Construct the string-matching automaton for the pattern P = aabab and illustrate its operation on the text string T = aaababaabaababaab.

Exercises 32.3-2

Draw a state-transition diagram for a string-matching automaton for the pattern abababbababbababbabb over the alphabet $\Sigma = \{a, b\}$.

Exercises 32.3-3

We call a pattern P ***nonoverlappable*** if $P_k \sqsupset P_q$ implies $k = 0$ or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

Exercises 32.3-4: □

Given two patterns P and P', describe how to construct a finite automaton that determines all occurrences of *either* pattern. Try to minimize the number of states in your automaton.

Exercises 32.3-5

Given a pattern P containing gap characters (see Exercise 32.1-4), show how to build a finite automaton that can find an occurrence of P in a text T in $O(n)$ matching time, where $n = |T|$.

32.4 □ The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. Their algorithm avoids the computation of the transition function δ altogether, and its matching time is $\Theta(n)$ using just an auxiliary function $\pi[1 \text{ □ } m]$ precomputed from the pattern in time $\Theta(m)$. The array π allows the transition function δ to be computed efficiently (in an amortized sense) "on the fly" as needed. Roughly speaking, for any state $q = 0, 1, \ldots, m$ and any character a □ Σ, the value $\pi[q]$ contains the information that is independent of a and is needed to compute $\delta(q, a)$. (This remark will be clarified shortly.) Since the array π has only m entries, whereas δ has $\Theta(m\,|\Sigma|)$ entries, we save a factor of $|\Sigma|$ in the preprocessing time by computing π rather than δ.

The prefix function for a pattern

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid testing useless shifts in the naive pattern-matching algorithm or to avoid the precomputation of δ for a string-matching automaton.

Consider the operation of the naive string matcher. Figure 32.10(a) shows a particular shift s of a template containing the pattern P = ababaca against a text T. For this example, $q = 5$ of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Knowing these q text characters allows us to determine immediately that certain shifts are invalid. In the example of the figure, the shift $s + 1$ is necessarily invalid, since the first pattern character (a) would be aligned with a text character that is known to match with the second pattern character (b). The shift $s' = s + 2$ shown in part (b) of the figure, however, aligns the first three pattern characters with three text characters that must necessarily match. In general, it is useful to know the answer to the following question:

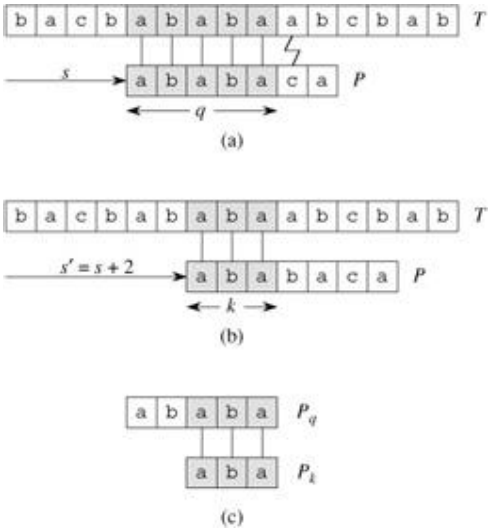

Figure 32.10: The prefix function π. *(a)* The pattern P = ababaca is aligned with a text T so that the first $q = 5$ characters match. Matching characters, shown shaded, are connected by vertical lines. *(b)* Using only our knowledge of the 5 matched characters, we can deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s + 2$ is consistent with everything we know about the text and therefore is potentially valid. *(c)* The useful information for such deductions can be precomputed by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of P_5 is P_3. This information is precomputed and represented in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$.

- Given that pattern characters $P[1 \,..\, q]$ match text characters $T[s + 1 \,..\, s + q]$, what is the least shift $s' > s$ such that

  (32.5) $P[1..k] = T[s' + 1..s' + k]$,

- where $s' + k = s + q$?

