BU CLA CS 530: Analysis of Algorithms

Spring 1999

Handout 2

Randomized Algorithms

Contents

A *randomized algorithm* is an algorithm that makes random choices during execution. In the analysis of randomized algorithms, we assume they have access to a source of independent unbiased random bits (for example, a random-number generator); this is a useful idealization which is, of course, only approximated in actual programming practice (a random-number generator is in fact only a "pseudorandom"-number generator).

1 Randomized Linear Search

Given an array $A[1\,..\,n]$ of $n \geqslant 1$ integers and a search element B, LINEAR-SEARCH determines whether $B = A[i]$ for some $i \in \{1, \ldots, n\}$, and if it is, LINEAR-SEARCH returns such an index i. The analysis of deterministic LINEAR-SEARCH is considered in Exercise 1.1-3, page 5, and Exercise 1.2-2, page 11, in [CLR]. We leave to you the task of writing the pseudocode for LINEAR-SEARCH.

In scanning the input array A, LINEAR-SEARCH can proceed from left to right, or from right to left, without affecting the best-case, worst-case and average-case running times. Measured by the required number of comparisons, the first two running times are $T_{\text{best}}(n) = 1$ and $T_{\text{worst}}(n) = n$. Under a *uniform-distribution* assumption on input pairs (A, B), namely, B occurs in A and each entry in A is equally likely to be B, the third running time is $T_{\text{ave}}(n) = (n+1)/2$.

Recall how the functions $T_{\text{best}}(n)$, $T_{\text{worst}}(n)$ and $T_{\text{ave}}(n)$ are obtained precisely. Let $\mathcal{I}_n$ be the set of all input pairs (A, B) where A has $n \geqslant 1$ entries. Let $t(A, B)$ be the number of comparisons carried out by LINEAR-SEARCH on input (A, B). For the same (A, B), the value of $t(A, B)$ is uniquely defined, i.e., running LINEAR-SEARCH over and over again on the *same* input (A, B) requires the *same* number of comparisons, because LINEAR-SEARCH is deterministic. The precise definitions are:

$$T_{\text{best}}(n) = \min\{\ t(A,B) \mid (A,B) \in \mathcal{I}_n\ \} \quad \text{and} \quad T_{\text{worst}}(n) = \max\{\ t(A,B) \mid (A,B) \in \mathcal{I}_n\ \}$$

for all $n \geqslant 1$. (There is no harm in setting $T_{\text{best}}(0) = T_{\text{worst}}(0) = 0$ in order to make the two functions defined on all natural numbers.)

For $T_{\text{ave}}(n)$ we need to know a probability distribution on events of the form "LINEAR-SEARCH makes i comparisons on input (A, B)". We make the uniform-distribution assumption stated above: B occurs in A and each entry in A is equally likely to be B. To make this precise, we introduce a function $X : \mathcal{I}_n \to \{1, \ldots, n\}$, what probability theorists call a *random variable*, which uniquely associates with every input pair $(A, B) \in \mathcal{I}_n$ a number $i \in \{1, \ldots, n\}$ where i is the number of comparisons carried out by LINEAR-SEARCH on input (A, B). In the notation of Chapter 6 in [CLR], in particular Section 6.3, the uniform-distribution assumption above is expressed by writing:

$$(\dagger) \quad \Pr\{X = i\} = 1/n \quad \text{for every } i \in \{1, \ldots, n\} \qquad \text{and} \qquad \Pr\{X \neq 1, \ldots, X \neq n\} = 0\ .$$

This defines what is called a *probability density function* for the random variable X. Following convention, we write $\{X = i\}$ or $\{X(A, B) = i\}$ to denote a subset of $\mathcal{I}_n$ (i.e., a so-called *event*) defined by:

$$\{X=i\} \;=\; \{X(A,B)=i\} \;=\; \{\ (A,B)\in\mathcal{I}_n \mid X(A,B)=i\ \}\ .$$

The statement "$\Pr\{X=i\}=1/n$" therefore means "the probability that LINEAR-SEARCH makes i comparisons on input (A,B) is $1/n$", for all $i\in\{1,\dots,n\}$. Hence, under the stated assumption, a precise definition for $T_{\text{ave}}(n)$ is:

$$\begin{aligned} T_{\text{ave}}(n) &= \mathrm{E}[X] = \sum_i i\cdot\Pr\{X=i\} \\ &= 1\cdot(1/n) + 2\cdot(1/n) + \cdots + n\cdot(1/n) \\ &= \frac{1+2+\cdots+n}{n} = \frac{n+1}{2} \end{aligned}$$

We now randomize LINEAR-SEARCH in order to obtain a uniform expected behavior. The result is RANDOMIZED-LINEAR-SEARCH, which starts by flipping a fair coin once at the beginning, in order to decide whether to call LINEAR-SEARCH with a left-to-right scan or a right-to-left scan. Different executions of RANDOMIZED-LINEAR-SEARCH on the same input pair (A,B) have different running times. Thus, on the same input (A,B), the number of comparisons $t(A,B)$ is no longer uniquely defined. What we now measure instead is the *expected* number of comparisons $t^{\text{exp}}(A,B)$ for the same input (A,B). For RANDOMIZED-LINEAR-SEARCH we have therefore the functions $T^{\text{exp}}_{\text{best}}(n)$, $T^{\text{exp}}_{\text{worst}}(n)$ and $T^{\text{exp}}_{\text{ave}}(n)$, denoting the expected best-case, the expected worst-case, and the expected average-case running times.[1] These are precisely defined by

$$\begin{aligned} T^{\text{exp}}_{\text{best}}(n) &= \min\{\ t^{\text{exp}}(A,B) \mid (A,B)\in\mathcal{I}_n\ \}\ , \\ T^{\text{exp}}_{\text{worst}}(n) &= \max\{\ t^{\text{exp}}(A,B) \mid (A,B)\in\mathcal{I}_n\ \}\ , \\ T^{\text{exp}}_{\text{ave}}(n) &= \mathrm{E}[Y] = \sum_i i\cdot\Pr\{Y=i\}\ . \end{aligned}$$

In the definition of $T^{\text{exp}}_{\text{ave}}(n)$, we use a different random variable Y, because $\Pr\{Y(A,B)=i\}=p$ now expresses "the probability that RANDOMIZED-LINEAR-SEARCH (not LINEAR-SEARCH) makes i comparisons on input (A,B) is p". Accordingly, the probability density function for Y does not have to be the one stated in (†) for X. The probability density function of Y will now depend on assumptions we make about input pairs $(A,B)\in\mathcal{I}_n$ and how they make RANDOMIZED-LINEAR-SEARCH behave. This analysis of RANDOMIZED-LINEAR-SEARCH is continued in Exercise 1.

