Advanced Algorithms **Fall 2015**

Lecture 7: Randomized Algorithms, Dec. 08, 2015

Faculty: K.R. Chowdhary *: Professor of CS*

7.1 Introduction

Goal of Deterministic Algorithm: To prove that the algorithm solves the problem correctly (always) and quickly (typically, the number of steps should be polynomial in the size of the input) (see fig. 7.1.

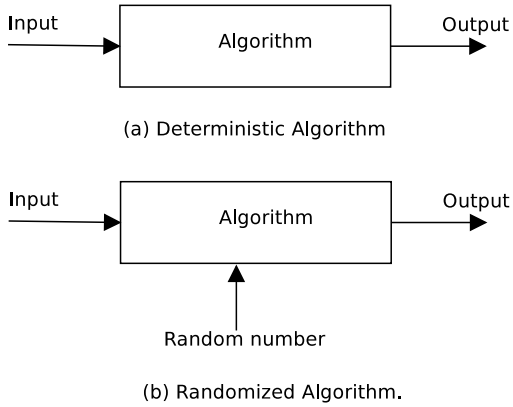

Figure 7.1: (a) Deterministic algorithm, (b) Random algorithm

Goal of Randomized Algorithm: In addition to input, algorithm takes a source of random numbers and makes random choices during execution. The behavior can vary even on a fixed input. The design of the algorithm and the analysis show that this behavior is likely to be good, on every input. The likely hood is over the random numbers only. While analyzing the algorithm, we should not be confused with the probabilistic analysis of the algorithms, where input is assumed to be from a probability distribution, which show that the algorithm works for most inputs.

There are two approaches:

- Monte Carlo: A Monte Carlo algorithm runs for a fixed number of steps, and produces an answer that is correct with probability $\geq 1 / 3$.
- Las Vegas: A Las Vegas algorithm always produces the correct answer; its running time is a random variable whose expectation is bounded (say by a polynomial).

For both, the probabilities/expectations are only over the random choices made by the algorithm, i.e., independent of the input. Thus independent repetitions of Monte Carlo algorithms drive down the failure probability exponentially.

The advantages of both are: Simplicity, and Performance. For many problems, a randomized algorithm is the simplest, the fastest, or both

Scope: Following are some applications of randomized (also called probabilistic) algorithms.

- Number-theoretic algorithms: Primality testing (Monte Carlo).
- Data structures: Sorting, order statistics, searching, computational geometry.
- Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
- Mathematical programming: Faster algorithms for linear programming. Rounding linear program solutions to integer program solutions.
- Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
- Counting and enumeration: Matrix permanent. Counting combinatorial structures.
- Parallel and distributed computing: Deadlock avoidance, distributed consensus.
- Probabilistic existence proofs: Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.
- Derandomization: First devise a randomized algorithm, then argue that it can be "derandomized" to yield a deterministic algorithm.

7.2 Sequential Search

Consider the simplest problem: search for a given element in a set of n integers. If the numbers are given one by one (this is called an online problem), the obvious solution is to use sequential search. That is, we compare every element, and, in the worst case, we need n comparisons (either it is the last element or it is not present). Under the traditional RAM model, this algorithm is optimal. This is the algorithm used to search in an unsorted array storing n elements, and is advisable when n is small or when we do not have enough time or space to store the elements (for example in a very fast communication line). Clearly, $U_n = n$. If finding an element in any position has the same probability, then $ES_n = (n+1)/2$.

7.2.1 Randomized Sequential Search

We can improve the worst case of sequential search in a probabilistic sense, if the element belongs to the set (successful search) and we have all the elements in advance (off-line case). Consider the following randomized algorithm. We flip a coin. If it is a head, we search the set from 1 to n, otherwise, from n to 1. The worst case for each possibility is n comparisons. However, we have two algorithms and not only one. Suppose that the element we are looking for is in position i and that the coin is fair (that is, the probability of head or tail is the same). So, the number of comparisons to find the element is i, if it is a head, or $n - i + 1$ if it is a tail. So, averaging over both algorithms (note that we are not averaging over all possible inputs), the expected worst case is

$$\frac{1}{2} \times i + \frac{1}{2} \times (n - i + 1) = \frac{n+1}{2}$$

which is independent of where the element is! This is better than n. In other words, an adversary would have to place the element in the middle position because he/she does not know which algorithm will be used.

