A Class Note on Basic Algorithmic Techniques

Kun-Mao Chao[1,2,3]

[1]Graduate Institute of Biomedical Electronics and Bioinformatics
[2]Department of Computer Science and Information Engineering
[3]Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106

September 24, 2008

An *algorithm* is a step-by-step procedure for solving a problem by a computer. Although the act of designing an algorithm is considered as an art and can never be automated, its general strategies are learnable. Here we introduce a few frameworks of computer algorithms including greedy algorithms, divide-and-conquer strategies, and dynamic programming.

This note starts with the definition of algorithms and their complexity in Section 1. We introduce the asymptotic O-notation used in the analysis of the running time and space of an algorithm. Two tables are used to demonstrate that the asymptotic complexity of an algorithm will ultimately determine the size of problems that can be solved by the algorithm.

Then, we introduce greedy algorithms in Section 2. For some optimization problems, greedy algorithms are more efficient. A greedy algorithm pursues the best choice at the moment in the hope that it will lead to the best solution in the end. It works quite well for a wide range of problems. Huffman's algorithm is used as an example of a greedy algorithm.

Section 3 describes another common algorithmic technique, called divide-and-conquer. This strategy divides the problem into smaller parts, conquers each part individually, and then combines them to form a solution for the whole. We use the mergesort algorithm to illustrate the divide-and-conquer algorithm design

paradigm.

Following its introduction by Needleman and Wunsch, dynamic programming has become a major algorithmic strategy for many optimization problems in sequence comparison. The development of a dynamic-programming algorithm has three basic components: the recurrence relation for defining the value of an optimal solution, the tabular computation for computing the value of an optimal solution, and the backtracking procedure for delivering an optimal solution. In Section 4, we introduce these basic ideas by developing dynamic-programming solutions for problems from different application areas, including the maximum-sum segment problem, the longest increasing subsequence problem, and the longest common subsequence problem.

1 Algorithms and Their Complexity

An *algorithm* is a step-by-step procedure for solving a problem by a computer. When an algorithm is executed by a computer, the central processing unit (CPU) performs the operations and the memory stores the program and data.

Let n be the size of the input, the output, or their sum. The time or space complexity of an algorithm is usually denoted as a function $f(n)$. Table 1 calculates the time needed if the function stands for the number of operations required by an algorithm, and we assume that the CPU performs one million operations per second.

Exponential algorithms grow pretty fast and become impractical even when n is small. For those quadratic and cubic functions, they grow faster than the linear functions. The constant and log factor matter, but are mostly acceptable in practice. As a rule of thumb, algorithms with a quadratic time complexity or higher are often impractical for large data sets.

Table 2 further shows the growth of the input size solvable by polynomial and exponential time algorithms with improved computers. Even with a million-times faster computer, the 10^n algorithm only adds 6 to the input size, which makes it hopeless for handling a moderate-size input.

These observations lead to the definition of the O-notation, which is very useful for the analysis of algorithms. We say $f(n) = O(g(n))$ if and only if there exist two positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. In other words, for sufficiently large n, $f(n)$ can be bounded by $g(n)$ times a constant. In this kind of asymptotic analysis, the most crucial part is the order of the function, not the constant. For example, if $f(n) = 3n^2 + 5n$, we can say $f(n) = O(n^2)$ by

Table 1: The time needed by the functions where we assume one million operations per second.

$f(n)$	$n = 10$	$n = 100$	$n = 100000$
$30n$	0.0003 second	0.003 second	3 seconds
$100n\log_{10} n$	0.001 second	0.02 second	50 seconds
$3n^2$	0.0003 second	0.03 second	8.33 hours
n^3	0.001 second	1 second	31.71 years
10^n	2.78 hours	3.17×10^{84} centuries	3.17×10^{99984} centuries

letting $c = 4$ and $n_0 = 10$. By definition, it is also correct to say $n^2 = O(n^3)$, but we always prefer to choose a tighter order if possible. On the other hand, $10^n \neq O(n^x)$ for any integer x. That is, an exponential function cannot be bounded above by any polynomial function.

Table 2: The growth of the input size solvable in an hour as the computer runs faster.

$f(n)$	Present speed	1000-times faster	10^6-times faster
n	x_1	$1000x_1$	$10^6 x_1$
n^2	x_2	$31.62x_2$	$10^3 x_2$
n^3	x_3	$10x_3$	$10^2 x_3$
10^n	x_4	$x_4 + 3$	$x_4 + 6$

2 Greedy Algorithms

A greedy method works in stages. It always makes a locally optimal (*greedy*) choice at each stage. Once a choice has been made, it cannot be withdrawn, even if later we realize that it is a poor decision. In other words, this greedy choice may

or may not lead to a globally optimal solution, depending on the characteristics of the problem.

It is a very straightforward algorithmic technique and has been used to solve a variety of problems. In some situations, it is used to solve the problem exactly. In others, it has been proved to be effective in approximation.

What kind of problems are suitable for a greedy solution? There are two ingredients for an optimization problem to be exactly solved by a greedy approach. One is that it has the so-called greedy-choice property, meaning that a locally optimal choice can reach a globally optimal solution. The other is that it satisfies the principle of optimality, *i.e.*, each solution substructure is optimal. We use Huffman coding, a frequency-dependent coding scheme, to illustrate the greedy approach.

