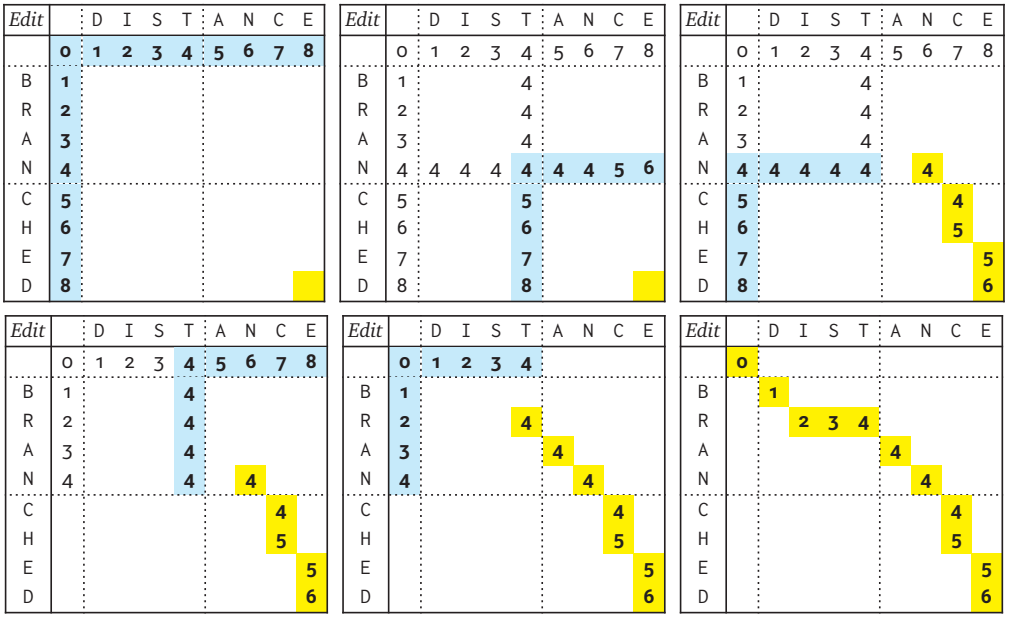


Figure D.4. Chowdhury and Ramachandran's algorithm in action: Input, EDITBOUNDARY calls in three quadrants, recursive calls in all four quadrants, and output. The bottom left recursive call is trivial.

D.2 Saving Time: Four Russians

This is still very rough. Figures would probably help.

★★★

The idea of breaking the memoization table and its iterative evaluation into blocks was already proposed in a brief note published in 1970 by Vladimir Arlazarov, Yefim Dinitz, Alexander Kronrod, and Igor Faradžev. These four Russians made two important observations. First, the entire input (upper left boundary) and output (lower left boundary) values of any sufficiently small block can be encoded to fit into a single machine word. Second, the number of distinct blocks with width w is only(!) exponential in w. Thus, we can precompute and store *all* blocks of sufficiently small width w in a lookup table in $O(n)$ time, and then evaluate blocks of width w in the actual memoization array in $O(1)$ time using that lookup table. This speedup is now commonly known as the "Four Russians" technique.

The Four Russians technique was first applied to computing the edit distance between two strings by William Masek and Mike Paterson in 1980. To make their algorithm concrete, and to simplify the presentation and analysis, I'll make two explicit assumptions:

- Each machine word (that is, each integer or pointer variable) holds $\Theta(\log n)$ bits.[2]
- The input strings are drawn from an alphabet Σ of *constant* size.[3]

Masek and Paterson observed that any two adjacent entries in the memoization table differ by at most 1. For example, deleting $B[j]$ from the optimal edit motif for strings $A[1..i]$ and $B[1..j]$ either changes a replacement into a deletion, or removes an insertion. In both cases, we conclude that $Edit[i, j-1] \le Edit[i, j] + 1$. On the other hand, the inequality $Edit[i, j] \le Edit[i, j-1] + 1$ follows directly from the recurrence.

It follows that the $w \times w$ block $Edit[i..i+w, j..j+w]$ is completely determined by the following data:

- the top-left edit distance $Edit[i, j]$,
- two substrings $\alpha = A[i+1..i+w]$ and $\beta = B[j+1..j+w]$,
- a *left offset* array $\delta L[1..w]$, where $\delta L[k] = Edit[i+k, j] - Edit[i+k-1, j] \in \{-1, 0, +1\}$, and
- a *top offset* array $\delta T[1..w]$, where $\delta T[\ell] = Edit[i, j+\ell] - Edit[i, j+\ell-1] \in \{-1, 0, +1\}$.

Even without knowing $Edit[i, j]$, we can compute *offset vectors for* the bottom row and right column of the block in $O(w^2)$ time using the standard dynamic programming

[2]This is actually a standard technical assumption in most algorithm analysis. In particular, this assumption implies that the entries in the memoization table each fit in a single machine word, and that each of the array lookups, comparisons, and arithmetic operations necessary to fill the table require only constant time.

[3]This restriction can be removed using additional preprocessing, but the resulting algorithm is slower by a factor of $O(\log^2 \log n)$.

algorithm for edit distance (with different base cases). If we are then given the top left distance $Edit[i, j]$, we can compute the bottom-right distance $Edit[i + w, j + w]$ in $O(w)$ time by adding all the left and bottom (or top and right) offsets.

