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)

⋆6 Advanced Dynamic Programming

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 $\mathit{Half}(i,j)$ 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.

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 $B[1\,..\,\mathit{Half}(m,n)]$ 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 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:

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

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.

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

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.

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**

The *LCS* memoization table for ALGORITHMS and ALTRUISTIC; the brackets « and » are sentinel characters. Match points are indicated in red.

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 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 $i \gets 1$ to $m$:  $I[i] \gets i$
  for $j \gets 1$ to $n$:  $J[j] \gets j$

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

  $i \gets 1$;  $j \gets 1$
  while $i < m$ and $j < n$
      if $A[i] < B[j]$
          $i \gets i + 1$
      else if $A[i] > B[j]$
          $j \gets j + 1$
      else                    ⟨⟨Found a match!⟩⟩
          $ii \gets i$
          while $A[ii] = A[i]$
              $jj \gets j$
              while $B[jj] = B[j]$
                  report $(I[ii], J[jj])$
                  $jj \gets jj + 1$
              $ii \gets i + 1$
          $i \gets ii$;  $j \gets jj$
```

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($A[1..m], B[1..n]$):
  $Match[1..K] \gets$ FINDMATCHES($A, B$)
  $Match[K+1] \gets (m+1, n+1)$        ⟨⟨Add end sentinel⟩⟩
  Sort $M$ lexicographically
  for $k \gets 1$ to $K$
      $(i, j) \gets Match[k]$
      $LCS[k] \gets 1$                 ⟨⟨From start sentinel⟩⟩
      for $\ell \gets 1$ to $k - 1$
          $(i', j') \gets Match[\ell]$
          if $i' < i$ and $j' < j$
              $LCS[k] \gets \min\{LCS[k], 1 + LCS[\ell]\}$
  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. With some additional work, the running time can be further improved to $O(m \log m + n \log n + K \log K)$.

6.3 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, 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 $\mathit{OptRoot}[i,j]$ 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:

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

In other words, every row and column in the array $\mathit{OptRoot}[1\,..\,n, 1\,..\,n]$ is sorted. This (nontrivial!) observation immediately suggests the following more efficient algorithm:

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

```
COMPUTECOSTANDROOT($i, j$):
  $OptCost[i, j] \leftarrow \infty$

  for $r \leftarrow OptRoot[i, j-1]$ to $OptRoot[i+1, j]$
      $tmp \leftarrow OptCost[i, r-1] + OptCost[r+1, j]$
      if $OptCost[i, j] > tmp$
          $OptCost[i, j] \leftarrow tmp$
          $OptRoot[i, j] \leftarrow r$