2.1 Huffman Codes

Suppose we are given a very long DNA sequence where the occurrence probabilities of nucleotides A (adenine), C (cytosine), G (guanine), T (thymine) are 0.1, 0.1, 0.3, and 0.5, respectively. In order to store it in a computer, we need to transform it into a binary sequence, using only 0's and 1's. A trivial solution is to encode A, C, G, and T by "00," "01," "10," and "11," respectively. This representation requires two bits per nucleotide. The question is "Can we store the sequence in a more compressed way?" Fortunately, by assigning longer codes for frequent nucleotides G and T, and shorter codes for rare nucleotides A and C, it can be shown that it requires less than two bits per nucleotide on average.

In 1952, Huffman proposed a greedy algorithm for building up an optimal way of representing each letter as a binary string. It works in two phases. In phase one, we build a binary tree based on the occurrence probabilities of the letters. To do so, we first write down all the letters, together with their associated probabilities. They are initially the unmarked terminal nodes of the binary tree that we will build up as the algorithm proceeds. As long as there is more than one unmarked node left, we repeatedly find the two unmarked nodes with the smallest probabilities, mark them, create a new unmarked internal node with an edge to each of the nodes just marked, and set its probability as the sum of the probabilities of the two nodes.

The tree building process is depicted in Figure 1. Initially, there are four unmarked nodes with probabilities 0.1, 0.1, 0.3, and 0.5. The two smallest ones are with probabilities 0.1 and 0.1. Thus we mark these two nodes and create a new node with probability 0.2 and connect it to the two nodes just marked. Now we have three unmarked nodes with probabilities 0.2, 0.3, and 0.5. The two smallest

ones are with probabilities 0.2 and 0.3. They are marked and a new node connecting them with probabilities 0.5 is created. The final iteration connects the only two unmarked nodes with probabilities 0.5 and 0.5. Since there is only one unmarked node left, *i.e.*, the root of the tree, we are done with the binary tree construction.

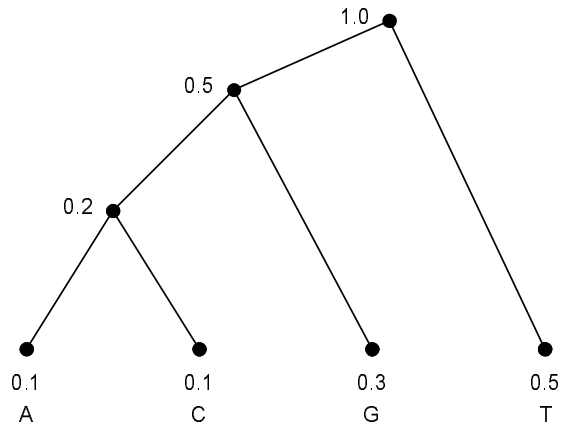

Figure 1: Building a binary tree based on the occurrence probabilities of the letters.

After the binary tree is built in phase one, the second phase is to assign the binary strings to the letters. Starting from the root, we recursively assign the value "zero" to the left edge and "one" to the right edge. Then for each leaf, *i.e.*, the letter, we concatenate the 0's and 1's from the root to it to form its binary string representation. For example, in Figure 2 the resulting codewords for A, C, G, and T are "000," "000," "01," and "1," respectively. By this coding scheme, a 20-nucleotide DNA sequence "GTTGTTATCGTTTATGTGGC" will be represented as a 34-bit binary sequence "01110111000100101111000010110101001." In general, since $3 \times 0.1 + 3 \times 0.1 + 2 \times 0.3 + 1 \times 0.5 = 1.7$, we conclude that, by Huffman coding techniques, each nucleotide requires 1.7 bits on average, which is superior to 2 bits by a trivial solution. Notice that in a Huffman code, no codeword is also a prefix of any other codeword. Therefore we can decode a binary sequence without any ambiguity. For example, if we are given "01110111000100101111000010110101001," we decode the binary sequence as "01" (G), "1" (T), "1" (T), "01" (G), and so forth.

The correctness of Huffman's algorithm lies in two properties: (1) greedy-choice property and (2) optimal-substructure property. It can be shown that there exists an optimal binary code in which the codewords for the two smallest-probability

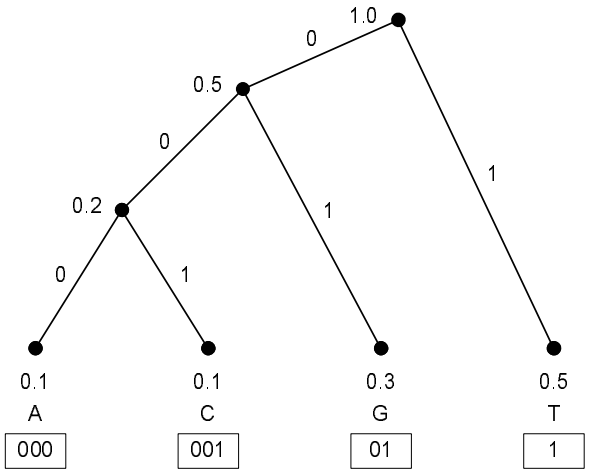

Figure 2: Huffman code assignment.

nodes have the same length and differ only in the last bit. That's the reason why we can contract them greedily without missing the path to the optimal solution. Besides, after contraction, the optimal-substructure property allows us to consider only those unmarked nodes.