There are trivially at most $|\Sigma|^w$ distinct strings α, at most $|\Sigma|^w$ distinct strings β, at most 3^w distinct offset vectors δL, and at most 3^w distinct offset vectors δT. Thus, in $O((3|\Sigma|)^{2w} w^2)$ time, we can preprocess *all possible* blocks. If we set $w = \frac{1}{2}\log_b n$, where $b = (3|\Sigma|)^2$, the entire preprocessing takes $\boldsymbol{O(\sqrt{n}\log^2 n)}$ ***time***—less time than reading the input strings!

Moreover, for each pair of strings and pair of input offset vectors, the resulting pair of output offset vectors, as well as the sum of the left and bottom offsets, can be encoded in $(2 \lg 3) \cdot w + O(\log w) = O(\log n)$ bits, and therefore in a constant number of machine words. We can store this data for *all possible* blocks in a lookup table of size $O((3|\Sigma|)^{2w}) = O(\sqrt{n})$. Accessing the data for any particular block, given its determining substrings and offset vectors, takes only constant time.

Now partition the $n \times n$ memoization table into n^2/w^2 blocks, each of size $w \times w$. After computing the top and left offset vectors for the entire table in $O(n)$ time, we can compute the final edit distance $Edit[n, n]$ in $O(n^2/w^2)$ time, by performing one table lookup per block in row-major order. The overall algorithm runs in $\boldsymbol{O(n^2/\log^2 n)}$ ***time***; the time bound is dominated by two nested loops of table lookups.

Finally, it is possible to combine Masek and Paterson's "Four Russian" optimization with Chowdhury and Ramachandran's divide-and-conquer algorithm to compute the optimal edit sequence between two strings of length n in $O(n^2/\log^2 n)$ time and $O(n/\log n)$ space (not including the output sequence itself).

D.3 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.

Fix the two input strings $A[1..n]$ and $B[1..m]$. For any indices i and j, let $LCS(i, j)$ denote the length of the longest common subsequence of the prefixes $A[1..i]$ and $B[1..j]$.

This function can be defined recursively as follows:

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

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

LCS	«	A	L	G	O	R	I	T	H	M	S	»
«	**0**	0	0	0	0	0	0	0	0	0	0	0
A	0	**1**	1	1	1	1	1	1	1	1	1	1
L	0	1	**2**	2	2	2	2	2	2	2	2	2
T	0	1	2	2	2	2	2	**3**	3	3	3	3
R	0	1	2	2	2	**3**	3	3	3	3	3	3
U	0	1	2	2	3	3	3	3	3	3	3	3
I	0	1	2	2	3	3	**4**	4	4	4	4	4
S	0	1	2	2	3	3	4	4	4	4	**5**	5
T	0	1	2	2	3	3	4	**5**	5	5	5	5
I	0	1	2	2	3	3	**4**	5	5	5	5	5
C	0	1	2	2	3	3	4	5	5	5	5	5
»	0	1	2	2	3	3	4	5	5	5	5	**6**

Figure D.5. The *LCS* memoization table for the strings ALGORITHMS and ALTRUISTIC; the brackets « and » are sentinel characters. Match points are highlighted.

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 can reconstruct the entire table using the following recurrence:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ \max\left\{LCS(i',j') \;\middle|\; A[i'] = B[j'] \text{ and } i' < i \text{ and } j' < j\right\} + 1 & \text{if } A[i] = B[j] \\ \max\left\{LCS(i',j') \;\middle|\; A[i'] = B[j'] \text{ and } i' \le i \text{ and } j' \le j\right\} & \text{otherwise} \end{cases}$$

(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 (brackets « and » in the example above). These sentinels ensure 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 in $O(m \log m + n \log n + K)$ time. Sort the characters in each input string, remembering the original index of each character, and then essentially merge the two sorted arrays, as shown in Figure D.6.

To efficiently evaluate our modified recurrence, we once again turn to dynamic programming. We consider the match points in lexicographic order—the order they

```
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
```

Figure D.6. Computing matches between characters in two strings.

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
```

Figure D.7. Computing the longest common subsequence in $O(m \log m + n \log n + K^2)$ time.

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

additional work, the running time can be further improved to $O(m \log m + n \log n + K \log K)$.

D.4 Saving Time: Monotonicity

Recall the optimal binary search tree problem from the previous lecture. Given an array $f[1..n]$ of access frequencies for n items, we want 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.

Once again, this algorithm can be improved by exploiting structure in the memoization table. In this case, however, the relevant structure isn't in the table of *costs*, but rather in an auxiliary 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:

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

In other words, every row and column in the array $OptRoot[1..n, 1..n]$ is sorted.

Cost	3	1	4	1	5	9	2	6	5
3	3	5	13	15	26	47	53	67	82
1	0	1	6	8	19	37	43	57	72
4		0	4	6	16	34	39	53	68
1			0	1	7	22	26	40	55
5				0	5	19	23	37	52
9					0	9	13	27	40
2						0	2	10	20
6							0	6	16
5								0	5

Root	3	1	4	1	5	9	2	6	5
3	1	1	3	3	3	5	5	6	6
1		2	3	3	3	5	5	6	6
4			3	3	5	5	6	6	6
1				4	5	6	6	6	6
5					5	6	6	6	6
9						6	6	6	6
2							7	8	8
6								8	8
5									9

Figure D.8. Optimal costs and optimal roots for the frequency array $[3, 1, 4, 1, 5, 9, 2, 6, 5]$.

This (nontrivial!) observation suggests the following more efficient algorithm. The only differences from our previous diagonal-by-diagonal dynamic programming algorithm are the lines in red. In particular, the main loop in ComputeCostAndRoot only runs from $OptRoot[i, j-1]$ to $OptRoot[i+1, j]$, instead of the wider range from i to j.