Exercise 1

1. Determine $T_{\text{ave}}(n)$ for LINEAR-SEARCH under a different uniform distribution, namely, B is equally likely to be one of the following $B=A[1]$, …, $B=A[n]$, $B\notin\{A[1],\dots,A[n]\}$. These are $(n+1)$ equally probable events.

[1] "Average" and "expected" mean the same thing here. I purposely use two different words in order to avoid using the expression "average average-case running time".

2. Write the pseudocode for RANDOMIZED-LINEAR-SEARCH.

3. Suppose we execute RANDOMIZED-LINEAR-SEARCH over and over again on input pairs (A, B) satisfying the uniform-distribution assumption that each B occurs in A and each entry in A is equally likely to be B. Show that the expected running times are $T^{\text{exp}}_{\text{best}}(n) = T^{\text{exp}}_{\text{worst}}(n) = T^{\text{exp}}_{\text{ave}}(n) = (n+1)/2$. ■

There is nothing particularly unique in our analysis above. For example, we can give a slightly different analysis, which now ties more closely together the 6 running-time functions (3 for LINEAR-SEARCH and 3 for RANDOMIZED-LINEAR-SEARCH). For every $n \geq 1$, we introduce a set of $n+1$ formal elements $\{\alpha_1, \ldots, \alpha_{n+1}\}$. We use a new random variable $Z : \mathcal{I}_n \to \{\alpha_1, \ldots, \alpha_{n+1}\}$. We simplify the problem a little, by assuming that B occurs at most once in A, for every input pair $(A, B) \in \mathcal{I}_n$. We take the statement $Z(A, B) = \alpha_i$ to be shorthand for one of two assertions:

- either "B occurs as entry $A[i]$" if $1 \leq i \leq n$,
- or "B does not occur in A" if $i = n+1$.

In contrast to X, the random variable Z makes no reference to LINEAR-SEARCH. We define a probability density function for Z, which will produce the same final result as the probability density function for X in (†), by writing:

(‡) $\Pr\{Z(A, B) = \alpha_i\} = 1/n$ for every $i \in \{1, \ldots, n\}$ and $\Pr\{Z(A, B) = \alpha_{n+1}\} = 0$.

Let $u : \{\alpha_1, \ldots, \alpha_{n+1}\} \to \{1, \ldots, n\}$ be a function which counts the number of comparisons carried out by LINEAR-SEARCH on input $(A, B) \in \mathcal{I}_n$; specifically, assuming that LINEAR-SEARCH starts to scan A from the left, we define:

$$u(\alpha_i) = \begin{cases} i & \text{if } 1 \leq i \leq n, \\ n & \text{if } i = n+1. \end{cases}$$

The justification of this definition of u is easy and left to you. We can now write for the running-time functions for LINEAR-SEARCH:

$$\begin{aligned} T_{\text{best}}(n) &= \min\{\ u(\alpha) \mid (A, B) \in \mathcal{I}_n \text{ and } Z(A, B) = \alpha\ \}\ , \\ T_{\text{worst}}(n) &= \max\{\ u(\alpha) \mid (A, B) \in \mathcal{I}_n \text{ and } Z(A, B) = \alpha\ \}\ , \\ T_{\text{ave}}(n) &= \mathrm{E}[Z] = \sum_{(A,B) \in \mathcal{I}_n} u(\alpha) \cdot \Pr\{Z(A, B) = \alpha\}\ . \end{aligned}$$

Turning to RANDOMIZED-LINEAR-SEARCH, let $u^{\text{exp}} : \{\alpha_1, \ldots, \alpha_{n+1}\} \to \{1, \ldots, n\}$ be the function which counts the *expected* number of comparisons carried out on input $(A, B) \in \mathcal{I}_n$; specifically,

$$u^{\text{exp}}(\alpha_i) = \begin{cases} \frac{1}{2}u(\alpha_i) + \frac{1}{2}(n - u(\alpha_i) + 1) & \text{if } 1 \leq i \leq n, \\ n & \text{if } i = n+1. \end{cases}$$

Again, the justification of the definition of $u^{\rm exp}$ is left to you. We can now write for the running-time functions for RANDOMIZED-LINEAR-SEARCH:

$$\begin{aligned} T^{\rm exp}_{\rm best}(n) &= \min\{\ u^{\rm exp}(\alpha) \mid (A,B) \in \mathcal{I}_n \text{ and } Z(A,B) = \alpha\ \}\ , \\ T^{\rm exp}_{\rm worst}(n) &= \max\{\ u^{\rm exp}(\alpha) \mid (A,B) \in \mathcal{I}_n \text{ and } Z(A,B) = \alpha\ \}\ , \\ T^{\rm exp}_{\rm ave}(n) &= \mathrm{E}[Z] = \sum_{(A,B)\in\mathcal{I}_n} u^{\rm exp}(\alpha) \cdot \Pr\{Z(A,B) = \alpha\}\ . \end{aligned}$$

In this version of the analysis, the 3 running-time functions for RANDOMIZED-LINEAR-SEARCH are obtained from those of LINEAR-SEARCH simply by replacing "$u(\alpha)$" by "$u^{\rm exp}(\alpha)$".