Let n be the number of letters under consideration. For DNA, n is 4 and for English, n is 26. Since a heap can be used to maintain the minimum dynamically in $O(\log n)$ time for each insertion or deletion, the time complexity of Huffman's algorithm is $O(n \log n)$.

3 Divide-and-Conquer Strategies

The divide-and-conquer strategy *divides* the problem into a number of smaller subproblems. If the subproblem is small enough, it *conquers* it directly. Otherwise, it *conquers* the subproblem recursively. Once the solution to each subproblem has been done, it combines them together to form a solution to the original problem.

One of the well-known applications of the divide-and-conquer strategy is the design of sorting algorithms. We use mergesort to illustrate the divide-and-conquer algorithm design paradigm.

3.1 Mergesort

Given a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$, the sorting problem is to sort these numbers into a nondecreasing sequence. For example, if the given sequence is $\langle 65, 16, 25, 85, 12, 8, 36, 77 \rangle$, then its sorted sequence is $\langle 8, 12, 16, 25, 36, 65, 77, 85 \rangle$.

To sort a given sequence, mergesort splits the sequence into half, sorts each of them recursively, then combines the resulting two sorted sequences into one sorted sequence. Figure 3 illustrates the dividing process. The original input sequence consists of eight numbers. We first divide it into two smaller sequences, each consisting of four numbers. Then we divide each four-number sequence into two smaller sequences, each consisting of two numbers. Here we can sort the two numbers by comparing them directly, or divide it further into two smaller sequences, each consisting of only one number. Either way we'll reach the boundary cases where sorting is trivial. Notice that a sequential recursive process won't expand the subproblems simultaneously, but instead it solves the subproblems at the same recursion depth one by one.

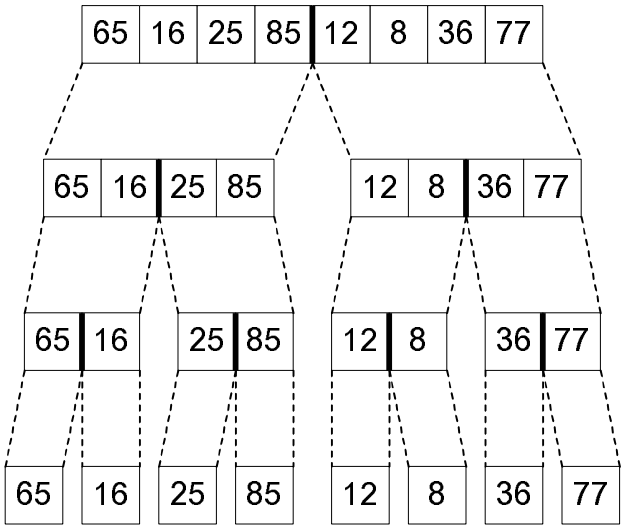

Figure 3: The top-down dividing process of mergesort.

How to combine the solutions to the two smaller subproblems to form a solution to the original problem? Let us consider the process of merging two sorted sequences into a sorted output sequence. For each merging sequence, we maintain a cursor pointing to the smallest element not yet included in the output sequence.

At each iteration, the smaller of these two smallest elements is removed from the merging sequence and added to the end of the output sequence. Once one merging sequence has been exhausted, the other sequence is appended to the end of the output sequence. Figure 4 depicts the merging process. The merging sequences are $\langle 16, 25, 65, 85\rangle$ and $\langle 8, 12, 36, 77\rangle$. The smallest elements of the two merging sequences are 16 and 8. Since 8 is a smaller one, we remove it from the merging sequence and add it to the output sequence. Now the smallest elements of the two merging sequences are 16 and 12. We remove 12 from the merging sequence and append it to the output sequence. Then 16 and 36 are the smallest elements of the two merging sequences, thus 16 is appended to the output list. Finally, the resulting output sequence is $\langle 8, 12, 16, 25, 36, 65, 77, 85\rangle$. Let N and M be the lengths of the two merging sequences. Since the merging process scans the two merging sequences linearly, its running time is therefore $O(N+M)$ in total.

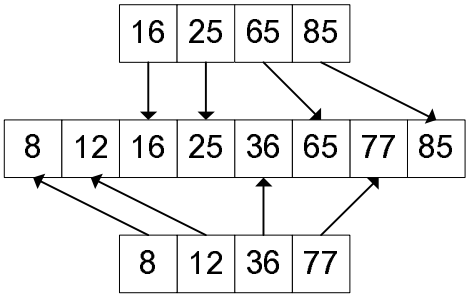

Figure 4: The merging process of mergesort.

After the top-down dividing process, mergesort accumulates the solutions in a bottom-up fashion by combining two smaller sorted sequences into a larger sorted sequence as illustrated in Figure 5. In this example, the recursion depth is $\lceil \log_2 8 \rceil = 3$. At recursion depth 3, every single element is itself a sorted sequence. They are merged to form sorted sequences at recursion depth 2: $\langle 16, 65\rangle$, $\langle 25, 85\rangle$, $\langle 8, 12\rangle$, and $\langle 36, 77\rangle$. At recursion depth 1, they are further merged into two sorted sequences: $\langle 16, 25, 65, 85\rangle$ and $\langle 8, 12, 36, 77\rangle$. Finally, we merge these two sequences into one sorted sequence: $\langle 8, 12, 16, 25, 36, 65, 77, 85\rangle$.