```
FASTEROPTIMALSEARCHTREE(f[1..n]):
  INITF(f[1..n])
  for i ← 1 to n
      OptCost[i, i−1] ← 0
      OptRoot[i, i−1] ← i
  for d ← 0 to n
      for i ← 1 to n−d
          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[4] computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

D.5 Saving Time: More Monotonicity

Knuth's algorithm can be significantly generalized by considering a more subtle form of monotonicity in the *cost* array. A common (but often implicit) operation in many dynamic programming algorithms is finding the minimum element in every row of a two-dimensional array. For example, consider a single iteration of the outer loop in FASTEROPTIMALSEARCHTREE, for some fixed value of d. Define an array M by setting

$$M[i,r] = \begin{cases} OptCost[i, r-1] + OptCost[r+1, i+d] & \text{if } i \le r \le i+d \\ \infty & \text{otherwise} \end{cases}$$

[4]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.

Each call to ComputeCostAndRoot$(i, i+d)$ computes the smallest element in the ith row of this array M.

Let $M[1..m, 1..n]$ be an arbitrary two-dimensional array. We say that M is ***monotone*** if the leftmost smallest element in any row is either directly above or to the left of the leftmost smallest element in any later row. To state this condition more formally, let $LM(i)$ denote the index of the smallest item in the ith row $M[i, \cdot]$; if there is more than one smallest item, choose the one furthest to the left. Then M is monotone if and only if $LM(i) \le LM(i+1)$ for every index i. For example, the following 5×5 array is monotone (as shown by the highlighted row minima).

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 27 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 25 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

Given a monotone $m \times n$ array M, we can compute the array $LM[i]$ containing the index of the leftmost minimum element using the following algorithm, which I'll call Filter. We begin by recursively computing the leftmost minimum elements in all odd-indexed rows of M. Then for each even index $2i$, the monotonicity of M implies the bounds

$$LM[2i-1] \le LM[2i] \le LM[2i+1],$$

so we can compute $LM[2i]$ by brute force by searching only within that range of indices. This search requires exactly $LM[2i+1] - LM[2i-1]$ comparisons, because finding the minimum of k numbers requires $k-1$ comparisons. In particular, if $LM[2i-1] = LM[2i+1]$, then we don't need to perform any comparisons on row $2i$. Summing over all even indices, we find that the total number of comparisons is

$$\sum_{i=1}^{m/2} LM[2i+1] - LM[2i-1] \;=\; LM[m+1] - LM[1] \;\le\; n.$$

We also need to spend constant time on each row, as overhead for the main loop, so the total running time of Filter is $O(n+m)$ plus the time for the recursive call. Because the number of rows is halved in the recursive call, the running time satisfies the recurrence

$$T(m,n) = T(m/2, n) + O(n+m),$$

which implies that Filter runs in $\boldsymbol{O(m + n \log m)}$ ***time***.

Alternatively, we could use the following divide-and-conquer procedure, similar in spirit to Hirschberg's divide-and-conquer algorithm. Compute the middle leftmost-minimum index $h = LM[m/2]$ by brute force in $O(n)$ time, and then recursively find the leftmost minimum entries in the following submatrices:

$$M\,[1..m/2-1,\ 1..h] \qquad M\,[m/2+1..m,\ h..n]$$

The worst-case running time $T(m,n)$ for this algorithm obeys the following recurrence (after removing some irrelevant ± 1s from the recursive arguments, as usual):

$$T(m,n) = \begin{cases} 0 & \text{if } m < 1 \\ O(n) + \max_k \big(T(m/2,k) + T(m/2,n-k)\big) & \text{otherwise} \end{cases}$$

The recursion tree for this recurrence is a balanced binary tree of depth $\log_2 m$, and therefore with $O(m)$ nodes. The total number of *comparisons* performed at each level of the recursion tree is $O(n)$, but we also need to spend at least constant time at each node in the recursion tree. Thus, this divide-and-conquer formulation also runs in $\boldsymbol{O(m + n \log m)}$ ***time***.

In fact, these two algorithms are morally identical. Both algorithms examine the same subset of array entries and perform the same pairwise comparisons, although in different orders. Specifically, the divide-and-conquer algorithm performs the usual depth-first traversal of its recursion tree. The even-odd algorithm actually performs a *breadth*-first traversal of the exactly the same recursion tree, handling each level of the recursion tree in a single for-loop. The breadth-first formulation of the algorithm will prove more useful in the long run.

D.6 Total Monotonicity

A more general technique for exploiting structure in dynamic programming arrays was discovered by Alok Aggarwal, Maria Klawe, Shlomo Moran, Peter Shor, and Robert Wilber in 1987; their algorithm is now universally known as ***SMAWK*** (pronounced "smoke") after their suitably-permuted initials.

SMAWK requires a stricter form of monotonicity in the input array. We say that M is ***totally monotone*** if the subarray defined by any subset of (not necessarily consecutive) rows and columns is monotone. For example, the following 5×5 array is monotone (as shown by the highlighted row minima), but not totally monotone (as shown by the gray 2×2 subarray).

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 27 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 25 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

On the other hand, the following array is totally monotone:

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 89 \\ 47 & 14 & 14 & 29 & 60 \\ 21 & 8 & 20 & 10 & 71 \\ 68 & 16 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

Given a totally monotone $n \times n$ array as input, SMAWK finds the leftmost smallest element of every row in only $\boldsymbol{O(n)}$ ***time***.

