Probabilistic/Randomized Algorithms

Ming-Hwa Wang, Ph.D.
COEN 279/AMTH 377 Design and Analysis of Algorithms
Department of Computer Engineering
Santa Clara University

Probabilistic or Randomized algorithm

- At least once during the algorithm, a random number is used to make a decision instead of spending time to work out which alternative is best.
- The worst-case running time of a randomized algorithm is almost always the same as the worst-case running time of the non-randomized algorithm.
- A good randomized algorithm has no bad input, but only bad random numbers.
- The random numbers are important, and we can get an expected running time, where we now average over all possible random numbers instead of over all possible inputs, or the mean time that it would take to solve the same instance over and over again.
- A randomized algorithm runs quickly but occasionally makes an error. The probability of error can, however, be make negligibly small. Any purported solution can be verified efficiently for correctness.
- A randomized algorithm may give probabilistic answers which are not necessarily exact.
- The same algorithm may behave differently when it is applied twice to the same instance. Its execution time, and even the result obtained, may vary considerable from one use to the next. If the algorithm gets stuck (e.g., core dump), simply restart it on the same instance for a fresh chance of success. If there is more than one correct answer, several different ones may be obtained by running the probabilistic algorithm more than once.
- An expected running time bound is somewhat stronger than an average-case bound, but is weaker than the corresponding worst-case bound.

Random Number Generators

- True randomness is virtually impossible to do on a computer. Pseudorandom numbers. What really needed is a sequence of random numbers appear independently.
- The linear congruential generator: $x_{i+1} = Ax_i \% M$, where x_0 is the seed and $1 \le x_0 < M$. If M is prime, x_i is never 0. After M-1 numbers, the sequence repeat (period of M-1). Some choices of A gets shorter period than M-1.
- If M is chosen to be a large, 31-bit prime, the period should be significantly large for most applications. $M = 2^{31} - 1 = 2{,}147{,}483{,}647$ and $A = 48{,}271$.
- Same sequence occurs all the time for easy debugging, and input seed (e.g., use system clock) for real runs.
- Usually a random real number in the open interval (0,1), which can be done by dividing by M.
- Multiplication overflow prevention: let $Q = M / A = 44{,}488$ and $R = M \% A = 3{,}399$,
  - $x_{i+1} = Ax_i \% M = A(x_i \% Q) - R(x_i / Q) + M\delta(x_i)$, where $\delta(x_i) = x_i / Q - Ax_i / M = 1$ iff the remaining terms evaluate to less than zero, 0 otherwise.
  - $x_{i+1} = Ax_i \% M = Ax_i - M(Ax_i / M) = Ax_i - M(x_i / Q) + M(x_i / Q) - M(Ax_i / M) = Ax_i - M(x_i / Q) + M(x_i / Q - Ax_i / M) = A(Q(x_i / Q) + x_i \% Q) - M(x_i / Q) + M(x_i / Q - Ax_i / M) = (AQ - M)(x_i / Q) + A(x_i \% Q) + M(x_i / Q - Ax_i / M) = -R(x_i / Q) + A(x_i \% Q) + M(x_i / Q - Ax_i / M) = A(x_i \% Q) - R(x_i / Q) + M\delta(xi)$

Numerical Probabilistic Algorithms

For certain real-life problems, computation of an exact solution is not possible even in principle, e.g., uncertainties in the experimental data, digital computers handle only binary values, etc. For other problems, a precise answer exists but it would take too long to figure it out exactly. Numerical algorithms yield a confidence interval, and the expected precision improves as the time available to the algorithm increase. The error is usually inversely proportional to the square root of the amount of work performed.

- Buffon's Needle: throw a needle at random on a floor made of planks of constant width, if the needle is exactly half as long as the planks in the floor and if the width of the cracks between the planks are zero, the probability that the needle will fall across a crack is $1/\pi$. The probability that a randomly thrown needle will fall across a crack is $2\lambda/\omega\pi$, where λ is needle length and ω is plank width. The result estimate will be between $\pi - \varepsilon$ and $\pi + \varepsilon$ with probability at least ρ (desired reliability).
- Numerical integration - Monte Carlo integration: deterministic integration algorithms are easy to be fooled, and very expensive when evaluating a multiple integral. The hybrid techniques that partly systematic and partly probabilistic is called quasi Monte Carlo integration.
- Probabilistic Counting:
  - Counting twice as far to up to $2^{n+1} - 2$ by initialize to 0, each time tick is called, flip a fair coin. If it comes up head, add 1 to the register, otherwise, do nothing. When count is called, return twice the value stored in the register.

- Counting exponentially farther from 0 to $_2 2^n$-1 $_-$ $_1$. Keep in the register an estimate of the logarithm of the actual number of ticks and count(*c*) returns 2^c-1. Keep the relative error in control instead of absolute.

Monte Carlo Algorithms

Monte Carlo algorithms give exact answer with high probability whatever the instance considered, although sometimes they provide a wrong answer. Generally you cannot tell if the answer is correct, but you can reduce the error probability arbitrarily by allowing the algorithm more time (amplifying the stochastic). A Monte Carlo algorithm is p-correct if it returns a correct answer with probability at least p ($0 < p < 1$), whatever the instance considered. p depends on the instance size but not on the instance itself.