2 Randomized Quicksort

Another example of a randomized algorithm is RANDOMIZED-QUICKSORT. A presentation is given in Section 8.3 of [CLR]. The expected number of comparisons used by RANDOMIZED-QUICKSORT on an input array with $n \geqslant 1$ entries is given by the recurrence:

$$(1) \quad T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n)$$

This is equation (8.4), page 165, in [CLR]. One way of solving this recurrence is given on pages 166-167, based on the "substitution" method (Section 4.1 in [CLR]). Another simpler approach is given here, which does not fit (at least directly) into one of the three methods proposed in Chapter 4 ("substitution", "iteration", and "master").

The $\Theta(n)$ part in equation (1) can be replaced by $c \cdot (n+1)$, for some small constant c. This depends on the particular implementation of RANDOMIZED-QUICKSORT given in [CLR]. For our approach here, we need to replace $\Theta(n)$ by a specific function. We therefore solve

$$(2) \quad T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + c \cdot (n+1)$$

for all $n \geqslant 1$, with the boundary condition $T(1) = 1$. Multiplying both side of equation (2) by n, and subtracting the same equation with n replaced by $n-1$, we obtain:

$$(3) \quad n \cdot T(n) - (n-1) \cdot T(n-1) = 2 \cdot T(n-1) + c \cdot n \cdot (n+1) - c \cdot (n-1) \cdot n$$

for all $n \geqslant 2$, from which we obtain

$$n \cdot T(n) = (n+1) \cdot T(n-1) + 2cn$$

Dividing both sides by $n \cdot (n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

Now we can iterate, to obtain:

$$
\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2c}{n+1} \\
&= \frac{T(n-2)}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\
&= \frac{T(n-3)}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\
&\vdots \\
&= \frac{T(1)}{2} + \frac{2c}{3} + \frac{2c}{4} + \cdots + \frac{2c}{n} + \frac{2c}{n+1} \\
&= \frac{1}{2} + 2c \sum_{k=3}^{n+1} \frac{1}{k}
\end{aligned}
$$

From equation (3.5), page 44, in [CLR], we have $\sum_{k=1}^{n}(1/k) = \ln n + \Theta(1)$. (In fact, the $\Theta(1)$ part here is a small constant $< .57721\cdots$.) Hence, $\sum_{k=3}^{n+1}(1/k) = \ln (n+1) + \Theta(1)$, so that finally:

$$T(n) = 2c \cdot (n+1) \cdot \ln (n+1) + \Theta(n) = \Theta(n \log n)$$

3 Randomized Selection

This is algorithm RANDOMIZED-SELECTION in Section 10.2 in [CLR].

On page 188 in [CLR], the first sentence of the second paragraph is "the worst-case running time for RANDOMIZED-SELECTION is $\Theta(n^2)$, even to find the minimum ...". The "worst-case running time" for a randomized algorithm is different from what I call in this handout the "*expected* worst-case running time", which is a meaningful measure for a randomized algorithm but not for a deterministic one. The *expected* worst-case running time is measured by running the randomized algorithm repeatedly on the same input, rather than by running it exactly once. For RANDOMIZED-SELECTION, the expected best-case, the expected worst-case, and the expected average-case running times are all the same, i.e., $\Theta(n)$.

4 Finding an Element $\geq$ the Median

The following is a common problem in the design of combinatorial algorithms: Given an array $A[1\,..\,n]$ of integers, for some $n \geqslant 1$, find an entry $A[i]$ which is larger than or equal to the median of all the entries $\{A[1], \ldots, A[n]\}$.

One approach is to find the maximum value in $A[1\,..\,n]$, which is certainly $\geqslant$ the median. This requires $n-1$ comparisons.

Another option is to start the maximum-finding algorithm and to stop after $\lceil n/2 \rceil$ entries have been examined. An integer which is greater than or equal to one-half of the integers is certainly $\geqslant$ the median. This requires no more than $n/2$ comparisons.

But we can do better, if we accept a very, very small probability of returning a wrong answer. UPPER-HALF is a randomized algorithm for this problem, which makes no more than 100 comparisons (regardless of how large n is!).

UPPER-HALF(A)
1. $n \leftarrow \mathit{length}(A)$
2. **if** $n < 200$
3. **then return** MAXIMUM$(A[1\,..\,\lceil n/2 \rceil])$
4. **else for** $i \leftarrow 1$ **to** 100 **do** $B[i] \leftarrow$ A[RANDOM$(1,n)$]
5. **return** MAXIMUM$(B[1\,..\,100])$

The array B in line 4 is of dimension 100, and 100 randomly selected integers from A are entered into B. The probability that all of these 100 integers are less than the median is $< (1/2)^{100}$, because the probability of each of these to be greater than or equal to the median is at least 1/2 (it is more than 1/2 if several entries are equal to the median). Hence, the error probability of UPPER-HALF is less than 2^{-100} which, for all practical purposes, is 0. (The probability of a programming error or a computer hardware error — or an earthquake in Boston as you read this handout — far exceeds 2^{-100}.)

Exercise 2 Design an efficient algorithm UPPER-FIFTH which, given an array $A[1\,..\,n]$ of integers, makes a constant number of comparisons to find an integer larger than 80% of all the entries in the array — with probability error less than 2^{-100}. ■

5 Searching an Array with Repeated Entries

A variation of the search problem is to start with an array $A[1\,..\,n]$ of $n \geqslant 1$ integers where every entry occurs at least $k \geqslant 1$ times. RANDOM-SEARCH is a randomized algorithm for this problem. On the face of it, RANDOM-SEARCH is based on a totally wrongheaded reasoning: With some bad luck, RANDOM-SEARCH will run for a very, very long time.

RANDOM-SEARCH(A, B)
1. $n \leftarrow \mathit{length}(A)$
2. **repeat** $x \leftarrow$ A[RANDOM$(1,n)$]
3. **until** $B = A[x]$
4. **return** x

Measuring the running time by the number of comparisons carried out (in the test of line 3 in the algorithm), the expected best-case complexity is $T^{\text{exp}}_{\text{best}}(n) = 1$, which certainly happens when $A[1] = \cdots = A[n] = B$.