Such a shift s' is the first shift greater than s that is not necessarily invalid due to our knowledge of $T[s + 1 \; . \; . \; s + q]$. In the best case, we have that $s' = s + q$, and shifts $s + 1$, $s + 2$, . . . , $s + q - 1$ are all immediately ruled out. In any case, at the new shift s' we don't need to compare the first k characters of P with the corresponding characters of T, since we are guaranteed that they match by equation (32.5).

The necessary information can be precomputed by comparing the pattern against itself, as illustrated in Figure 32.10(c). Since $T[s' + 1 \; . \; . \; s' + k]$ is part of the known portion of the text, it is a suffix of the string P_q. Equation (32.5) can therefore be interpreted as asking for the largest $k < q$ such that $P_k \sqsupset P_q$. Then, $s' = s+(q - k)$ is the next potentially valid shift. It turns out to be convenient to store the number k of matching characters at the new shift s', rather than storing, say, $s' - s$. This information can be used to speed up both the naive string-matching algorithm and the finite-automaton matcher.

We formalize the precomputation required as follows. Given a pattern $P[1 \; . \; . \; m]$, the ***prefix function*** for the pattern P is the function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m - 1\}$ such that

$$\pi[q] = \max \{k : k < q \text{ and } P_k \sqsupset P_q\}.$$

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q. As another example, Figure 32.11(a) gives the complete prefix function π for the pattern abababacba.

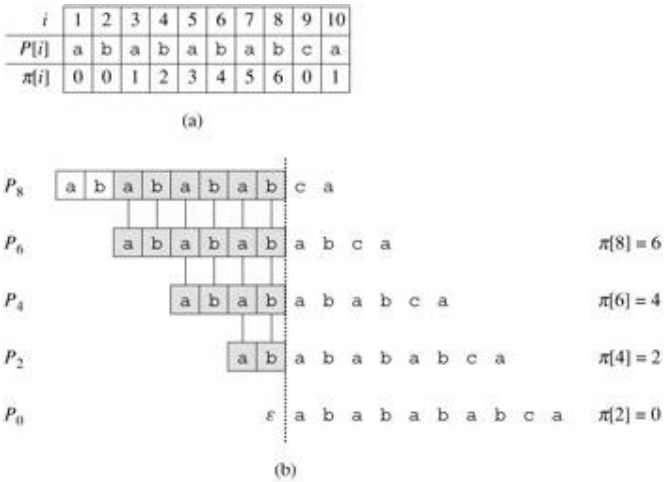

Figure 32.11: An illustration of Lemma 32.5 for the pattern P = abababacba and $q = 8$. *(a)* The π function for the given pattern. Since $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, and $\pi[2] = 0$, by iterating π we obtain $\pi^*[8] = \{6, 4, 2, 0\}$. *(b)* We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_8; this happens for $k = 6$, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_8. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_8. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : k < q \text{ and } P_k \sqsupset P_q\} = \{6, 4,2, 0\}$. The lemma claims that $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

The Knuth-Morris-Pratt matching algorithm is given in pseudocode below as the procedure KMP-MATCHER. It is mostly modeled after FINITE-AUTOMATON-MATCHER, as we

shall see. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π.

```
KMP-MATCHER(T, P)
 1 n ← length[T]
 2 m ← length[P]
 3 π ← COMPUTE-PREFIX-FUNCTION(P)
 4 q ← 0                          ▹Number of characters matched.
 5 for i ← 1 to n                 ▹Scan the text from left to right.
 6      do while q > 0 and P[q + 1] ≠ T[i]
 7            do q ← π[q]         ▹Next character does not match.
 8         if P[q + 1] = T[i]
 9            then q ← q + 1      ▹Next character matches.
10         if q = m                    ▹Is all of P matched?
11            then print "Pattern occurs with shift" i - m
12                 q ← π[q]        ▹Look for the next match.
COMPUTE-PREFIX-FUNCTION(P)
 1 m ← length[P]
 2 π[1] ← 0
 3 k ← 0
 4 for q ← 2 to m
 5      do while k > 0 and P[k + 1] ≠ P[q]
 6            do k ← π[k]
 7         if P[k + 1] = P[q]
 8            then k ← k + 1
 9         π[q] ← k
10 return π
```

We begin with an analysis of the running times of these procedures. Proving these procedures correct will be more complicated.

Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$, using the potential method of amortized analysis (see Section 17.3). We associate a potential of k with the current state k of the algorithm. This potential has an initial value of 0, by line 3. Line 6 decreases k whenever it is executed, since $\pi[k] < k$. Since $\pi[k] \geq 0$ for all k, however, k can never become negative. The only other line that affects k is line 8, which increases k by at most one during each execution of the **for** loop body. Since $k < q$ upon entering the **for** loop, and since q is incremented in each iteration of the **for** loop body, $k < q$ always holds. (This justifies the claim that $\pi[q] < q$ as well, by line 9.) We can pay for each execution of the **while** loop body on line 6 with the corresponding decrease in the potential function, since $\pi[k] < k$. Line 8 increases the potential function by at most one, so that the amortized cost of the loop body on lines 5-9 is $O(1)$. Since the number of outer-loop iterations is $\Theta(m)$, and since the final potential function is at least as great as the initial potential function, the total actual worst-case running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$.

A similar amortized analysis, using the value of q as the potential function, shows that the matching time of KMP-MATCHER is $\Theta(n)$.

Compared to FINITE-AUTOMATON-MATCHER, by using π rather than δ, we have reduced the time for preprocessing the pattern from $O(m\,|\Sigma|)$ to $\Theta(m)$, while keeping the actual matching time bounded by $\Theta(n)$.

Correctness of the prefix-function computation

We start with an essential lemma showing that by iterating the prefix function π, we can enumerate all the prefixes P_k that are proper suffixes of a given prefix P_q. Let

$$\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\},$$

where $\pi^{(i)}[q]$ is defined in terms of functional iteration, so that $\pi^{(0)}[q] = q$ and $\pi^{(i+1)}[q] = \pi[\pi^{(i)}[q]]$ for $i \geq 1$, and where it is understood that the sequence in $\pi^*[q]$ stops when $\pi^{(t)}[q] = 0$ is reached. The following lemma characterizes $\pi^*[q]$, as Figure 32.11 illustrates.

Lemma 32.5: (Prefix-function iteration lemma)

Let P be a pattern of length m with prefix function π. Then, for $q = 1, 2, \ldots, m$, we have $\pi^*[q] = \{k : k < q \text{ and } P_k \supset P_q\}$.

Proof We first prove that

(32.6) $i \in \pi^*[q]$ implies $P_i \supset P_q$.

If $i \in \pi^*[q]$, then $i = \pi^{(u)}[q]$ for some $u > 0$. We prove equation (32.6) by induction on u. For $u = 1$, we have $i = \pi[q]$, and the claim follows since $i < q$ and $P_{\pi[q]} \supset P_q$. Using the relations $\pi[i] < i$ and $P_{\pi[i]} \supset P_i$ and the transitivity of $<$ and $\supset$ establishes the claim for all i in $\pi^*[q]$. Therefore, $\pi^*[q] \subseteq \{k : k < q \text{ and } P_k \supset P_q\}$.

