**BU CLA CS 530: Analysis of Algorithms**

**Spring 1996**

**Handout 2**

# Randomized Algorithms

---

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 Quicksort

Our first 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:

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

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

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

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:

$$n \cdot T(n) - (n-1) \cdot T(n-1) = 2 \cdot T(n-1) + c \cdot n \cdot (n+1) - c \cdot (n-1) \cdot n \tag{3}$$

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:

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

## 2 Order Statistics

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)
    n ← length(A)
    if n < 200
    then return MAXIMUM(A[1 .. ⌈n/2⌉])
    else  for i ← 1 to 100 do B[i] ← A[RANDOM(1, n)]
          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 1** 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 $2^{-100}$. ∎

## 3 Las Vegas vs. Monte Carlo

A randomized algorithm that always gives the correct answer 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). RANDOMIZED-QUICKSORT is a Las Vegas algorithm.

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.

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.[1]

**Exercise 2** 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 $\varepsilon_1$ and $\varepsilon_2$, say $\varepsilon_1 = 1/4$ and $\varepsilon_2 = 1/6$, then generalize to arbitrary $\varepsilon_1$ and $\varepsilon_2$. ∎

**Exercise 3** 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

[1] 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].

answer with probability $\gamma(n)$. Assume that given a solution for an instance of $\mathcal{P}$, we can verify its 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 $\leqslant (T(n) + t(n))/\gamma(n)$. ∎

## 4 Verifying a Polynomial Identity

Our third 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,[2] 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 4** (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*

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

**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:

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

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.

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

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

[2] 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".

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,

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

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

$$\Pr\{A\} \leqslant \Pr\{B\} + \Pr\{A|\overline{B}\} \tag{7}$$

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

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

which concludes the induction. ■

**Exercise 5**

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
    if b = 0
    then return "f is identically 0"
    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 4, 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.

**Exercise 6** 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 $f(1)$, $f(2)$, ..., $f(k+1)$; 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 7** 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 4). 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 8** 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 9** 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. ■

# 5 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 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$'(T, P, d)$, which is a Las Vegas algorithm, as 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 and 7 in [MR].

---

*Assaf Kfoury*
*Created: 96.01.18*
*Modified: 96.02.26*