It can be easily shown that the recursion depth of mergesort is $\lceil \log_2 n \rceil$ for sorting n numbers, and the total time spent for each recursion depth is $O(n)$. Thus, we conclude that mergesort sorts n numbers in $O(n \log n)$ time.

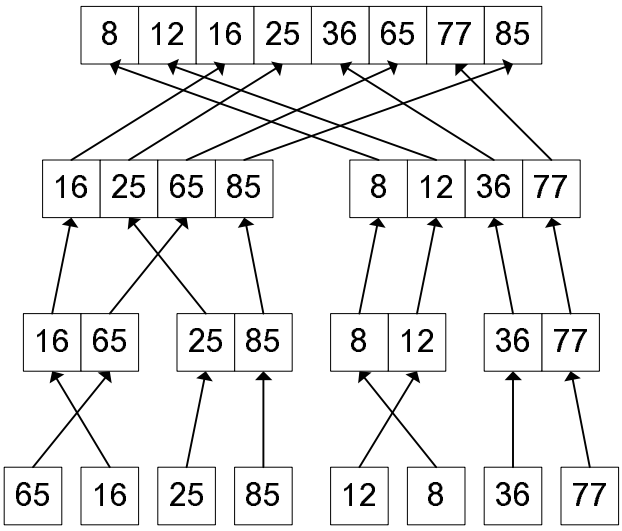

Figure 5: Accumulating the solutions in a bottom-up manner.

4 Dynamic Programming

Dynamic programming is a class of solution methods for solving sequential decision problems with a compositional cost structure. It is one of the major paradigms of algorithm design in computer science. Like the usage in *linear programming*, the word "*programming*" refers to *finding an optimal plan* of action, rather than *writing programs*. The word "*dynamic*" in this context conveys the idea that choices may depend on the current state, rather than being decided ahead of time.

Typically, dynamic programming is applied to optimization problems. In such problems, there exist many possible solutions. Each solution has a value, and we wish to find a solution with the optimum value. There are two ingredients for an optimization problem to be suitable for a dynamic-programming approach. One is that it satisfies the principle of optimality, *i.e.*, each solution substructure is optimal. Greedy algorithms require this very same ingredient, too. The other ingredient is that it has overlapping subproblems, which has the implication that it can be solved more efficiently if the solutions to the subproblems are recorded. If the subproblems are not overlapping, a divide-and-conquer approach is the choice.

The development of a dynamic-programming algorithm has three basic components: the recurrence relation for defining the value of an optimal solution, the tabular computation for computing the value of an optimal solution, and the back-

tracking procedure for delivering an optimal solution. Here we introduce these basic ideas by developing dynamic-programming solutions for problems from different application areas.

First of all, the Fibonacci numbers are used to demonstrate how a tabular computation can avoid recomputation. Then we use three classic problems, namely, the maximum-sum segment problem, the longest increasing subsequence problem, and the longest common subsequence problem, to explain how dynamic-programming approaches can be used to solve the sequence-related problems.

4.1 Fibonacci Numbers

The Fibonacci numbers were first created by Leonardo Fibonacci in 1202. It is a simple series, but its applications are nearly everywhere in nature. It has fascinated mathematicians for more than 800 years. The *Fibonacci numbers* are defined by the following recurrence:

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2. \end{cases}$$

By definition, the sequence goes like this: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, and so forth. Given a positive integer n, how would you compute F_n? You might say that it can be easily solved by a straightforward divide-and-conquer method based on the recurrence. That's right. But is it efficient? Take the computation of F_{10} for example (see Figure 6). By definition, F_{10} is derived by adding up F_9 and F_8. What about the values of F_9 and F_8? Again, F_9 is derived by adding up F_8 and F_7; F_8 is derived by adding up F_7 and F_6. Working toward this direction, we'll finally reach the values of F_1 and F_0, *i.e.*, the end of the recursive calls. By adding them up backwards, we have the value of F_{10}. It can be shown that the number of recursive calls we have to make for computing F_n is exponential in n.

Those who are ignorant of history are doomed to repeat it. A major drawback of this divide-and-conquer approach is to solve many of the subproblems repeatedly. A tabular method solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Figure 7 explains that F_n can be computed in $O(n)$ steps by a tabular computation. It should be noted that F_n can be computed in just $O(\log n)$ steps by applying matrix computation.

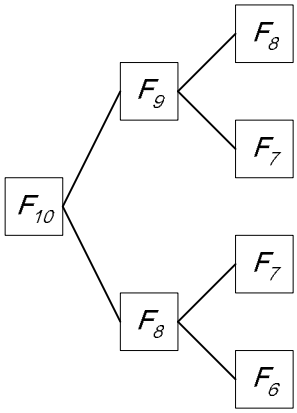

Figure 6: Computing F_{10} by divide-and-conquer.