We prove that $\{k : k < q \text{ and } P_k \supset P_q\} \subseteq \pi^*[q]$ by contradiction. Suppose to the contrary that there is an integer in the set $\{k : k < q \text{ and } P_k \supset P_q\} - \pi^*[q]$, and let j be the largest such value. Because $\pi[q]$ is the largest value in $\{k : k < q \text{ and } P_k \supset P_q\}$ and $\pi[q] \in \pi^*[q]$, we must have $j < \pi[q]$, and so we let j' denote the smallest integer in $\pi^*[q]$ that is greater than j. (We can choose $j' = \pi[q]$ if there is no other number in $\pi^*[q]$ that is greater than j.) We have $P_j \supset P_q$ because $j \in \{k : k < q \text{ and } P_k \supset P_q\}$, and we have $P_{j'} \supset P_q$ because $j' \in \pi^*[q]$. Thus, $P_j \supset P_{j'}$ by Lemma 32.1, and j is the largest value less than j' with this property. Therefore, we must have $\pi[j'] = j$ and, since $j' \in \pi^*[q]$, we must have $j \in \pi^*[q]$ as well. This contradiction proves the lemma.

The algorithm COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ in order for $q = 1, 2, \ldots, m$. The computation of $\pi[1] = 0$ in line 2 of COMPUTE-PREFIX-FUNCTION is certainly correct, since $\pi[q] < q$ for all q. The following lemma and its corollary will be used to prove that COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ correctly for $q > 1$.

Lemma 32.6

Let P be a pattern of length m, and let π be the prefix function for P. For $q = 1, 2, \ldots, m$, if $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^*[q - 1]$.

Proof If $r = \pi[q] > 0$, then $r < q$ and $P_r \supset P_q$; thus, $r - 1 < q - 1$ and $P_{r-1} \supset P_{q-1}$ (by dropping the last character from P_r and P_q). By Lemma 32.5, therefore, $\pi[q] - 1 = r - 1 \in \pi^*[q - 1]$.

For $q = 2, 3, \ldots, m$, define the subset $E_{q-1} \subseteq \pi^*[q - 1]$ by

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q - 1] : P[k + 1] = P[q]\} \\ &= \{k : k < q - 1 \text{ and } P_k \supset P_{q-1} \text{ and } P[k + 1] = P[q]\} \text{ (by Lemma 32.5)} \\ &= \{k : k < q - 1 \text{ and } P_{k+1} \supset P_q\}. \end{aligned}$$

The set E_{q-1} consists of the values $k < q - 1$ for which $P_k \supset P_{q-1}$ and for which $P_{k+1} \supset P_q$, because $P[k + 1] = P[q]$. Thus, E_{q-1} consists of those values $k \in \pi^*[q - 1]$ such that we can extend P_k to P_{k+1} and get a proper suffix of P_q.

Corollary 32.7

Let P be a pattern of length m, and let π be the prefix function for P. For $q = 2, 3, \ldots, m$,

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset, \\ 1 + \max\{k \in E_{q-1}\} & \text{if } E_{q-1} \neq \emptyset. \end{cases}$$

Proof If E_{q-1} is empty, there is no $k \in \pi^*[q - 1]$ (including $k = 0$) for which we can extend P_k to P_{k+1} and get a proper suffix of P_q. Therefore $\pi[q] = 0$.

If E_{q-1} is nonempty, then for each $k \in E_{q-1}$ we have $k + 1 < q$ and $P_{k+1} \supset P_q$. Therefore, from the definition of $\pi[q]$, we have

(32.7) $$\pi[q] \geq 1 + \max\{k \in E_{q-1}\}.$$

Note that $\pi[q] > 0$. Let $r = \pi[q] - 1$, so that $r + 1 = \pi[q]$. Since $r + 1 > 0$, we have $P[r + 1] = P[q]$. Furthermore, by Lemma 32.6, we have $r \in \pi^*[q - 1]$. Therefore, $r \in E_{q-1}$, and so $r \leq \max\{k \in E_{q-1}\}$ or, equivalently,

(32.8) $$\pi[q] \leq 1 + \max\{k \in E_{q-1}\}.$$

Combining equations (32.7) and (32.8) completes the proof.