The expected worst-case complexity is achieved when the search element B occurs exactly k times. Minimizing the number of times B occurs in A maximizes the work done by the algorithm;

we do this because we are now interested in the worst-case behavior of the algorithm. Under this assumption, every iteration of the **repeat** loop finds B with probability k/n and does not find B with probability $(n-k)/n$. Hence, the probability p_i that the loop is iterated i times is

$$p_i \;=\; \left(\frac{n-k}{n}\right)^{i-1} \cdot \frac{k}{n} \;=\; \frac{k \cdot (n-k)^{i-1}}{n^i}\,.$$

This implies the following sequence of equalities

$$\begin{aligned}
T^{\text{exp}}_{\text{worst}}(n) &= \sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i \cdot \frac{k \cdot (n-k)^{i-1}}{n^i} \\
&= \frac{k}{n-k} \sum_{i=0}^{\infty} i \cdot \frac{k \cdot (n-k)^{i-1}}{n^i} \\
&= \frac{k}{n-k} \cdot \frac{n-k}{n} \cdot \frac{1}{(1-((n-k)/n))^2} \\
&= \frac{n}{k}
\end{aligned}$$

where the last but one equality follows from formula (3.6), page 44, in [CLR]. If k is a fixed percentage of n, say k is 20% of n regardless of how large or small n is, then n/k is a constant independent of n and $T^{\text{exp}}_{\text{worst}}(n) = \mathcal{O}(1)$. The corresponding worst-case complexity for LINEAR-SEARCH is $T_{\text{worst}}(n) = n - k + 1$ (why?).

6 Las Vegas vs. Monte Carlo

Five different randomized algorithms were presented in the preceding sections. We now point out some common features and some contrasting features of these algorithms.

RANDOMIZED-LINEAR-SEARCH (Section 1) and RANDOMIZED-QUICKSORT (Section 2) are the result of randomizing the deterministic algorithms LINEAR-SEARCH and QUICKSORT. What is achieved by randomizing these two algorithms is a uniform expected behavior, by equalizing the expected best-case, expected worst-case and expected average-case running times. This is a situation encountered in the randomization of many deterministic algorithms: If algorithm **B** is obtained by randomizing deterministic algorithm **A**, the expected average complexity of **B** is no better than the average complexity of **A**. So, there is no improvement in the average complexity. The gain in randomizing **A** to obtain **B** is elsewhere: If the average-case complexity of **A** is significantly smaller than its worst-case complexity, then the *expected worst-case complexity* of **B** will usually be significantly smaller than the *worst-case complexity* of **A**.

RANDOMIZED-SELECTION (Section 3) has different features as a randomized algorithm. It is not derived from a deterministic algorithm for the same problem. Nevertheless, just as the two preceding randomized algorithms, RANDOMIZED-SELECTION exhibits a uniform behavior by equalizing the expected best-case, expected worst-case and expected average-case running times.

UPPER-HALF (Section 4) is an altogether different kind of randomized algorithm. For one thing, its internal operation bears little resemblance to a deterministic algorithm for the same problem, and this is a feature that RANDOMIZED-SELECTION and UPPER-HALF have in common. However, in contrast to the three preceding algorithms, UPPER-HALF may return an incorrect answer. A randomized algorithm that may sometimes produce an incorrect answer is often called a *Monte Carlo algorithm*. We accept the possibility of an incorrect answer because we are able to bound its probability (a very small one) and because we can greatly reduce the use of computing resources (by comparison to an algorithm that always produces a correct answer). UPPER-HALF is a Monte Carlo algorithm.

RANDOM-SEARCH (Section 5) is another kind yet of a randomized algorithm, which always gives the correct answer, if and when it stops, but which may occasionally run for an arbitrarily long time. Such a randomized algorithm is often called a *Las Vegas algorithm*. The only variation from one execution to another is the use of computing resources (running time, memory space). In this sense, RANDOMIZED-LINEAR-SEARCH, RANDOMIZED-QUICKSORT and RANDOMIZED-SELECTION all qualify also as Las Vegas algorithms, because they always return the correct answer. However, many researchers in the field tend to limit the name "Las Vegas" to algorithms that will occasionally run for arbitrarily long and, therefore, exclude RANDOMIZED-LINEAR-SEARCH, RANDOMIZED-QUICKSORT and RANDOMIZED-SELECTION from this group of algorithms. None of these three algorithms is subject to this extreme behavior of running for arbitrarily long without producing an answer.

This classification of randomized algorithms, into Las Vegas and Monte Carlo, is a useful one (and the two names are easy to remember). The two kinds are in fact closely related. First, we can take a Las Vegas algorithm as a special case of a Monte Carlo algorithm, simply by viewing it as a Monte Carlo algorithm with error probability 0. Second, from a Monte Carlo algorithm, we can also derive a Las Vegas algorithm for the same computational problem, whose efficiency depends on the time required to verify the correctness of a solution to the problem.[2]

Exercise 3 Let $0 < \varepsilon_2 < \varepsilon_1 < 1$. Let **A** be Monte Carlo algorithm for a decision problem (*yes-no* output) which gives a correct answer with probability $\geqslant 1 - \varepsilon_1$, regardless of the input. How many independent executions of **A** suffice to raise the probability of obtaining a correct answer to $1 - \varepsilon_2$, regardless of the input? *Hint*: If **A** is run repeatedly with independent random choices every time, the error probability can be made arbitrary small, but at the expense of running time. Start with specific values for ε_1 and ε_2, say $\varepsilon_1 = 1/4$ and $\varepsilon_2 = 1/6$, then generalize to arbitrary ε_1 and ε_2. ∎

Exercise 4 Let **A** be a Monte Carlo algorithm for a computational problem $\mathcal{P}$, whose expected running time is $\leqslant T(n)$ on every input instance of size $n \geqslant 1$, and which produces a correct answer with probability $\gamma(n)$. Assume that given a solution for an instance of $\mathcal{P}$, we can verify its

[2] You may wonder what the difference is between "finding a correct answer" and "verifying that an answer is correct". For some computational problems, the time to verify an answer can be considerably less than to find a correct one. See for example the discussion on "polynomial-time verification", Section 36.2, in [CLR].