  $OptCost[i, j] \leftarrow OptCost[i, j] + F[i, j]$
```

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!

6.4 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

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

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

[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.

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 hilighted 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 in every row as follows. 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 time for our algorithm is $O(n+m)$ plus the time for the recursive call. Thus, the running time satisfies the recurrence

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

which implies that our algorithm runs in ***$O(m + n\log m)$ 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 $h = LM[\frac{m}{2}]$ by brute force in $O(n)$ time, and then recursively find the leftmost minimum entries in the following submatrices:

$$M\left[1 \,..\, \tfrac{m}{2} - 1,\ 1 \,..\, h\right] \qquad M\left[\tfrac{m}{2} + 1 \,..\, m,\ h \,..\, n\right]$$

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):

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

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 ***$O(m + n\log m)$ 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.

6.5 Saving more time: 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 $m \times n$ array as input, SMAWK finds the leftmost smallest element of every row in only $\boldsymbol{O(n+m)}$ ***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***[2] 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 is sometimes called the *Monge property* or the *quadrangle inequality* or *submodularity*.

Lemma: *Every Monge array is totally monotone.*

[2] 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 (in context) should be translated as "Treatise on the Theory of Holes and Dirt".

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. □

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.

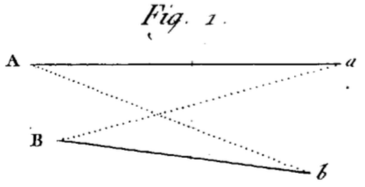
Lemma: *An array M is Monge if and only if $M[i, j] + M[i+1, j+1] \le M[i, j+1] + M[i+1, j]$ for all indices i and j.*

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:

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

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 $p_{i'} q_j$.

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



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: *The following arrays are Monge:*
(a) Any array with constant rows.
(b) Any array with constant columns.
(c) Any array that is all 0s except for an upper-right rectangular block of 1s.
(d) Any array that is all 0s except for an lower-left rectangular block of 1s.
(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.)

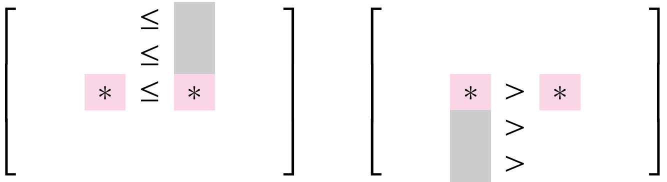
6.6 The SMAWK algorithm

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 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.

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

Call an array entry $M[i,j]$ ***dead*** if we have enough information to conclude that $LM[i] \ne j$. 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.



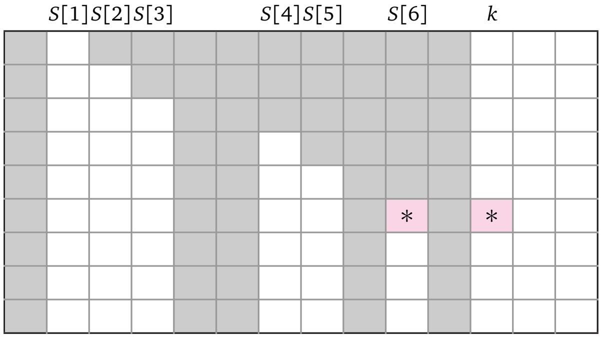
The following algorithm maintains a stack of column indices in an array $S[1..m]$ (yes, really m, the number of *rows*) and an index t indicating the number of indices on the stack.

```
REDUCE($M[1..m, 1..n]$):
  $t \leftarrow 1$
  $S[t] \leftarrow 1$
  for $k \leftarrow 1$ to $n$
      while $t > 0$ and $M[t, S[t]] \ge M[t, k]$
          $t \leftarrow t - 1$        ⟨⟨pop⟩⟩
      if $t < m$
          $t \leftarrow t + 1$
          $S[t] \leftarrow k$        ⟨⟨push k⟩⟩
  return $S[1..t]$
```

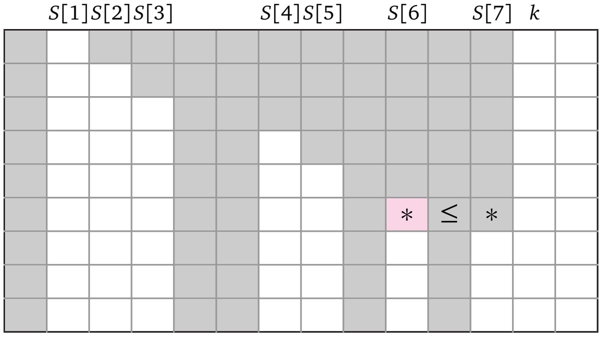
This algorithm maintains three important invariants:

- $S[1\,..\,t]$ is sorted in increasing order.
- For all $1 \leq j \leq t$, 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.

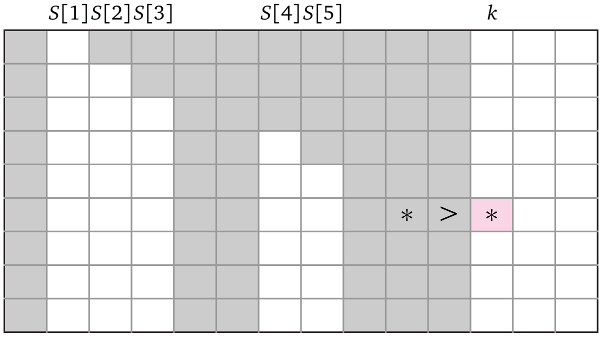
The first invariant follows immediately from the fact that indices are pushed onto the stack in increasing order. The second and third are more subtle, but both follow inductively from the total monotonicity of the array. The following figure shows a typical state of the algorithm when it is about to compare the entries $M[t, S[t]]$ and $M[t, k]$ (indicated by stars). Dead entries are indicated in gray.



If $M[t, S[t]] < M[t, k]$, then $M[t, k]$ and everything above it is dead, so we can safely push column k onto the stack (unless we already have $t = n$, in which case every entry in column k is dead) and then increment k.



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.



In both cases, all 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 a column 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 $S[1\,..\,t]$.

Finally, the main SMAWK algorithm first REDUCEs the input array (if it has fewer rows than columns), then recursively computes the leftmost row minima in every other, and finally fills in the rest of the row minima in $O(m+n)$ additional time. The argument of the recursive call is an array with exactly $m/2$ rows and at most $\min\{n, m\}$ columns, so the running time obeys the following recurrence:

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

The algorithm needs $O(n)$ time to REDUCE the input array to at most m rows, after which every recursive call reduces the (worst-case) number of rows and columns by a factor of two. We conclude that SMAWK runs in ***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.

6.7 Using SMAWK

Finally, let's consider an (admittedly artificial) example of a dynamic programming algorithm that can be improved by SMAWK. Recall the Shimbel-Bellman-Ford algorithm for computing shortest paths:

```
SHIMBEL($V, E, w$):
  $dist(s) \leftarrow 0$
  for every vertex $v \neq s$
      $dist(v) \leftarrow \infty$
  repeat $V - 1$ times
      for every edge $u{\to}v$
          if $dist(v) > dist(u) + w(u{\to}v)$
              $dist(v) \leftarrow dist(u) + w(u{\to}v)$
```

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 the natural dynamic-programming algorithm to compute shortest paths.)

```
MODSHIMBEL($V, E, w$):
  $dist(s) \leftarrow 0$
  for every vertex $v \neq s$
      $dist(v) \leftarrow \infty$
  repeat $V - 1$ times
      for every vertex $v$
          $mind(v) \leftarrow \min_u (dist(u) + w(u{\to}v))$
      for every vertex $v$
          $dist(v) \leftarrow \min\{dist(v), mind(v)\}$
```

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

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

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

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

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

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

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*(*n*²) *time*.**[3]

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.

6.8 Saving Time: Four Russians

Some day. Maybe.

Exercises

1. Describe an algorithm to compute the edit distance between two strings $A[1\,..\,m]$ and $B[1\ldots n]$ 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.]*

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

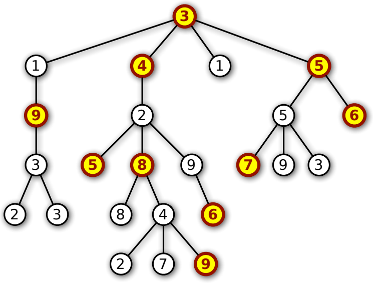
[3] Yes, we could also achieve this running time using Disjktra's algorithm with Fibonacci heaps.

(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.

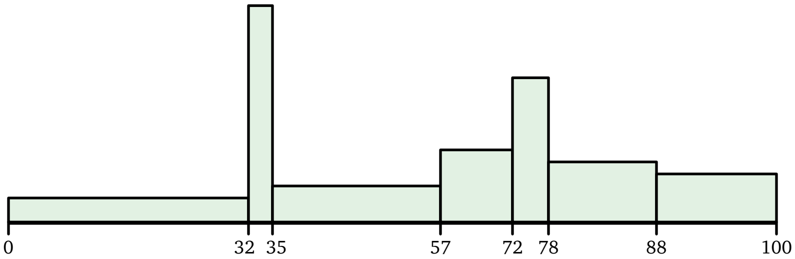
(b) Consider the two-dimensional matrix $M[1\,..\,n, 1\,..\,n]$ defined as follows:

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

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.[4] In particular, b_0 must be the minimum value in S, and b_k must be the maximum value in S. Informally, you are trying to compute a histogram with k rectangles whose areas are as close to equal as possible.

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 $B[k] = S[n]$.

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