7.3 Randomized Algorithm

The randomized algorithms are also called probabilistic algorithms. One important way to provide a performance guarantee is to introduce randomness. For example, the quicksort algorithm for sorting that we study (perhaps the most widely used sorting algorithm) is quadratic in the worst case, but randomly ordering the input gives a probabilistic guarantee that its running time is linearithmic. Every time you run the algorithm, it will take a different amount of time, but the chance that the time will not be linearithmic is so small as to be negligible. Similarly, the hashing algorithms for symbol tables (again, perhaps the most widely used approach) are linear-time in the worst case, but constant-time under a probabilistic guarantee. These guarantees are not absolute, but the chance that they are invalid is less than the chance your computer will be struck by lightning. Thus, such guarantees are as useful in practice as worst-case guarantees.

7.3.1 Sequences of operations

For many applications, algorithm's "input" might be not just data, but a sequence of operations performed by the client. For example, a pushdown stack where the client pushes N values, then pops them all, may have quite different performance characteristics from one where the client issues an alternating sequence N of push and pop operations. Our analysis has to take both situations into account (or to include a reasonable model of the sequence of operations).

A randomized algorithm flips coins during its execution to determine what to do next. When considering a randomized algorithm, we usually care about its expected worst-case performance, which is the average amount of time it takes on the worst input of a given size. This average is computed over all the possible outcomes of the coin flips during the execution of the algorithm. We may also ask for a high-probability bound, showing that the algorithm does not consume too much resources most of the time.

In studying randomized algorithms, we consider almost the same issues as for deterministic algorithms:

- how to design a good randomized algorithm, and
- how to prove that it works within given time or error bounds.

The main difference is that it is often easier to design a randomized algorithm, because the randomness turns out to be a good substitute for cleverness, but it is harder to analyze it. So, much of what one does is develop a good technique for analyzing the often very complex random processes that arise in the execution of an algorithm. In doing so we should use techniques already developed for probability and statistics.

Example 7.1 *Suppose you have two copies of the same long binary file at two different locations: call them F_1 and F_2. Suppose a hacker has tried to disturb F_1. You would like to check whether the file is corrupted.*

Solution. Obviously, this can be achieved by sending the first file to the other location and then comparing them. However, we would like to minimize the number of bits that are communicated. This can be done with random check-bits. Assuming that you have the same random number generator at both ends, you can do the following: Compute a random subset S of the bits by tossing a coin for each bit. Then compute the XOR of the bits of F_i corresponding to S: call the result c_i. That is, $c_i = F_i \oplus S$, or $c_1 = F_1 \oplus S$, and $c_2 = F_2 \oplus S$. Obviously, if F_i not corrupted then $c_1 = c_2$.

Lemma 7.2 *If $F_1 \neq F_2$ then $c_1 \neq c_2$ 50% of time.*

Proof. Consider the last bit in which the files differ: call it b. Let u_1 and u_2 be the calculations so far. If $u_1 = u_2$, then $c_1 \neq c_2$ iff S includes b: which is a 50:50 event. However, if $u_1 \neq u_2$, then $c_1 \neq c_2$ iff S does not include b, again a 50:50 event. □

Thus, if $F_1 \neq F_2$ there is a 50:50 chance that this random check-bit is different. Thus we have a typical Monte carlo algorithm. If you perform this computation 200 times, and each check bit matches, then there is only a 1 in 2^{200} chance that the files are different. 1 in 2^{200} is roughly 1 in 10^{66}, and that chance is extremely less. For all practical purposes, we can safely say that the algorithm will give a correct result with a probability of 1. □

This reduces the communication Complexity, because otherwise it requires to send n bits to the other side.