correctness in time $\leqslant t(n)$. Design a Las Vegas algorithm for $\mathcal{P}$ (which, by definition, always gives a correct answer) whose expected running time is $\leqslant (T(n) + t(n))/\gamma(n)$. ■

7 Verifying a Polynomial Identity

Another example of a randomized algorithm checks whether a rational polynomial $f(x_1, \ldots, x_n)$, with $n \geqslant 1$ variables, is identically 0. Such a polynomial is identically 0 iff, after writing f as a sum of monomial terms,[3] all the terms cancel each other out. This elementary fact can be used to write a simple algorithm for checking whether f is identically 0, but whose running time can be (unfortunately) exponential in the size of the input expression. For example, if f is of the form $(x_1 + y_1)(x_2 + y_2) \cdots (x_n + y_n)$, which mentions $2 \cdot n$ variable occurrences, then multiplying out the n factors we obtain 2^n monomial terms:

$$x_1 x_2 \cdots x_n + x_1 x_2 \cdots x_{n-1} y_n + \ \cdots + \ y_1 y_2 \cdots y_n$$

and a total of $n \cdot 2^n$ variable occurrences. A faster algorithm is POLY-IDENTITY, which is based on the following lemma.

Lemma 5 (Schwartz-Zippel) *Let $f(x_1, \ldots, x_n)$ be a polynomial in $n \geqslant 1$ variables with degree at most $k \geqslant 1$ in each variable. If f is not identically 0 and $a_1, \ldots, a_n$ are drawn from a finite set S of integers, independently of each other according to the uniform distribution, then*

$$\Pr\{f(a_1, \ldots, a_n) = 0\} \leqslant \frac{k \cdot n}{|S|}$$

Using the notions of Sections 6.2 and 6.3 in [CLR], you can view f as a *random variable*, mapping n-tuples $(a_1, \ldots, a_n)$ from the *sample space* $S^n = S \times \cdots \times S$ (n times) to the set of rational numbers. The *event* $\{f = 0\}$ (also written $\{f(a_1, \ldots, a_n) = 0\}$) therefore refers to the subset of S^n of all n-tuples $(a_1, \ldots, a_n)$ such that $f(a_1, \ldots, a_n) = 0$.

Proof: By induction on $n \geqslant 1$. The base case $n = 1$ is straightforward, because a polynomial in one variable x_1 of degree k can have at most k roots.

For the induction hypothesis, we assume the result true for every polynomial with up to $n - 1$ variables. Consider now an arbitrary polynomial $f(x_1, \ldots, x_n)$ in n variables. We can factor out the variable x_1 to obtain:

(4) $$f(x_1, \ldots, x_n) = f_0(x_2, \ldots, x_n) + x_1 f_1(x_2, \ldots, x_n) + x_1^2 f_2(x_2, \ldots, x_n) + \cdots + x_1^p f_p(x_2, \ldots, x_n)$$

for some polynomials $f_i(x_2, \ldots, x_n)$ in $n - 1$ variables, for every $i \in \{1, \ldots, p\}$, where p is the largest exponent of x_1 in f. We assume x_1 has an effect on the value of f, so that $p > 0$ and $f_p(x_2, \ldots, x_n)$ is not identically 0.

[3] Writing f as a sum of monomial terms corresponds to putting f in what is called in [CLR], Section 32.1, "coefficient form" or "coefficient representation form".

By the induction hypothesis, the probability that $f_p(a_2, \ldots, a_n) = 0$ is $\leqslant k \cdot (n-1)/|S|$, i.e.

$$(5) \quad \Pr\{f_p(a_2, \ldots, a_n) = 0\} \leqslant \frac{k \cdot (n-1)}{|S|}$$

In case $f_p(a_2, \ldots, a_n) \neq 0$, consider the following polynomial in the single variable x_1:

$$g(x_1) = f(x_1, a_2, \ldots, a_n) = f_0(a_2, \ldots, a_n) + x_1 f_1(a_2, \ldots, a_n) + \cdots + x_1^p f_p(a_2, \ldots, a_n)$$

The polynomial g has degree $p \leqslant k$, and is not identically 0 because the coefficient of x_1^p is $f_p(a_2, \ldots, a_n) \neq 0$. Hence, by the base case of the induction,

$$(6) \quad \Pr\{f(a_1, \ldots, a_n) = 0 \mid f_p(a_2, \ldots, a_n) \neq 0\} \leqslant \frac{p}{|S|} \leqslant \frac{k}{|S|}$$

Using the axioms of probability, page 105 in [CLR], for all events A and B we have the following inequality:

$$(7) \quad \Pr\{A\} \leqslant \Pr\{B\} + \Pr\{A|\overline{B}\}$$

Hence, using (7), we can combine (5) and (6) to obtain

$$(8) \quad \Pr\{f(a_1, \ldots, a_n) = 0\} \leqslant \frac{k \cdot (n-1)}{|S|} + \frac{k}{|S|} = \frac{k \cdot n}{|S|}$$

which concludes the induction. ∎

Exercise 6

1. Supply the missing details in the preceding proof. In particular show how to obtain equation (4) and inequality (7).

2. Show that $\Pr\{A\} \leqslant \Pr\{A|B\} + \Pr\{A|\overline{B}\}$, and give a simple counterexample to show that, in general, it is not the case that $\Pr\{A\} = \Pr\{A|B\} + \Pr\{A|\overline{B}\}$. ∎

Poly-Identity is a Monte Carlo algorithm. Its input is an arbitrary rational polynomial f.

```
POLY-IDENTITY(f)
    n ← number of variables in f
    k ← largest variable-degree in f
    ℓ ← 0
    repeat
        for i ← 1 to n do a[i] ← RANDOM(1, 2kn)
        b ← f(a[1], ..., a[n])
        ℓ ← ℓ + 1
    until ℓ = 100 or b ≠ 0
```

```
9.    if b = 0
10.   then return "f is identically 0"
11.   else  return "f is not identically 0"
```