We now finish the proof that COMPUTE-PREFIX-FUNCTION computes π correctly. In the procedure COMPUTE-PREFIX-FUNCTION, at the start of each iteration of the **for** loop of

lines 4-9, we have that $k = \pi[q - 1]$. This condition is enforced by lines 2 and 3 when the loop is first entered, and it remains true in each successive iteration because of line 9. Lines 5-8 adjust k so that it now becomes the correct value of $\pi[q]$. The loop on lines 5-6 searches through all values $k \in \pi^*[q - 1]$ until one is found for which $P[k + 1] = P[q]$; at that point, k is the largest value in the set E_{q-1}, so that, by Corollary 32.7, we can set $\pi[q]$ to $k + 1$. If no such k is found, $k = 0$ in line 7. If $P[1] = P[q]$, then we should set both k and $\pi[q]$ to 1; otherwise we should leave k alone and set $\pi[q]$ to 0. Lines 7-9 set k and $\pi[q]$ correctly in either case. This completes our proof of the correctness of COMPUTE-PREFIX-FUNCTION.

Correctness of the KMP algorithm

The procedure KMP-MATCHER can be viewed as a reimplementation of the procedure FINITE-AUTOMATON-MATCHER. Specifically, we shall prove that the code in lines 6-9 of KMP-MATCHER is equivalent to line 4 of FINITE-AUTOMATON-MATCHER, which sets q to $\delta(q, T[i])$. Instead of using a stored value of $\delta(q, T[i])$, however, this value is recomputed as necessary from p. Once we have argued that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER, the correctness of KMP-MATCHER follows from the correctness of FINITE-AUTOMATON-MATCHER (though we shall see in a moment why line 12 in KMP-MATCHER is necessary).

The correctness of KMP-MATCHER follows from the claim that we must have either $\delta(q, T[i]) = 0$ or $\delta(q, T[i]) - 1 \in \pi^*[q]$. To check this claim, let $k = \delta(q, T[i])$. Then, $P_k \sqsupset P_q T[i]$ by the definitions of δ and σ. Therefore, either $k = 0$ or else $k \geq 1$ and $P_{k-1} \sqsupset P_q$ by dropping the last character from both P_k and $P_q T[i]$ (in which case $k - 1 \in \pi^*[q]$). Therefore, either $k = 0$ or $k - 1 \in \pi^*[q]$, proving the claim.

The claim is used as follows. Let q' denote the value of q when line 6 is entered. We use the equivalence $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ from Lemma 32.5 to justify the iteration $q \leftarrow \pi[q]$ that enumerates the elements of $\{k : P_k \sqsupset P_{q'}\}$.

Lines 6-9 determine $\delta(q', T[i])$ by examining the elements of $\pi^*[q']$ in decreasing order. The code uses the claim to begin with $q = \varphi(T_{i-1}) = \sigma(T_{i-1})$ and perform the iteration $q \leftarrow \pi[q]$ until a q is found such that $q = 0$ or $P[q + 1] = T[i]$. In the former case, $\delta(q', T[i]) = 0$; in the latter case, q is the maximum element in $E_{q'}$, so that $\delta(q', T[i]) = q + 1$ by Corollary 32.7.

Line 12 is necessary in KMP-MATCHER to avoid a possible reference to $P[m + 1]$ on line 6 after an occurrence of P has been found. (The argument that $q = \sigma(T_{i-1})$ upon the next execution of line 6 remains valid by the hint given in Exercise 32.4-6: $\delta(m, a) = \delta(\pi[m], a)$ or, equivalently, $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ for any $a \in \Sigma$.) The remaining argument for the correctness of the Knuth-Morris-Pratt algorithm follows from the correctness of FINITE-AUTOMATON-MATCHER, since we now see that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER.

Exercises 32.4-1

Compute the prefix function π for the pattern ababbabbabbababbabb when the alphabet is $\Sigma = \{a, b\}$.

Exercises 32.4-2

Give an upper bound on the size of $\pi^*[q]$ as a function of q. Give an example to show that your bound is tight.