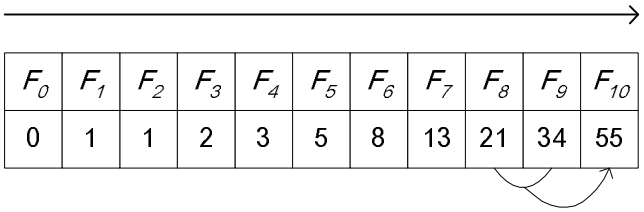

Figure 7: Computing F_{10} by a tabular computation.

4.2 The Maximum-Sum Segment Problem

Given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, the maximum-sum segment problem is to find, in A, a consecutive subsequence, *i.e.*, a substring or segment, with the maximum sum. For each position i, we can compute the maximum-sum segment ending at that position in $O(i)$ time. Therefore, a naive algorithm runs in $\sum_{i=1}^{n} O(i) = O(n^2)$ time.

Now let us describe a more efficient dynamic-programming algorithm for this problem. Define $S[i]$ to be the maximum sum of segments ending at position i of A. The value $S[i]$ can be computed by the following recurrence:

$$S[i] = \begin{cases} a_i + \max\{S[i-1], 0\} & \text{if } i > 1, \\ a_1 & \text{if } i = 1. \end{cases}$$

If $S[i-1] < 0$, concatenating a_i with its previous elements will give a smaller

sum than a_i itself. In this case, the maximum-sum segment ending at position i is a_i itself.

By a tabular computation, each $S[i]$ can be computed in constant time for i from 1 to n, therefore all S values can be computed in $O(n)$ time. During the computation, we record the largest S entry computed so far in order to report where the maximum-sum segment ends. We also record the traceback information for each position i so that we can trace back from the end position of the maximum-sum segment to its start position. If $S[i-1] > 0$, we need to concatenate with previous elements for a larger sum, therefore the traceback symbol for position i is "$\leftarrow$." Otherwise, "$\uparrow$" is recorded. Once we have computed all S values, the traceback information is used to construct the maximum-sum segment by starting from the largest S entry and following the arrows until a "$\uparrow$" is reached. For example, in Figure 8, $A = \langle 3, 2, -6, 5, 2, -3, 6, -4, 2\rangle$. By computing from $i = 1$ to $i = n$, we have $S = \langle 3, 5, -1, 5, 7, 4, 10, 6, 8\rangle$. The maximum S entry is $S[7]$ whose value is 10. By backtracking from $S[7]$, we conclude that the maximum-sum segment of A is $\langle 5, 2, -3, 6\rangle$, whose sum is 10.

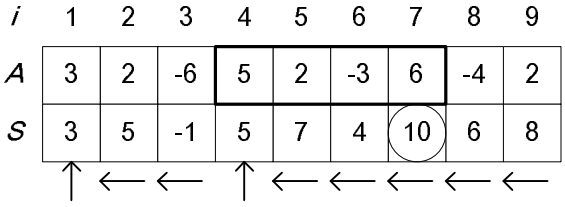

Figure 8: Finding a maximum-sum segment.

Let *prefix sum* $P[i] = \sum_{j=1}^{i} a_j$ be the sum of the first i elements. It can be easily seen that $\sum_{k=i}^{j} a_k = P[j] - P[i-1]$. Therefore, if we wish to compute for a given position the maximum-sum segment ending at it, we could just look for a minimum prefix sum ahead of this position. This yields another linear-time algorithm for the maximum-sum segment problem.

4.3 Longest Increasing Subsequences

Given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n\rangle$, the longest increasing subsequence problem is to find an increasing subsequence in A whose length is maximum. Without loss of generality, we assume that these numbers are distinct. Formally speaking, given a sequence of distinct real numbers $A = \langle a_1, a_2, \ldots, a_n\rangle$, se-

quence $B = \langle b_1, b_2, \ldots, b_k \rangle$ is said to be a subsequence of A if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of A such that for all $j = 1, 2, \ldots, k$, we have $a_{i_j} = b_j$. In other words, B is obtained by deleting zero or more elements from A. We say that the subsequence B is increasing if $b_1 < b_2 < \ldots < b_k$. The longest increasing subsequence problem is to find a maximum-length increasing subsequence of A.

For example, suppose $A = \langle 4,\ 8,\ 2,\ 7,\ 3,\ 6,\ 9,\ 1,\ 10,\ 5 \rangle$, both $\langle 2,3,6 \rangle$ and $\langle 2,7,9,10 \rangle$ are increasing subsequences of A, whereas $\langle 8,7,9 \rangle$ (not increasing) and $\langle 2,3,5,7 \rangle$ (not a subsequence) are not.

Note that we may have more than one longest increasing subsequence, so we use "*a* longest increasing subsequence" instead of "*the* longest increasing subsequence." Let $L[i]$ be the length of a longest increasing subsequence ending at position i. They can be computed by the following recurrence:

$$L[i] = \begin{cases} 1 + \max_{j=0,\ldots,i-1}\{L[j] \mid a_j < a_i\} & \text{if } i > 0, \\ 0 & \text{if } i = 0. \end{cases}$$

Here we assume that a_0 is a dummy element and smaller than any element in A, and $L[0]$ is equal to 0. By tabular computation for every i from 1 to n, each $L[i]$ can be computed in $O(i)$ steps. Therefore, they require in total $\sum_{i=1}^{n} O(i) = O(n^2)$ steps. For each position i, we use an array P to record the index of the best previous element for the current element to concatenate with. By tracing back from the element with the largest L value, we derive a longest increasing subsequence.