The Monge property

Before we go into the details of the SMAWK algorithm, it's useful to consider the most common special case of totally monotone matrices. A ***Monge array***[5] is a two-dimensional array M where

$$M[i,j] + M[i',j'] \le M[i,j'] + M[i',j]$$

for all row indices $i < i'$ and all column indices $j < j'$. This inequality (sometimes with additional retrictions on the indices) is sometimes called the *Monge property* or *concavity* or the *quadrangle inequality* or *submodularity*.

Lemma D.1. *Every Monge array is totally monotone.*

Proof: Let M be a two-dimensional array that is *not* totally monotone. Then there must be row indices $i < i'$ and column indices $j < j'$ such that $M[i,j] > M[i,j']$ and $M[i',j] \le M[i',j']$. These two inequalities imply that $M[i,j] + M[i',j'] > M[i,j'] + M[i',j]$, from which it follows immediately that M is *not* Monge. □

The converse of this lemma is false; for example, the totally monotone array on the bottom of the previous page is not Monge.

Monge arrays have several useful properties that make identifying them easier. For example, an easy inductive argument implies that we do not need to check every quadruple of indices; in fact, we can determine whether an $m \times n$ array is Monge in just $O(mn)$ time.

Lemma D.2. *An array M is Monge if and only if $M[i,j] + M[i+1,j+1] \le M[i,j+1] + M[i+1,j]$ for all indices i and j.*

Monge arrays arise naturally in geometric settings; the following canonical example of a Monge array was suggested by Monge himself in 1781. Fix two parallel lines ℓ and ℓ' in the plane. Let $p_1, p_2, \ldots, p_m$ be points on ℓ, and let $q_1, q_2, \ldots, q_n$ be points on ℓ', with each set indexed in order along their respective lines. Let $M[1..m, 1..n]$ be the array of Euclidean distances between these points:

$$M[i,j] := |p_i q_j| = \sqrt{(p_i.x - q_i.x)^2 + (p_i.y - q_i.y)^2}$$

[5] Monge arrays are named after Gaspard Monge, a French geometer who was one of the first to consider problems related to flows and cuts, in his 1781 *Mémoire sur la Théorie des Déblais et des Remblais*, which translates loosely as "Treatise on the Theory of Holes and Dirt". (Civil engineers are perhaps more familiar with the terminology "cut and fill", but let's not overload the word "cut".)

An easy argument with the triangle inequality implies that this array is Monge. Fix arbitrary indices $i < i'$ and $j < j'$, and let x denote the intersection point of segments $p_i q_{j'}$ and $p_{i'} q_j$.

$$\begin{aligned} M[i,j] + M[i',j'] &= |p_i q_j| + |p_{i'} q_{j'}| \\ &\le |p_i x| + |x q_j| + |p_{i'} x| + |x q_{j'}| \qquad \text{[triangle inequality]} \\ &= \big(|p_i x| + |x q_{j'}|\big) + \big(|p_{i'} x| + |x q_j|\big) \\ &= |p_i q_{j'}| + |p_{i'} q_j| \\ &= M[i,j'] + M[i',j] \end{aligned}$$

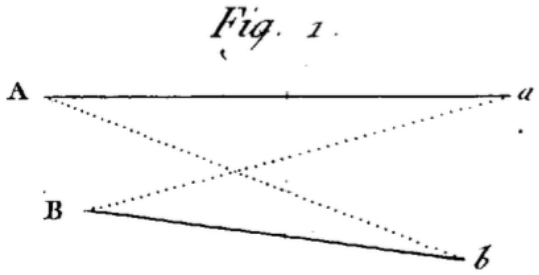


Figure D.9. The cheapest way to move two specks of dirt (on the left) into two tiny holes (on the right). From Monge's 1781 treatise on the theory of holes and dirt.

There are also several easy ways to construct and combine Monge arrays, either from scratch or by manipulating other Monge arrays.

Lemma D.3. *The following arrays are Monge:*

(a) Any array with constant rows.
(b) Any array with constant columns.
(c) Any array that is all 0*s except for an upper-right rectangular block of* 1*s.*
(d) Any array that is all 0*s except for an lower-left rectangular block of* 1*s.*
(e) Any positive multiple of any Monge array.
(f) The sum of any two Monge arrays.
(g) The transpose of any Monge array.

Each of these properties follows from straightforward definition chasing. For example, to prove part (f), let X and Y be arbitrary Monge arrays, and let $Z = X + Y$. For all row indices $i < i'$ and column indices $j < j'$, we immediately have

$$\begin{aligned} Z[i,j] + Z[i',j'] &= X[i,j] + X[i',j'] + Y[i,j] + Y[i',j'] \\ &\le X[i',j] + X[i,j'] + Y[i',j] + Y[i,j'] \\ &= Z[i',j] + Z[i,j']. \end{aligned}$$

The other properties have similarly elementary proofs.

In fact, the properties listed in the previous lemma precisely characterize Monge arrays: ***Every*** Monge array (including the geometric example above) is a positive linear combination of row-constant, column-constant, and upper-right block arrays. (This characterization was proved independently by Rudolf and Woeginger in 1995, Bein and Pathak in 1990, Burdyok and Trofimov in 1976, and possibly others.)