Simplicity: This is the first and foremost reason for using randomized algorithms. There are numerous examples where an easy randomized algorithm can match (or even surpass) the performance of a deterministic algorithm.

Example 7.3 *The Merge-sort algorithm is asymptotically the best deterministic algorithm. It is not too hard to describe. However, the same asymptotic running time can be achieved by the simple randomized QuickSort algorithm. The algorithm picks a random element as a pivot and partitions the rest of the elements: those smaller than the pivot and those bigger than the pivot. Recurse on these two partitions.* □

Speed: For some problems, the best known randomized algorithms are faster than the best known deterministic algorithms. This is achieved by requiring that the correct answer be found only with high probability or that the algorithm should run in expected polynomial time. This means that the randomized algorithm may not find a correct answer in some cases or may take a very long time.

Example 7.4 *Checking if a multi-variate polynomial is the zero polynomial. There is no known deterministic P-time algorithm that tells if the given multi-variate polynomial is the zero polynomial. On the other hand, there is a very simple and efficient randomized algorithm for this problem: just evaluate the given polynomial at random points. Note that such a polynomial could be represented implicitly. e.g., as a determinant of a matrix whose entries contain different variables (see figure 7.2).*

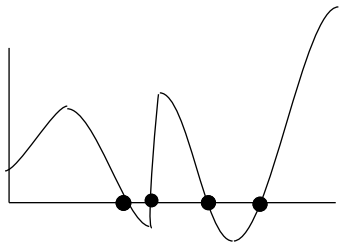

Figure 7.2: Polynomial curve.

It is important to note that a randomized algorithm is not the same thing as probabilistic analysis. In probabilistic analysis, (deterministic) algorithms are analyzed assuming random input, e.g., in sorting one might analyze the deterministic Quicksort (with fixed pivot) on the input which is a random permutation of n elements.

On the other hand, randomized algorithms use random coin flips for their execution, which amounts to picking a deterministic algorithm from a suite of algorithms. Moreover, randomized algorithms give guarantees for the worst case input.

7.4 Probabilistic Analysis of Algorithms

Consider that you want to hire an office assistant. For this you need to pay small fee to employment agency for sending each candidate for interview. Since, actually, hiring a candidate is more costly, as you have to pay substantial fee to the employment agency. So, after interviewing the new candidate, if the new is better, you will fire the previous and hire the new. you need to estimate all the fee (i.e., expenses).

Let the candidates are numbered 1 through n. The procedure is such that after interviewing the i-th candidate you are able to decide weather to hire it or not. Initially you create a dummy variable, numbered zero (0) who is less qualified than the rest of. Following is the algorithm.

Algorithm 1 Hire-Assistant (n)

```
1: best = 0 //0 is least qualified dummy variable
2: for i = 1 to n do
3:    interview candidate i
4:    if i is better than the best then
5:       best = i
6:       hire candidate i
7:    end if
8: end for
```

On the surface, analyzing cost of this algorithm may seem very difficult from analyzing running time of, say, *Mergesort*. The analytical techniques used, however, are identical whether we are analyzing the number of times certain basic operations are executed.

Interviewing has low cost, say C_i, but hiring is expensive, say, C_h. Let m be the number of candidates hired, the total cost associated with the algorithm is $O(C_i n + C_h m)$. No matter how many we hire, we always interview n number of candidates, so cost is $C_i n$ for interviwing. Thus, what is to be found out is $C_h m$ - the hiring cost. This quantity varies with each run of the algorithm. This, above serves as a model for common computing.

Worst Case: In the worst case, we hire every one we interview. This comes when candidates come in strictly increasing order of quality (or qualification). In that case it is $C_h n$. But that is not true. So, it is natural to ask "What we expect to happen in a typical or average case?"

7.4.1 Probabilistic Analysis

Probabilistic analysis is used to analyze running time of the algorithm. For this we must use knowledge of or make assumptions about, the distribution of inputs. Then we analyze our algorithm, computing an average running time, where we take average over the distribution of the possible inputs. Thus, in effect, we are averaging the running time over all possible inputs. This we call as "average case running time".