- Verifying Matrix Multiplication:
  - straightforward matrix multiplication algorithm $\Theta(n^3)$, Strassen's algorithm $\Omega(n^{2.37})$
  - Let $D = AB - C$, $S \subseteq \{1,2, .., n\}$, and $\Sigma_S(D)$ denote the vector of length n obtained by adding pointwise the rows of D indexed by the elements of S. $\Sigma_S(D)$ is always 0 if AB equal C, otherwise, assume i be an integer such that the i^{th} row of D contains at least one nonzero element. The probability that $\Sigma_S(D) \neq 0$ is at least one-half. Let X be a binary vector of length of n such that $X_j = 1$ if $j \in S$ and $X_j = 0$ otherwise. Then $\Sigma_S(D) = XD$, and we want to verify if $XAB = XC$, where $(XA)B$ need $\Theta(n^2)$. Getting the answer false just once allows you conclude that $AB \neq C$. The probability that k successive calls each return the wrong answer is at most 2^{-k}, so it is $(1 - 2^{-k})$ correct. Alternatively, Monte Carlo algorithms can be given an explicit upper bound on the tolerable error probability in $\Theta(n^2 \lg \varepsilon^{-1})$.
- Primality Testing
  - $O(2^d/2)$ to test whether a d-digit number is a prime
  - Randomized polynomial-time algorithm: if the algorithm declares that the number is not prime, then it is certainly not a prime. If the algorithm declares that the number is a prime, then with high probability but not 100% sure, the number is prime.
  - Fermat's Lesser Theorem: If P is prime, and $0 < A < P$, then $A^{P-1} \equiv 1 \% P$
  - Pick $1 < A < N$-1 at random. If $A^{N-1} \equiv 1 \% N$, declare that N is probably prime, otherwise declare that N is definitely not prime.
  - False witness of primality: Carmichael numbers are not prime but satisfy $A^{N-1} \equiv 1 \% N$ for all $0 < A < N$ that are relatively prime to N.
  - If P is prime and $0 < X < P$, the only solutions to $X^2 = 1 \% P$ are $X = 1, P$-1.
- Skip Lists
  - Every 2^ith node has a pointer to the node 2^i ahead of it. The total number of pointers has only doubled, but now at most $\lceil \lg N \rceil$ nodes are examined during a search. The search consists of either advancing to a new node or dropping to a lower pointer in the same node.
  - A level k node is a node that has k pointers, the i^{th} pointer in any level k node ($k \geq i$) points to the next node with at least i levels. Roughly half the nodes are level 1 nodes, roughly a quarter are level 2, and, in general, approximately $1/2^i$ nodes are level i. We choose the level randomly.
  - Find: start at the highest pointer at the header, traverse along this level until find that the next node is larger than the one we are looking for (or nil). When this occurs, go to the next lower level and continue the strategy. When progress is stopped at level 1, either we are in front of the node we are looking for, or it is not in the list.
  - Insert: proceed as in a Find, and keep track of each point where we switch to a lower level. The new node, whose level is determined randomly, is then spliced into the list.
  - $O(\lg N)$ expected cost
  - Skip lists need an estimate of the number of elements that will be in the list to determine the number of levels. Different level of nodes need different type declarations.

Las Vegas Algorithms

Las Vegas algorithms make probabilistic choices to help guide them more quickly to a correct solution, they never return a wrong answer. Two main categories of Las Vegas algorithms: it take longer time to solve a problem when unfortunate choice are made (e.g., Quicksort), and alternatively, they allow themselves go to a dead end and admit that they cannot find a solution in this run of the algorithm. A Las Vegas algorithm has the Robin Hood effect, with high probability, instances that took a long time deterministically are now solved much faster, but instances on which the deterministic algorithm was particularly good are slowed down to average. Let $p(x)$ be the probability of success of the algorithm, then the expected time $t(x)$ is $1/p(x)$. However, a correct analysis must consider separately the expected time taken by LV(x) in case of success $s(x)$ and in case of failure $f(x)$. $t(x) = s(x) + ((1-p(x))/p(x))f(x)$.

- The Eight Queens Problem
  - Combine backtracking with probabilistic algorithm, first places a number of queens on the board in a random way, and then uses backtracking to try and add the remaining queens without reconsidering the positions of the queens that were placed randomly.

- The more queens we place randomly, the smaller the average time needed by the subsequent backtracking stage, whether it fails or succeeds, but the greater the probability of failure. This is the fine-tuning knob.

- Probabilistic Quickselect and Quicksort
- Universal Hashing
  - Las Vegas hashing allows us to retain the efficiency of hashing on the average, without arbitrarily favoring some programs at the expense of others. Choose the hash function randomly at the beginning of each compilation and again whenever rehashing becomes necessary, ensure that collision lists remain reasonably well-balanced with high probability.
  - Universal hashing: Let $U = \{1,2, .., a\text{-}1\}$ be the universe of potential indexes for the associative table, and let $B = \{1,2, ..,N\text{-}1\}$ be the set of indexes in the hash table. Let two distinct x and y in U, a set H of functions from U to B, and $h:U \rightarrow B$ is a function chosen randomly from H, H is a universal$_2$ class of hash functions if the probability that $h(x) = h(y)$ is at most $1/N$. Let p be a prime number at least as large as a, and i, j be two integers ($1 \leq i < p$ and $0 \leq j < p$), then $h_{ij}(x) = ((ix + j)\%p)\%N$, and H is universal$_2$.
- Factorizing Large Integers
  - The factorization problem consists of finding the unique decomposition of n into a product of prime factors. The splitting consists of finding one nontrivial divisor of n, provided n is composite. Factorizing reduces to splitting and primality testing. An integer is k-smooth if all its prime divisors are among the k smallest prime numbers. k-smooth integers can be factorized efficiently by trial division if k is small. A hard composite number is the product of two primes of roughly equal size.
  - Let n be a composite integer, Let a and b be distinct integers between 1 and n-1 such that $a + b \neq n$. If $a^2 \% n \equiv b^2 \% n$, then $\gcd(a+b, n)$ is a nontrivial divisor of n.