Exercises 32.4-3

Explain how to determine the occurrences of pattern P in the text T by examining the π function for the string PT (the string of length $m + n$ that is the concatenation of P and T).

Exercises 32.4-4

Show how to improve KMP-MATCHER by replacing the occurrence of π in line 7 (but not line 12) by π', where π' is defined recursively for $q = 1, 2, \ldots, m$ by the equation

$$\pi'[q] = \begin{cases} 0 & \text{if } \pi[q] = 0\,, \\ \pi'[\pi[q]] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] = P[q+1]\,, \\ \pi[q] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] \neq P[q+1]\,. \end{cases}$$

Explain why the modified algorithm is correct, and explain in what sense this modification constitutes an improvement.

Exercises 32.4-5

Give a linear-time algorithm to determine if a text T is a cyclic rotation of another string T'. For example, arc and car are cyclic rotations of each other.

Exercises 32.4-6: □

Give an efficient algorithm for computing the transition function δ for the string-matching automaton corresponding to a given pattern P. Your algorithm should run in time $O(m\,|\Sigma|)$. (*Hint:* Prove that $\delta(q, a) = \delta(\pi[q], a)$ if $q = m$ or $P[q + 1] \neq a$.)

Problems 32-1: String matching based on repetition factors

Let y^i denote the concatenation of string y with itself i times. For example, $(ab)^3 = ababab$. We say that a string $x \in \Sigma^*$ has ***repetition factor*** r if $x = y^r$ for some string $y \in \Sigma^*$ and some $r > 0$. Let $\rho(x)$ denote the largest r such that x has repetition factor r.

a. Give an efficient algorithm that takes as input a pattern $P[1 \ldots m]$ and computes the value $\rho(P_i)$ for $i = 1, 2, \ldots, m$. What is the running time of your algorithm?
b. For any pattern $P[1 \ldots m]$, let $\rho^*(P)$ be defined as $\max_{1 \le i \le m} \rho(P_i)$. Prove that if the pattern P is chosen randomly from the set of all binary strings of length m, then the expected value of $\rho^*(P)$ is $O(1)$.
c. Argue that the following string-matching algorithm correctly finds all occurrences of pattern P in a text $T[1 \ldots n]$ in time $O(\rho^*(P)n + m)$.

```
d.   REPETITION-MATCHER(P, T)
e.    1  m ← length[P]
f.    2  n ← length[T]
g.    3  k ← 1 + ρ*(P)
h.    4  q ← 0
i.    5  s ← 0
j.    6  while s ≤ n - m
k.    7       do if T[s + q + 1] = P[q + 1]
l.    8             then q ← q + 1
m.    9                  if q = m
n.   10                     then print "Pattern occurs with shift" s
o.   11          if q = m or T[s + q + 1] ≠ P[q + 1]
p.   12             then s ← s + max(1, ⌈q/k⌉)
q.   13                  q ← 0
```

This algorithm is due to Galil and Seiferas. By extending these ideas greatly, they obtain a linear-time string-matching algorithm that uses only $O(1)$ storage beyond what is required for P and T.

Chapter notes

The relation of string matching to the theory of finite automata is discussed by Aho, Hopcroft, and Ullman [5]. The Knuth-Morris-Pratt algorithm [187] was invented independently by Knuth and Pratt and by Morris; they published their work jointly. The Rabin-Karp algorithm was proposed by Rabin and Karp [175]. Galil and Seiferas [107] give an interesting deterministic linear-time string-matching algorithm that uses only $O(1)$ space beyond that required to store the pattern and text.

Chapter 33: Computational Geometry

Overview

Computational geometry is the branch of computer science that studies algorithms for solving geometric problems. In modern engineering and mathematics, computational geometry has applications in, among other fields, computer graphics, robotics, VLSI design, computer-aided design, and statistics. The input to a computational-geometry problem is typically a