If f is identically 0, then b always evaluates to 0 in each of the 100 iterations of the **repeat-until** loop (lines 4-8). In this case, when POLY-IDENTITY stops, it returns the correct answer "f is identically 0". If f is not identically 0 and b evaluates to a value $\neq 0$, then when POLY-IDENTITY stops, it again returns the correct answer "f is not identically 0". However, if f is not identically 0 but b still evaluates to 0 in each of the 100 iterations of the **repeat-until** loop, then when POLY-IDENTITY stops, it returns the wrong answer "f is identically 0". But this last case, in which the wrong answer is returned, occurs with a probability $\leqslant (1/2)^{100} = 2^{-100}$, because by Lemma 5, the probability that $b = 0$ in each iteration of the **repeat-until** loop is $\leqslant kn/2kn = 1/2$. For all practical purposes, the probability 2^{-100} is negligible. We have thus shown that POLY-IDENTITY(f) returns the correct answer with error probability at most ε, where:

$$\varepsilon = \begin{cases} 0 & \text{if } f \text{ is identically } 0, \\ 2^{-100} & \text{if } f \text{ is not identically } 0. \end{cases}$$

We can now state the main result of this section.

Theorem 7 POLY-IDENTITY *is a linear-time Monte Carlo algorithm to check whether a rational polynomial f is identically* 0. POLY-IDENTITY(f) *returns the correct answer with probability* $\geqslant 1 - 2^{-100}$.

Exercise 8 Let $f(x)$ be a rational polynomial of degree $k \geqslant 1$ in a single variable x. We would like to determine whether f is identically 0. A *deterministic* (not randomized) algorithm is based on the following idea: First, evaluate $f(1), f(2), \ldots, f(k+1)$; second, if $f(1) = f(2) = \ldots = f(k+1)$ then "f is identically 0" else "f is not identically 0".

1. Why is this algorithm correct?

2. What would be a generalization of this idea to $n \geqslant 2$, and why can it not be used to implement a deterministic algorithm to decide whether an arbitrary rational polynomial in n variables is identically 0?

3. Discuss the merits of the deterministic algorithm in this exercise relative to algorithm POLY-IDENTITY, keeping in mind issues such as the model of computation and the sizes of the integers being operated upon. *Hint*: If k is "very large", for example $k \geqslant 10^5$, the algorithm in this exercise may be highly inefficient. ∎

Exercise 9 Let f and g be two rational polynomials in $n \geqslant 1$ variables, with degree at most $k \geqslant 1$ in each variable. If f and g are explicitly given, each as a sum of monomial terms, then we can determine whether f and g are identically equal by comparing corresponding coefficients in f and g. This can be done in time $\mathcal{O}(\max\{|f|,|g|\})$ if the monomials in f and g are given in the same order, or in time $\mathcal{O}(\max\{|f|,|g|\}\log\max\{|f|,|g|\})$ if the monomials in f and g have to be sorted in the same order first.

What if f and g are not explicitly given as sums of monomials? For example, let $f(x,y) = (5x^3(4x^3+3y^2-x-1))^2$ and $g(x,y) = (5x^3+3y^2-1)^2+(6x^2-3y^3+x-1)^3$; in this case, we have to put f and g in coefficient form first, which may in general lead to an exponential growth in the expressions for f and g (see the example in the opening paragraph of Section 7). More generally, there are situations where f and g are provided implicitly, each by a "black box" to compute it and with no direct access to its coefficients. Are f and g identically equal?

Design an efficient Monte Carlo algorithm to decide whether two such rational polynomials, f and g, are identically equal. ∎

Exercise 10 The *polynomial product verification* problem is this: Given rational polynomials $f(x)$, $g(x)$, and $h(x)$ in a single variable x, verify that $f(x)\times g(x) = h(x)$. Assume that each of f, g and h is explicitly given as a sum of monomial terms. If f and g are of degree at most $k \geqslant 1$, then h cannot have degree exceeding $2k$. We can evaluate $f\times g$ and put the resulting polynomial in coefficient form (i.e. a sum of monomials) in time $\mathcal{O}(k\log k)$, according to Theorem 32.2 in [CLR], and then compare with the corresponding coefficients in h in time $\mathcal{O}(k)$ — for a total time of $\mathcal{O}(k\log k)$. But we can do better.

Using the fact that the evaluation of a polynomial of degree k in a single variable x, at a fixed value $x=a$, can be carried out in time $\mathcal{O}(k)$, design an efficient Monte Carlo algorithm that runs in time $\mathcal{O}(k)$ for the polynomial product verification problem.

Exercise 11 Consider the problem of deciding whether two integer multisets S_1 and S_2 are identical, in the sense that each integer occurs with the same multiplicity in both sets. This problem can be solved by sorting the two sets in time $\mathcal{O}(n\log n)$, where n is the cardinality of the multisets. Suggest a way of solving this problem by reducing it to a problem involving a verification of a polynomial identity, thereby obtaining an efficient randomized algorithm. Briefly discuss the relative merits of the two methods. ∎

8 Verifying Matrix Multiplication

Given three $n\times n$ integer matrices A, B and C, we want to check whether $AB = C$. The straightforward approach is to multiply A and B and compare the result with C. Standard matrix multiplication can be carried out in time $\Theta(n^3)$, on the assumption that the dominant operation in this procedure is scalar multiplication. If the single comparison of two integers requires no more time than a single scalar multiplication, then the total running time to test $AB = C$ is still $\Theta(n^3)$. This approach can be made faster by using some of the algorithms for matrix multiplication discussed in Chapter 31 of [CLR], in particular Strassen's algorithm.

We now design a Monte Carlo algorithm for this problem which runs faster than any available deterministic algorithm for the same problem. This is algorithm TEST_2 below, which is based on the following simple lemma.