We must be very careful in deciding the distribution of inputs. We may reasonably assume about possible set of inputs, and then use probabilistic analysis as a technique for designing the efficient algorithm, and means for gaining insight into the problem. Where we cannot describe the distribution of inputs, we cannot use probabilistic analysis.

In the hiring problem, the candidates came in random order means they can be in any order in the permutation of $\langle 1, \ldots, c \rangle$.

Example 7.5 *A Trivial example: Find the Lady.*

Let us consider a variant of the classic card game "Find the Lady". Here a dealer puts down three cards and we want to find a specific card among the three (say, the Queen of Spades). In this version, the dealer will let us turn over as many cards as we want, but each card we turn over will cost us a dollar, irrespective of whether you get a queen. If we find the queen, we get two dollars.

Because this is a toy example, we assume that the dealer is not cheating.

A deterministic algorithm tries the cards in some fixed order. A clever dealer will place the Queen in the last place we look: so the worst-case payoff is a loss of a dollar.

In the average case, if we assume that all three positions are equally likely to hold the target card, then we turn over one card a third of the time, two cards a third of the time, and all three cards a third of the time. This gives an expected payoff of

$$\frac{1}{3}(1+2+3)-2=0.$$

But this requires making assumptions about the distribution of the cards, and we are no longer doing worst-case analysis.

The trick to randomized algorithms is that we can can obtain the same expected payoff even in the worst case by supplying the randomness ourselves. If we turn over cards in a random order, then the same analysis for the average case tells us we get the same expected payoff of 0but unlike the average case, we get this expected performance no matter where the dealer places the cards. □

A randomized algorithm is one that makes random choices during its execution. The behavior of such an algorithm may thus, be random even on a fixed input. The design and analysis of a randomized algorithm focuses on establishing that it is likely to behave "well" on every input; the likelihood in such a statement depends only on the probabilistic choices made by the algorithm during execution and not on any assumptions about the input.

It is especially important to distinguish a randomized algorithm from the average-case analysis of algorithms, where one analyzes an algorithm assuming that its input is drawn from a fixed probability distribution. With a randomized algorithm, in contrast, no assumption is made about the input.

Throughout this discussion we assume the RAM model of computation, in which we have a machine that can perform the following operations involving registers and main memory: input-output operations, memory-register transfers, indirect addressing, and branching and arithmetic operations. Each register or memory location may hold an integer that can be accessed as a unit, but an algorithm has no access to the representation of the number. The arithmetic instructions permitted are $+, , \times$, and $/$. In addition, an algorithm can compare two numbers, and evaluate the square root of a positive number. In this chapter $E[X]$ will denote the expectation of a random variable X, and $P_r[A]$ will denote the probability of an event A.

Review Questions

1. List the advantages of randomized algorithms.
2. What are the applications of randomized algorithms? Suggest any five with examples for each case.
3. If randomized algorithms are simple and efficient, then why they are not used always?
4. What is difference between average analysis of deterministic algorithms and average case analysis of random algorithms?

5. What are the disadvantages of randomized algorithms?
6. Are the randomized and probabilistic algorithms same? Explain, why/why not?
7. The deterministic algorithms are modeled using Turing machine. What is used to model a randomized algorithm?

Exercises

1. Using the World Wide Web or any book on probability, make yourself familiar with the following concepts and definitions:
   (a) Union bound (Boole's inequality)
   (b) Conditional probability
   (c) Random variables, expectation and variance of random variables
   (d) Linearity of expectation
   (e) Independence of events

   Some resources:http://en.wikipedia.org/wiki/Probability_theory, http://mathworld.wolfram.com/

References

[1] Thomas H. Corman, et al., "Introduction to Algorithms, 3rd ed." *PHI*, 2009.

[2] Mikhail J. Atallah and Marina Blanton, "Algorithms and Theory of Computation handbook, Second Edition, Special Topics and Techniques", CRC Press, Taylor and Francis Group - A Chapman and Hall Book, 2010.

[3] Allen B. Tucker, Jr., "The computer Science and Engineering Handbook," *CRC Press*, 1997.