D.7 The SMAWK algorithm

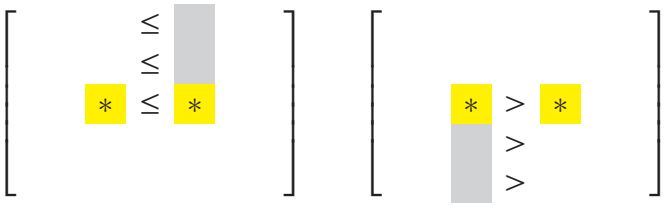
The SMAWK algorithm alternates between two different subroutines, one for "tall" arrays with more rows than columns, the other for "wide" arrays with more columns than rows. The "tall" subroutine is just the divide-and-conquer algorithm FILTER described in the previous section—recursively compute the leftmost row minima in every other row, and then fill in the remaining row minima in $O(n+m)$ time. The secret to SMAWK's success is its handling of "wide" arrays.

REDUCE-ing wide arrays

For any row index i and any two column indices $p < q$, the definition of total monotonizity implies the following observations:

- If $M[i,p] \le M[i,q]$, then for all $h \le i$, we have $M[h,p] \le M[h,q]$ and thus $LM[h] \neq q$.
- If $M[i,p] > M[i,q]$, then for all $j \ge i$, we have $M[j,p] > M[j,q]$ and thus $LM[j] \neq p$.

Call an array entry $M[i,j]$ ***dead*** if we have enough information to conclude that $LM[i] \neq j$. Then after we compare any two entries in the same row, either the left entry and everything below it is dead, or the right entry and everything above it is dead.



The REDUCE algorithm shown in Figure D.10 maintains a stack of column indices in an array $S[1..m]$ (yes, really m, the number of *rows*) and an index t indicating the number of indices on this stack.

```
REDUCE(M[1..m, 1..n]):
  t ← 1
  S[t] ← 1
  for k ← 1 to n
      while t > 0 and M[t, S[t]] ≥ M[t, k]
          t ← t − 1        ⟨⟨pop⟩⟩
      if t < m
          t ← t + 1
          S[t] ← k         ⟨⟨push k⟩⟩
  return S[1..t]
```

Figure D.10. The SMAWK algorithm to REDUCE wide arrays

The REDUCE algorithm maintains three important invariants:

- $S[1..t]$ is sorted in increasing order.
- For all $1 \le j \le t$, the top $j-1$ entires of column $S[j]$ are dead.
- If $j < k$ and j is not on the stack, then *every* entry in column j is dead.

The first invariant follows immediately from the fact that indices are pushed onto the stack in increasing order. The second and third are more subtle, but both follow inductively from the total monotonicity of the array. Figure D.11 shows a typical state of the algorithm when it is about to compare $M[t, S[t]]$ and $M[t, k]$.

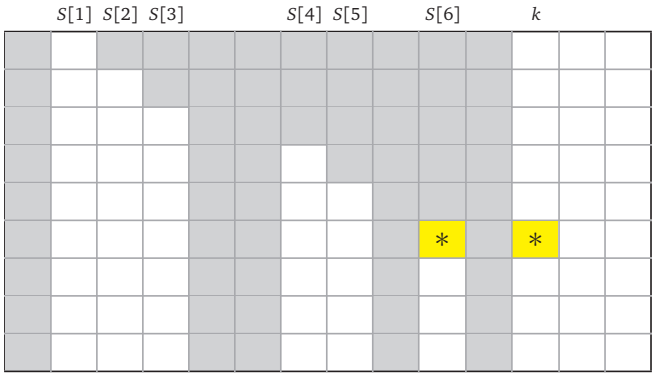


Figure D.11. A typical state of REDUCE, just before comparing the starred entries. Dead entries are gray.

There are two cases to consider:

- If $M[t, S[t]] < M[t, k]$, then $M[t, k]$ and every cell above it are dead, so we can safely push column k onto the stack (unless we already have $t = m$, in which case every entry in column k is dead) and then increment k. See Figure D.12.
- On the other hand, if $M[t, S[t]] > M[t, k]$, then we know that $M[t, S[t]]$ and everything below it is dead. But by the inductive hypothesis, every entry above $M[t, S[t]]$ is already dead. Thus, the entire column $S[t]$ is dead, so we can safely pop it off the stack. See Figure D.13.

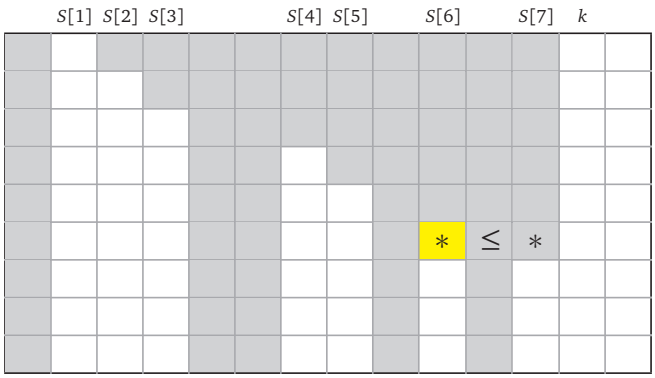


Figure D.12. Case 1: Push column k and increment k.

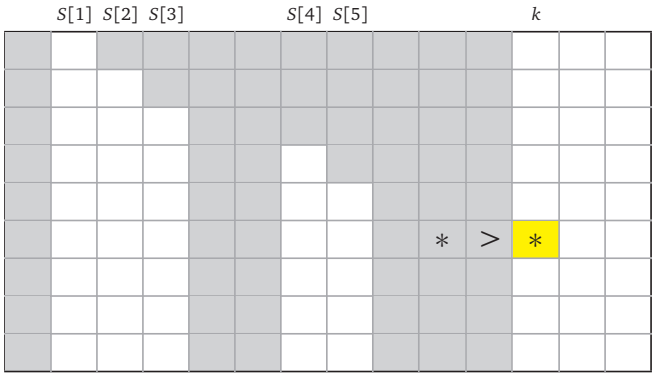


Figure D.13. Case 2: Kill column $S[t]$ and decrement t.