Lemma 12 *Let A, B and C be integer matrices of dimension $n \times n$, with $n \geqslant 1$. Then $AB = C$ if and only if $XAB = XC$ for every array $X = [a_1, \ldots, a_n]$ where $a_1, \ldots, a_n \in \{0, 1\}$.*

Proof: The left-to-right implication is immediate. For the right-to-left implication, define the array X_i with n entries, where $1 \leqslant i \leqslant n$, by:

$$X_i = [0, \ldots, 0, \underset{i}{1}, 0, \ldots, 0] \ ,$$

i.e., all entries in X_i except for the ith is 0. The ith row of AB equals the ith row of C if and only if $X_i AB = X_i C$. Hence, $AB = C$ if and only if

$$X_1 AB = X_1 C, \quad X_2 AB = X_2 C, \quad \ldots \quad , \quad X_n AB = X_n C \ .$$

This implies the desired conclusion. ∎

Before writing the pseudocode for TEST_2, we design a preliminary algorithm TEST_1 which will be repeatedly called by TEST_2:

```
TEST_1(A, B, C, n)
1.  for i ← 1 to n do  X[i] ← RANDOM(0, 1)
2.  if (XA)B = XC then  return "true"
3.                else  return "false"
```

If $AB = C$ then TEST_1 always returns the correct answer "true". On the other hand, if $AB \neq C$ then TEST_1 may return the wrong answer "true". What is the error probability in this case? If $AB \neq C$, there is at least one row, say i, which is not the same in both AB and C. Hence, if before the test in line 2 of TEST_1 is executed it happens that $X = [x_1, \ldots, x_i, \ldots, x_n]$ is a binary array with $x_i = 1$ (the values of the other entries in X do not matter here), then $XAB \neq XC$ and the correct answer "false" is returned. But the probability of assigning 1 to x_i in the **for**-loop of TEST_1 is exactly 1/2. Hence, when $AB \neq C$, the probability that TEST_1 returns the correct answer "false" is at least 1/2 (why not "exactly 1/2"?).

The error probability of TEST_1 is too large and clearly intolerable. But it is easy to decrease it, by calling TEST_1 on the same input several times, and this is what TEST_2 does.

We have included a fourth input parameter k in TEST_2 in order to adjust the error probability to any desirable level:

```
TEST_2(A, B, C, n, k)
    for i ← 1 to k do
        if TEST_1(A, B, C, n) = "false" then return "false"
    return "true"
```

If $AB = C$ then TEST_2 always returns the correct answer "true". And if $AB \neq C$ then TEST_2 returns the correct answer "false" with probability at least $1 - (1/2)^k$.

Exercise 13 To analyze the running time of TEST_2, we count comparisons and scalar multiplications, assuming that each of these two operations requires one time unit.

1. What is the time complexity of TEST_2? Justify briefly. Express it in the form "$\Theta(f(n))$" for some appropriate $f(n)$.

2. The test in line 2 of TEST_1 is "$(XA)B = XC$". What is the time complexity of TEST_2 if this test is changed to "$X(AB) = XC$"? Justify briefly. ∎

Exercise 14 Design an efficient Monte Carlo algorithm to decide, given two $n \times n$ integer matrices A and B, whether or not B is the inverse of A. Determine the precise runnning time of your algorithm (counting comparisons and scalar multiplications) in terms of n and the acceptable error probability ε. ∎

9 Finding a Min-Cut of a Multigraph

Let G be a connected undirected multigraph with $n \geqslant 1$ vertices. A *multigraph* may contain multiple edges between any two vertices. Let us assume that G contains no self-loops.[4]

Following Section 24.1 in [CLR], a *cut* $(S, V - S)$ of an undirected multigraph $G = (V, E)$ is a partition of V such that $S \neq \varnothing \neq V - S$.[5] An edge $(u, v) \in E$ is said to *cross* the cut $(S, V - S)$ if $u \in S$ and $v \in V - S$ (or vice-versa). Thus, removing all the edges crossing the cut $(S, V - S)$ results in a disconnected multigraph. Let $C(S, V - S)$ denote the set of edges crossing the cut $(S, V - S)$.

The *size* of the cut $(S, V - S)$ is the size of $C(S, V - S)$, i.e., the number of edges crossing the cut. A multiple edge contributes its multiplicity to the size of the cut. A *min-cut* is a cut of minimum size. We wish to design an efficient algorithm to determine a min-cut in a connected undirected multigraph. A randomized algorithm for this problem repeats the following steps as long as there are two vertices in the graph:

1. Pick an edge uniformly at random and merge the two vertices at its end-points.

2. If as a result there are several edges between some pairs of vertices, retain all the edges.

3. Edges between vertices that are merged are removed, so that there are never any self-loops.

We call a *contraction* the combination of the preceding 3 steps. With each contraction, the number of vertices decreases by one. When two vertices v_1 and v_2 remain, the algorithm stops and output the set $C(\{v_1\}, \{v_2\})$ of edges between v_1 and v_2.

[4]The book [CLR], bottom of page 89, allows a multigraph to have self-loops.
[5]The requirement "$S \neq \varnothing \neq V - S$" is omitted in the book [CLR].

The pseudo-code for the algorithm is given next, where we write the $G/(u_1, u_2)$ to refer to the multigraph obtained from G by contracting the edge (u_1, u_2).

```
CONTRACT(G)
    H ← G
    while H has more than 2 vertices do
        choose an edge (u1, u2) uniformly at random from the edges of H
        H ← H/(u1, u2)
    return C({v1}, {v2}) where v1 and v2 are the 2 vertices left in H
```

Why should CONTRACT be a good algorithm for our problem? First, observe that the set of edges $C(\{v_1\}, \{v_2\})$ returned by CONTRACT is indeed a cut of G, because every cut of H at any intermediate stage is also a cut of the original multigraph G. What is in doubt is whether the output $C(\{v_1\}, \{v_2\})$ is in fact a min-cut.

Lemma 15 *The probability that the cut $C(\{v_1\}, \{v_2\})$ is a min-cut exceeds $2/n^2$.*

Proof: Suppose $C = C(S, V - S)$ is a min-cut of G and $|C| = k \geqslant 1$. It follows that every vertex v of G has degree $\geqslant k$; otherwise, if the degree of v were less than k, the size of the cut $(\{v\}, V - \{v\})$ would also be less than k. Hence, the number of edges $m \geqslant n \cdot k/2$, because if the n vertices of G have degrees $d_1, \ldots, d_n$ respectively, then $2m = d_1 + \cdots + d_n$.