Figure 9 illustrates the process of finding a longest increasing subsequence of $A = \langle 4,\ 8,\ 2,\ 7,\ 3,\ 6,\ 9,\ 1,\ 10,\ 5 \rangle$. Take $i = 4$ for instance, where $a_4 = 7$. Its previous smaller elements are a_1 and a_3, both with L value equaling 1. Therefore, we have $L[4] = L[1] + 1 = 2$, meaning that the length of a longest increasing subsequence ending at position 4 is of length 2. Indeed, both $\langle a_1, a_4 \rangle$ and $\langle a_3, a_4 \rangle$ are an increasing subsequence ending at position 4. In order to trace back the solution, we use array P to record which entry contributes the maximum to the current L value. Thus, $P[4]$ can be 1 (standing for a_1) or 3 (standing for a_3). Once we have computed all L and P values, the maximum L value is the length of a longest increasing subsequence of A. In this example, $L[9] = 5$ is the maximum. Tracing back from $P[9]$, we have found a longest increasing subsequence $\langle a_3, a_5, a_6, a_7, a_9 \rangle$, *i.e.*, $\langle 2,3,6,9,10 \rangle$.

In the following, we briefly describe a more efficient dynamic-programming algorithm for delivering a longest increasing subsequence. A crucial observation is that it suffices to store only those smallest ending elements for all possible

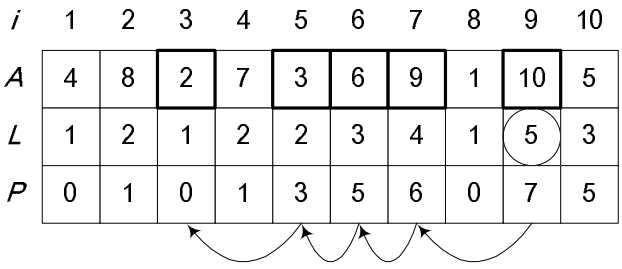

Figure 9: An $O(n^2)$-time algorithm for finding a longest increasing subsequence.

lengths of the increasing subsequences. For example, in Figure 9, there are three entries whose L value is 2, namely $a_2 = 8$, $a_4 = 7$, and $a_5 = 3$, where a_5 is the smallest. Any element after position 5 that is larger than a_2 or a_4 is also larger than a_5. Therefore, a_5 can replace the roles of a_2 and a_4 after position 5.

Let $SmallestEnd[k]$ denote the smallest ending element of all possible increasing subsequences of length k ending before the current position i. The algorithm proceeds for i from 1 to n. How do we update $SmallestEnd[k]$ when we consider a_i? By definition, it is easy to see that the elements in $SmallestEnd$ are in increasing order. In fact, a_i will affect only one entry in $SmallestEnd$. If a_i is larger than all the elements in $SmallestEnd$, then we can concatenate a_i to the longest increasing subsequence computed so far. That is, one more entry is added to the end of $SmallestEnd$. A backtracking pointer is recorded by pointing to the previous last element of $SmallestEnd$. Otherwise, let $SmallestEnd[k']$ be the smallest element that is larger than a_i. We replace $SmallestEnd[k']$ by a_i because now we have a smaller ending element of an increasing subsequence of length k'.

Since $SmallestEnd$ is a sorted array, the above process can be done by a binary search. A binary search algorithm compares the query element with the middle element of the sorted array, if the query element is larger, then it searches the larger half recursively. Otherwise, it searches the smaller half recursively. Either way the size of the search space is shrunk by a factor of two. At position i, the size of $SmallestEnd$ is at most i. Therefore, for each position i, it takes $O(\log i)$ time to determine the appropriate entry to be updated by a_i. Therefore, in total we have an $O(n \log n)$-time algorithm for the longest increasing subsequence problem.

Figure 10 illustrates the process of finding a longest increasing subsequence of $A = \langle 4, 8, 2, 7, 3, 6, 9, 1, 10, 5 \rangle$. When $i = 1$, there is only one increasing subsequence, *i.e.*, $\langle 4 \rangle$. We have $SmallestEnd[1] = 4$. Since $a_2 = 8$ is larger than $SmallestEnd[1]$, we create a new entry $SmallestEnd[2] = 8$ and set the back-

tracking pointer $P[2] = 1$, meaning that a_2 can be concatenated with a_1 to form an increasing subsequence $\langle 4, 8\rangle$. When $a_3 = 2$ is encountered, its nearest larger element in *SmallestEnd* is $SmallestEnd[1] = 4$. We know that we now have an increasing subsequence $\langle 2\rangle$ of length 1. So $SmallestEnd[1]$ is changed from 4 to $a_3 = 2$ and $P[3] = 0$. When $i = 4$, we have $SmallestEnd[1] = 2 < a_4 = 7 < SmallestEnd[2] = 8$. By concatenating a_4 with $Smallest[1]$, we have a new increasing subsequence $\langle 2, 7\rangle$ of length 2 whose ending element is smaller than 8. Thus, $SmallestEnd[2]$ is changed from 8 to $a_4 = 7$ and $P[4] = 3$. Continue this way until we reach a_{10}. When a_{10} is encountered, we have $SmallestEnd[2] = 3 < a_{10} = 5 < SmallestEnd[3] = 6$. We set $SmallestEnd[3] = a_{10} = 5$ and $P[10] = 5$. Now the largest element in *SmallestEnd* is $SmallestEnd[5] = a_9 = 10$. We can trace back from a_9 by the backtracking pointers P and deliver a longest increasing subsequence $\langle a_3, a_5, a_6, a_7, a_9\rangle$, *i.e.*, $\langle 2, 3, 6, 9, 10\rangle$.