In both cases, all three invariants are maintained.

Immediately after every comparison $M[t, S[t]] \geq M[t, k]$? in the REDUCE algorithm, we either increment the column index k or declare column $S[t]$ to be dead; each of these events can happen at most once per column. It follows that REDUCE performs at most $2n$ comparisons and thus runs in ***O(n) time*** overall.

Moreover, when REDUCE ends, every column whose index is not on the stack is completely dead. Thus, to compute the leftmost minimum element in every row of M, it suffices to examine only the $t < m$ columns with indices in the output array $S[1..t]$.

Combining the Algorithms

Finally, given any totally monotone $m \times n$ array, the main SMAWK algorithm proceeds as follows:

- If $m = 1$, find the leftmost minimum element in the only row by brute force in $O(n)$ time.

- Otherwise, if the array is wide ($m < n$), REDUCE the input array in $O(m+n) = O(n)$ time, and then recursively compute row minima in the resulting array, *which is always tall*.
- Finally, if the array is tall ($m \geq n$), FILTER out the odd-indexed rows, recursively compute the row minima in the resulting smaller array, and then compute the remaining row minima in $O(m+n) = O(m)$ time.

The running time of the combined algorithm obeys the following recurrence:

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

If the input array is tall, we may need to call several levels of FILTER recursion to reduce to a wide array, but since the running time of each level is dominated by $O(m)$ and each level of recursion halves m, the total time for this reduction is $O(m)$. It follows that we can modify our recurrence to

$$T(m,n) \leq \begin{cases} O(n) & \text{if } m = 1 \\ O(n) + T(m,m) & \text{if } 1 < m < n \\ O(m) + T(\mathbf{n}/2, n) & \text{if } n \leq m \end{cases}$$

Conversely, if the input array is wide, a single call to REDUCE extracts a square subarray in $O(n)$ time. Once the array becomes square, the algorithm alternates between wide arrays and tall arrays, halving both the height and width of the array every two levels of recursion. So again we can modify the recurrence as follows:

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

We conclude that the overall SMAWK algorithm runs in $\boldsymbol{O(m+n)}$ ***time***.

Later variants of SMAWK can compute row minima in totally monotone arrays in $O(m+n)$ time even when the rows are revealed one by one, and we require the smallest element in each row before we are given the next. These later variants can be used to speed up many dynamic programming algorithms by a factor of n when the final memoization table is totally monotone.

D.8 Using SMAWK

Finally, let's consider some examples of dynamic programming algorithms that can be improved by SMAWK.

Minimum Weight Subsequence

Suppose we are given a sorted array $S[1..n]$ of values, which we'd like to partition into exactly K intervals of roughly equal *span*, where the span of the interval $S[i..j]$ is defined as $span(i,j) = S[j] - S[i]$. Specifically, we want to choose a sorted array $B[0..K]$ of K *breakpoint* indices, where $B[0] = 1$ and $B[k] = n$, such that the sum of the squared spans

$$\sum_{i=1}^{K} \big(span(B[i], B[i+1])\big)^2 \;=\; \sum_{i=1}^{K} \big(S[B[i]] - S[B[i+1]]\big)^2$$

is as small as possible.

The standard dynamic programming algorithm for this problem runs in $O(n^2K)$ time. For any indices j and k, let $OptCost(j,k)$ denote the optimal cost of a partition of $A[1..j]$ into k intervals. This function satisfies the following recurrence:

$$OptCost(j,k) = \begin{cases} 0 & \text{if } k = 0 \text{ and } j = 0 \\ \infty & \text{if } k < j \\ \min\left\{ (span(i,j))^2 + OptCost(i-1,k) \;\middle|\; 1 \le i < j \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into an $n \times K$ array, which we can fill by increasing k in the outer loop and considering i in any order in the inner loop. Finally, we need $O(n)$ time to fill each entry, so the overall running time is $O(n^2K)$, as claimed.

We can improve the running time by a factor of n by applying SMAWK, as follows. For any fixed k, define an $n \times n$ upper-triangular array $Cost_k[1..n, 1..n]$ by setting

$$Cost_k[i,j] := \begin{cases} (span(i,j))^2 + OptCost(i-1,k) & \text{if } i < j \\ \infty & \text{if } i = j \end{cases}$$

and leaving $Cost_k[i,j]$ undefined when $i > j$. In the kth iteration of the outer loop of our dynamic programming algorithm, we are setting $OptCost[j,k]$ to the smallest element in the jth *column* of $Cost_k$, for every index j. This is *almost* the problem that SMAWK is designed to solve! As we will see shortly, every 2×2 subarray of $Cost_k$ *whose elements are defined* satisfies the Monge inequality. And we're looking for column minima instead of row minima, but that transposition is straightforward.

Unfortunately, SMAWK requires a full rectangular array, so we need to fill in the missing values. Naively replacing each undefined value with ∞ doesn't work; the resulting rectangular array is *not* totally monotone. Instead, we consider the behavior of the following modified array in the limit, as the variable X grows to infinity:

$$Cost_k[i,j] := \begin{cases} (span(i,j))^2 + OptCost(i-1,k) & \text{if } i < j \\ (2i - 2j + 1)X & \text{otherwise} \end{cases}$$

This array has value X in every cell of the main diagonal, $3X$ along the next lower diagonal, $5X$ along the next lower diagonal, and so on.