For $i \in \{1, \ldots, n-2\}$, let p_i be the probability that the edge randomly chosen in the ith iteration is *not* in C. Because $|C| = k$ and $m \geqslant n \cdot k/2$, it must be that

$$p_1 = 1 - (k/m) \geqslant 1 - k/(nk/2) = 1 - 2/n = \frac{n-2}{n} \ .$$

Assuming that *no* edge in C is chosen in the first iteration, there are at least $(n-1) \cdot k/2$ edges at the beginning of the second iteration, so that

$$p_2 \geqslant 1 - k/((n-1)k/2) = 1 - 2/(n-1) = \frac{n-3}{n-1} \ .$$

Similarly, assuming *no* edge in C is chosen in each of the first i iterations, where $0 \leqslant i \leqslant n-3$, there are at least $(n-i) \cdot k/2$ edges at the beginning of the $(i+1)$st iteration, so that

$$p_{i+1} \geqslant 1 - k/((n-i)k/2) = 1 - 2/(n-i) = \frac{n-i-2}{n-i} \ .$$

Hence, the probability that *no* edge in C is ever chosen is

$$p_1 \cdot p_2 \cdot \ \cdots \ \cdot p_{n-2} \geqslant \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) = \frac{2}{n(n-1)} > \frac{2}{n^2}$$

which is the desired conclusion. ■

Thus, CONTRACT is a Monte Carlo that may err in declaring that $C(\{v_1\}, \{v_2\})$ is a min-cut, with error probability $\leqslant (1 - (2/n^2))$. This error probability is relatively high, especially for larger values of n. A more tolerable error probability is obtained by running CONTRACT on the same input G several times, say, $n^2/2$ times, making independent random choices every time. The probability that a min-cut is *not* found in any of the $n^2/2$ attempts is at most

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} \leqslant 1/e \approx 0.3678 \;,$$

where we use equation (2.7) in [CLR], page 34: For all real numbers x, we have $e^{-x} \geqslant 1 - x$, so that we also have $(1/e)^x \geqslant 1 - x$ and $(1/e) \geqslant (1 - x)^{1/x}$. Further executions of CONTRACT will make the error probability arbitrarily small, at the cost of increasing the running time.

Exercise 16 Let $G = (V, E)$ be a connected undirected multigraph where $|V| = n \geqslant 1$ and $|E| = m \geqslant n - 1$.

1. Devise a data structure that processes any arbitrary sequence of edge contractions in G, such that at any given point where the set of edges already contracted is F, the graph G/F is specified in the form of an adjacency matrix.[6] Furthermore, it should be possible to efficiently determine for any edge in G/F the corresponding edge in G. Your data structure should require $\mathcal{O}(n)$ time per contraction and use an amount of space which is polynomial in n.
   Hint: We want the time bound to be independent of the number m of edges. For this, the multigraph needs to be specified as a graph with integer edge weights representing the multiplicities of the edges. You may assume that the number of edges in the multigraph is a polynomial in n, although this is not strictly necessary.

2. Show that an edge can be selected uniformly at random from E in time $\mathcal{O}(n)$, given also access to a source of random bits.
   Hint: Use the data structure of part 1.

3. Determine an asymptotic upper bound on the running time of algorithm CONTRACT based on the results of parts 1 and 2. ■

10 Primality Testing

11 String Matching

As written on page 860, in [CLR], RABIN-KARP-MATCHER(T, P, d, q) is not randomized: It nowhere makes use of a random-number generator. The modulus q is given as part of the input. For

[6] The notation G/F generalizes the notation $G/(u_1, u_2)$ used earlier. F is a non-empty set of edges in G. It can be easily shown, and you can assume, that the effect of contracting the edges in F is independent of the order of contraction. Each meta-vertex in G/F corresponds to a connected set of vertices in the original G, and the edges in G/F are exactly the edges in G whose end-points do not get collapsed into the same meta-vertex in G/F.

a randomized version, q is selected by the algorithm itself, uniformly at random from the set of primes smaller than an appropriately determined threshold. (The determination of such a threshold can be based on the analysis on pages 860-861. But there are other ways of choosing an appropriate threshold. We will not worry about this issue further here.) Call the adaptation of RABIN-KARP-MATCHER that chooses its own modulus RABIN-KARP-MATCHER$'(T, P, d)$. We can call RABIN-KARP-MATCHER$'$ a Las Vegas algorithm, because it never returns a wrong answer. And this is because it performs a double-check (line 11 of the algorithm on page 860) before declaring that a match was found.

We can also have a Monte Carlo version, call it RABIN-KARP-MATCHER$''$, which is obtained from RABIN-KARP-MATCHER$'$ simply by omitting the double-check in line 11: After it finds that the "fingerprints" p and t_s are equal (line 10), it directly returns "Pattern occurs with shift s" (line 12). Without the double-check, RABIN-KARP-MATCHER$''$ runs faster than RABIN-KARP-MATCHER$'$, but the price is the possibility of an occasional error. Equality of "fingerprints" does not guarantee that a match is found.

The analysis on page 861 shows that *with* the double-check (to rule out spurious hits on line 11), the algorithm runs in time $\Theta((n-m+1)m)$ in the worst-case, and *without* the double-check, the algorithm runs in time at most $\mathcal{O}(n+m)$ in the worst-case. Hence, we can say that RABIN-KARP-MATCHER$'$ is guaranteed not to err but it may run for longer than linear-time (with low probability), whereas RABIN-KARP-MATCHER$''$ is guaranteed to run in linear-time but it may err (with low probability).

Material in this handout is partly drawn from:

- Chapters 8 and 34 in [CLR].
- Chapter 6 in lecture notes by Lovász and Gacs.
- Chapters 1, 7 and 10 in [MR].
- Chapter 10 in [BB].

Assaf Kfoury
Created: 96.01.18
Modified: 99.02.22