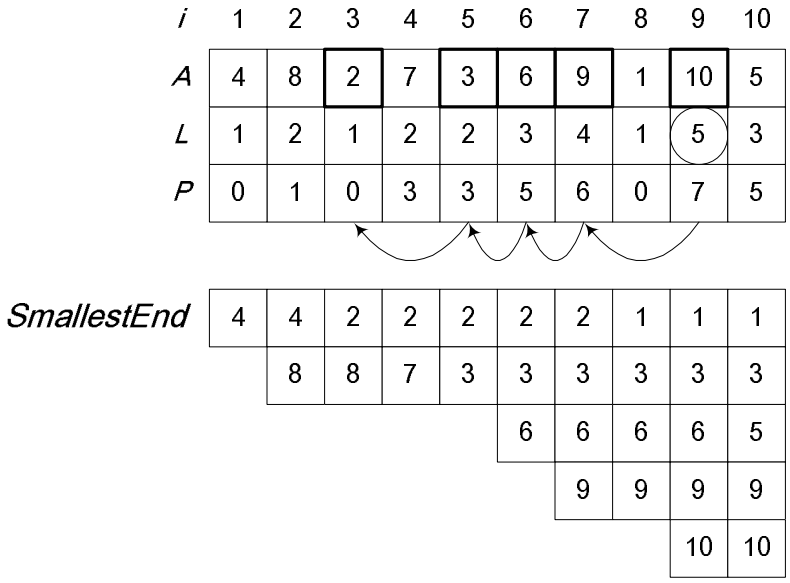

Figure 10: An $O(n \log n)$-time algorithm for finding a longest increasing subsequence.

4.4 Longest Common Subsequences

A subsequence of a sequence S is obtained by deleting zero or more elements from S. For example, $\langle P,R,E,D\rangle$, $\langle S,D,N\rangle$, and $\langle P,R,E,D,E,N,T\rangle$ are all subsequences of $\langle P,R,E,S,I,D,E,N,T\rangle$, whereas $\langle S,N,D\rangle$ and $\langle P,E,F\rangle$ are not.

Recall that, given two sequences, the longest common subsequence (LCS) problem is to find a subsequence that is common to both sequences and its length is maximized. For example, given two sequences

$$\langle P,R,E,S,I,D,E,N,T\rangle$$

and

$$\langle P,R,O,V,I,D,E,N,C,E\rangle,$$

$\langle P,R,D,N\rangle$ is a common subsequence of them, whereas $\langle P,R,V\rangle$ is not. Their LCS is $\langle P,R,I,D,E,N\rangle$.

Now let us formulate the recurrence for computing the length of an LCS of two sequences. We are given two sequences $A = \langle a_1,a_2,\ldots,a_m\rangle$, and $B = \langle b_1,b_2,\ldots,b_n\rangle$. Let $len[i,j]$ denote the length of an LCS between $\langle a_1,a_2,\ldots,a_i\rangle$ (a prefix of A) and $\langle b_1,b_2,\ldots,b_j\rangle$ (a prefix of B). They can be computed by the following recurrence:

$$len[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ len[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } a_i = b_j, \\ \max\{len[i,j-1], len[i-1,j]\} & \text{otherwise.} \end{cases}$$

In other words, if one of the sequences is empty, the length of their LCS is just zero. If a_i and b_j are the same, an LCS between $\langle a_1,a_2,\ldots,a_i\rangle$, and $\langle b_1,b_2,\ldots,b_j\rangle$ is the concatenation of an LCS of $\langle a_1,a_2,\ldots,a_{i-1}\rangle$ and $\langle b_1,b_2,\ldots,b_{j-1}\rangle$ and a_i. Therefore, $len[i,j] = len[i-1,j-1]+1$ in this case. If a_i and b_j are different, their LCS is equal to either an LCS of $\langle a_1,a_2,\ldots,a_i\rangle$, and $\langle b_1,b_2,\ldots,b_{j-1}\rangle$, or that of $\langle a_1,a_2,\ldots,a_{i-1}\rangle$, and $\langle b_1,b_2,\ldots,b_j\rangle$. Its length is thus the maximum of $len[i,j-1]$ and $len[i-1,j]$.