Lemma D.4. *$Cost_k$ is Monge, for all sufficiently large X.*

Proof: We can write $Cost_k = A + C_k$, where

$$A[i,j] = \begin{cases} (span(i,j))^2 & \text{if } i < j \\ (2i-2j+1)X & \text{otherwise,} \end{cases} \qquad \text{and} \qquad C_k[i,j] = OptCost(i-1,k).$$

Array C_k has constant rows and is therefore Monge by Lemma D.3(b). Lemma D.3(f) implies that to complete the proof, it suffices to show that the array A is also Monge.

X	9	16	64	81	196	529
$3X$	X	1	25	36	121	400
$5X$	$3X$	X	16	25	100	361
$7X$	$5X$	$3X$	X	1	36	225
$9X$	$7X$	$5X$	$3X$	X	25	196
$11X$	$9X$	$7X$	$5X$	$3X$	X	81
$13X$	$11X$	$9X$	$7X$	$5x$	$3X$	X

Figure D.14. The squared-span array A for the input $S = [0,3,4,8,9,14,23]$; for all sufficiently large X, this array is Monge.

Fix arbitrary indices i and j. To simplify calculation, let us write $w = S[i]$, $x = S[i+1]$, $y = S[j]$, and $z = S[j+1]$. There are four cases to consider.

- Suppose $i+1 < j$; this is the most interesting case. Because S is sorted, we know that $w < x < y < z$. High-school algebra implies that

$$\begin{aligned} A[i,j+1] + A[i+1,j] - A[i,j] - A[i+1,j+1] \\ &= (z-w)^2 + (y-x)^2 - (y-w)^2 - (z-x)^2 \\ &= -2zw - 2xy + 2wy + 2xz \\ &= 2(x-w)(z-y) > 0. \end{aligned}$$

- If $i+1 = j$, The Monge inequality becomes $A[i,j] + A[i+1,j+1] \le A[i,j+1] + X$, which holds for all sufficiently large X.
- If $i = j$, the Monge inequality becomes $X + X \le A[i,j+1] + 3X$, which holds for all $X \ge 0$ (because $A[i,j+1] > 0$).
- Finally, if $i < j$, the Monge inequality becomes

$$(2i-2j+1)X + (2i-2j+1)X \le (2i-2j+3)X + (2i-2j-1)X$$

  the two sides of this inequality are actually *equal* for *all* X.

In all four cases, the Monge inequality $A[i,j] + A[i+1,j+1] leA[i,j+1] + A[i+1,j]$ holds for all sufficiently large X. In particular, it suffices to set $X = 2 \cdot (span(1,n))^2$. □

This lemma implies that for any sufficiently large X, we can compute the minimum element of every column of $Cost_k$ in $O(n)$ time using SMAWK. (And for sufficiently large X, these column minima are always above the main diagonal!) With this optimization, our dynamic programming algorithm runs in only ***O(nK) time***!

An important feature of this algorithm is that ***it never explicitly constructs the n × n array Cost$_k$***; constructing just one $n \times n$ array would take longer than our entire algorithm! Instead, whenever SMAWK needs a particular entry $Cost_k[i, j]$, we compute it on the fly in $O(1)$ time.

Moreover, we don't need to compute an actual *value* for X; we can treat X purely symbolically, as follows. Every decision in the (transposed) SMAWK algorithm is based on either a comparison between indices, or a comparison between two entries *in the same column* of A. For all sufficiently large X, the latter comparisons are consistent with the following definition:

$$A[i,j] \preceq A[i',j] \iff \begin{cases} A[i,j] \le A[i',j] & \text{if } i < j \text{ and } i' < j \\ i \le i' & \text{otherwise} \end{cases}$$

In short, we don't need to define those lower array entries after all!!

Shortest Paths

As a second (admittedly somewhat artificial) example, recall the classical Bellman-Ford algorithm for computing shortest paths:

```
BellmanFord(V, E, w):
  dist(s) ← 0
  for every vertex v ≠ s
      dist(v) ← ∞
  repeat V − 1 times
      for every edge u→v
          if dist(v) > dist(u) + w(u→v)
              dist(v) ← dist(u) + w(u→v)
```

We can rewrite this algorithm slightly; instead of relaxing *every* edge, we first identify the *tensest* edge leading into each vertex, and then relax only those edges. (This modification is actually closer to Bellman's original dynamic-programming formulation.)

```
ModBellmanFord(V, E, w):
  dist(s) ← 0
  for every vertex v ≠ s
      dist(v) ← ∞
  repeat V − 1 times
      for every vertex v
          mind(v) ← min_u (dist(u) + w(u→v))
      for every vertex v
          dist(v) ← min{dist(v), mind(v)}
```

The two lines in red can be interpreted as finding the minimum element in every row of a two-dimensional array M indexed by vertices, where

$$M[v,u] := dist(u) + w(u \to v)$$

Now consider the following input graph. Fix n arbitrary points $p_1, p_2, \ldots, p_n$ on some line ℓ and n more arbitrary points $q_1, q_2, \ldots, q_n$ on another line parallel to ℓ, with each set indexed in order along their respective lines. Let G be the complete bipartite graph whose vertices are the points p_i or q_j, whose edges are the segments $p_i q_j$, and where the weight of each edge is its natural Euclidean length. The standard version of Bellman-Ford requires $O(VE) = O(n^3)$ time to compute shortest paths in this graph.