Figure 11 gives the pseudo-code for computing $len[i,j]$. For each entry (i,j), we retain the backtracking information in $prev[i,j]$. If $len[i-1,j-1]$ contributes the maximum value to $len[i,j]$, then we set $prev[i,j] =$"$\nwarrow$." Otherwise $prev[i,j]$ is set to be "$\uparrow$" or "$\leftarrow$" depending on which one of $len[i-1,j]$ and $len[i,j-1]$ contributes the maximum value to $len[i,j]$. Whenever there is a tie, any one of them will work. These arrows will guide the backtracking process upon reaching

```
Algorithm LCS_LENGTH(A = ⟨a_1, a_2, ..., a_m⟩, B = ⟨b_1, b_2, ..., b_n⟩)
begin
    for i ← 0 to m do len[i, 0] ← 0
    for j ← 1 to n do len[0, j] ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if a_i = b_j then
                len[i, j] ← len[i − 1, j − 1] + 1
                prev[i, j] ←"↖"
            else if len[i − 1, j] ≥ len[i, j − 1] then
                len[i, j] ← len[i − 1, j]
                prev[i, j] ←"↑"
            else
                len[i, j] ← len[i, j − 1]
                prev[i, j] ←"←"
    return len and prev
end
```

Figure 11: Computation of the length of an LCS of two sequences.

the terminal entry (m,n). Since the time spent for each entry is $O(1)$, the total running time of algorithm LCS_LENGTH is $O(mn)$.

Figure 12 illustrates the tabular computation. The length of an LCS of

$$\langle A,L,G,O,R,I,T,H,M\rangle$$

and

$$\langle A,L,I,G,N,M,E,N,T\rangle$$

is 4.

Besides computing the length of an LCS of the whole sequences, Figure 12 in fact computes the length of an LCS between each pair of prefixes of the two sequences. For example, by this table, we can also tell the length of an LCS between $\langle A,L,G,O,R\rangle$ and $\langle A,L,I,G\rangle$ is 3.

Once algorithm LCS_LENGTH reaches (m,n), the backtracking information retained in array *prev* allows us to find out which common subsequence contributes $len[m,n]$, the maximum length of an LCS of sequences A and B. Fig-

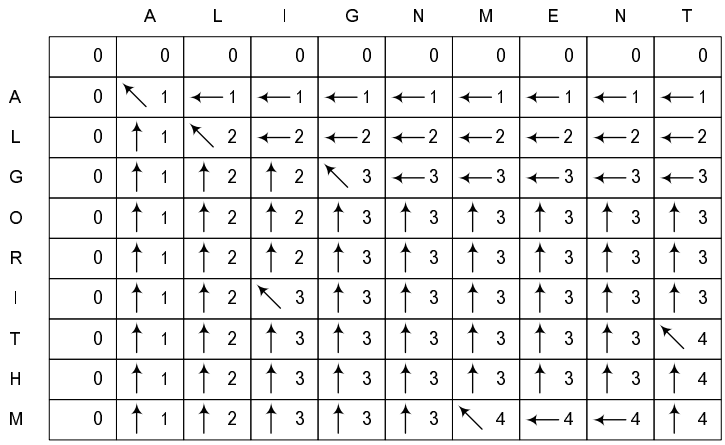

Figure 12: Tabular computation of the length of an LCS of $\langle A,L,G,O,R,I,T,H,M\rangle$ and $\langle A,L,I,G,N,M,E,N,T\rangle$.

ure 13 lists the pseudo-code for delivering an LCS. We trace back the dynamic-programming matrix from the entry (m,n) recursively following the direction of the arrow. Whenever a diagonal arrow "$\nwarrow$" is encountered, we append the current matched letter to the end of the LCS under construction. Algorithm LCS_OUTPUT takes $O(m+n)$ time in total since each recursive call reduces the indices i and/or j by one.

Figure 14 backtracks the dynamic-programming matrix computed in Figure 12. It outputs $\langle A,L,G,T\rangle$ (the shaded entries) as an LCS of

$$\langle A,L,G,O,R,I,T,H,M\rangle$$

and

$$\langle A,L,I,G,N,M,E,N,T\rangle.$$

Algorithm LCS_OUTPUT($A = \langle a_1, a_2, \ldots, a_m \rangle$, *prev*, *i*, *j*)
begin
  if $i = 0$ or $j = 0$ **then return**
  if *prev*$[i, j]$ =“↖” **then**
    LCS_OUTPUT(*A*, *prev*, $i-1$, $j-1$)
    print a_i
  else if *prev*$[i, j]$ =“↑” **then** LCS_OUTPUT(*A*, *prev*, $i-1$, j)
  else LCS_OUTPUT(*A*, *prev*, *i*, $j-1$)
end

Figure 13: Delivering an LCS.

		A	L	I	G	N	M	E	N	T
	0	0	0	0	0	0	0	0	0	0
A	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
L	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
G	0	↑ 1	↑ 2	↑ 2	↖ 3	← 3	← 3	← 3	← 3	← 3
O	0	↑ 1	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3
R	0	↑ 1	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3
I	0	↑ 1	↑ 2	↖ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3
T	0	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↖ 4
H	0	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 3	↑ 4
M	0	↑ 1	↑ 2	↑ 3	↑ 3	↑ 3	↖ 4	← 4	← 4	↑ 4

Figure 14: Backtracking process for finding an LCS of $\langle A, L, G, O, R, I, T, H, M \rangle$ and $\langle A, L, I, G, N, M, E, N, T \rangle$.