The $2n \times 2n$ array M is not itself Monge, but we can easily decompose it into $n \times n$ Monge arrays as follows. Index the rows and columns of M by the vertices $p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_n$ in that order. Then M decomposes naturally into four $n \times n$ blocks

$$M = \begin{bmatrix} \infty & U \\ V & \infty \end{bmatrix},$$

where every entry in the upper left and lower right blocks is ∞, and the other two blocks satisfy

$$U[i,j] = dist(q_j) + |p_i q_j| \qquad \text{and} \qquad V[i,j] = dist(p_j) + |p_j q_i|$$

for all indices i and j. Let P, Q, and D be the $n \times n$ arrays where

$$P[i,j] = dist(p_j) \qquad Q[i,j] = dist(q_j) \qquad D[i,j] = |p_i q_j|$$

for all indices i and j. Arrays P and Q are Monge, because all entries in the same column are equal, and the matrix D is Monge by the triangle inequality. It follows that the blocks $U = Q + D$ and $V = P + D^T$ are both Monge. Thus, by calling SMAWK twice, once on U and once on V, we can find all the row minima in M in $O(n)$ time. The resulting modification of Bellman-Ford runs in ***O(n²) time***.[6]

An important feature of this algorithm is that ***it never explicitly constructs the array M***. Instead, whenever the SMAWK algorithm needs a particular entry $M[u,v]$, we compute it on the fly in $O(1)$ 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 our earlier FINDMATCHES algorithm as a subroutine.]*

[6] Yes, we could also achieve this running time using Disjktra's algorithm with Fibonacci heaps.

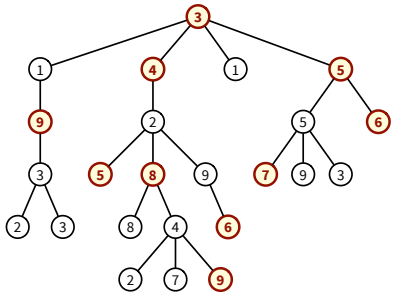
2. (a) Describe an algorithm to compute the *longest increasing subsequence* of a string $X[1..n]$ in $O(n \log n)$ time.

   (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).]*

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.]*

5. Suppose you are given a sorted array $X[1..n]$ of distinct numbers and a positive integer k. A set of k intervals ***covers*** X if every element of X lies inside one of the k intervals. Your aim is to find k intervals $[a_1, z_1], [a_2, z_2], \ldots, [a_k, z_k]$ that cover X where the function $\sum_{i=1}^{k} (z_i - a_i)^2$ is as small as possible. Intuitively, you are trying to cover the points with k intervals whose lengths are as close to equal as possible.

(a) Describe an algorithm that finds k intervals with minimum total squared length that cover X. The running time of your algorithm should be a simple function of n and k.

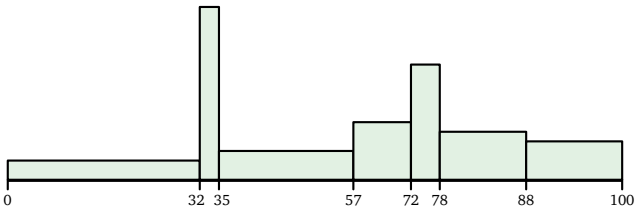
(b) Consider the two-dimensional matrix $M[1..n, 1..n]$ defined as follows:

$$M[i,j] = \begin{cases} (X[j]-X[i])^2 & \text{if } i \le j \\ \infty & \text{otherwise} \end{cases}$$

Prove that M satisfies the ***Monge property***: $M[i,j]+M[i',j'] \le M[i,j']+M[i',j]$ for all indices $i < i'$ and $j < j'$.

(c) Describe an algorithm that finds k intervals with minimum total squared length that cover X ***in $O(nk)$ time***. *[Hint: Solve part (a) first, then use part (b).]*

6. Suppose we want to summarize a large set S of values—for example, grade-point averages for every student who has ever attended UIUC—using a variable-width histogram. To construct a histogram, we choose a sorted sequence of ***breakpoints*** $b_0 < b_1 < \cdots < b_k$, such that every element of S lies between b_0 and b_k. Then for each interval $[b_{i-1}, b_i]$, the histogram includes a rectangle whose height is the number of elements of S that lie inside that interval.



A variable-width histogram with seven bins.

Unlike a standard histogram, which requires the intervals to have equal width, we are free to choose the breakpoints arbitrarily. For statistical purposes, it is useful for the *areas* of the rectangles to be as close to equal as possible. To that end, define the ***cost*** of a histogram to be the sum of the *squares* of the rectangle areas; we seek a histogram with minimum cost.

More formally, suppose we fix a sequence of breakpoints $b_0 < b_1 < \cdots < b_k$. For each index i, let n_i denote the number of input values in the ith interval:

$$n_i := \#\left\{x \in S \;\middle|\; b_{i-1} \le x < b_i\right\}.$$

Then the ***cost*** of the resulting histogram is $\sum_{i=1}^{k} (n_i(b_i - b_{i-1}))^2$. We want to compute a histogram with minimum cost for the given set S, where every breakpoint b_i is equal to some value in S.[7] In particular, b_0 must be the minimum value in S, and b_k must be the maximum value in S.

Describe and analyze an algorithm to compute a variable-width histogram with minimum cost for a given set of data values. Your input is a sorted array $S[1..n]$ of distinct real numbers and an integer k. Your algorithm should return a sorted array $B[0..k]$ of breakpoints that minimizes the cost of the resulting histogram, where every breakpoint $B[i]$ is equal to some input value $S[j]$, and in particular $B[0] = S[1]$ and $B[k] = S[n]$.

[7]Thanks to the non-linear cost function, removing this assumption makes the problem *considerably* more difficult!

