__/__/____

Randomized Algorithms

A randomized algorithm makes use of a randomizer, e.g., a random number generator, thereby inducing unpredictability in its output and/or time through different runs, though other inputs remain same.

- The unpredictability contrasts it from deterministic algorithms.

- Since output and/or time may be different for different runs depending on the randomizer, a randomized algorithm can be viewed as a family of algorithms.

- Randomized algorithms whose outputs might differ from run to run are called Monte Carlo algorithms. Consider a problem with only two possible answers, yes and no. A Monte Carlo algorithm to solve such a problem might give incorrect answers, depending on the input from the randomizer. We require that the probability of an incorrect answer from a Monte Carlo algorithm be low.

  In contrast, there are randomized algorithms that always produce the same output for the same input, irrespective of the randomizer output, though the time may vary with different runs. They are called Las Vegas algorithms, to contrast from Monte Carlo algorithms.

__/__/____

Examples:

- An assay $a[0..n-1]$ has $n/2$ distinct elements and $n/2$ copies of another element. The problem is to identify the repeated element.

Any deterministic algorithm for solving this problem will need at least $\frac{n}{2}+2$ time steps in the worst case; $\frac{n}{2}$ for the distinct elements, then one more and then the repetition!

A randomized algorithm:

```
fn RepeatedElement(a, n) {
    while (true) {
        i = random() mod n; j = random mod n;
        // so, i and j are in [0..n-1]
        if ((i ≠ j) and (a[i] = a[j])) then return i;
    }
}
```

Any iteration of the while loop will be successful in identifying the repeated number if i is any one of the $n/2$ assay indices corresponding to the repeated element and j is any one of the same $n/2$ indices but other than i. The probability that the algorithm quits in any given iteration of the while loop is

$$P = \frac{n/2}{n} \times \frac{n/2 - 1}{n} = \frac{n/2\,(n/2-1)}{n^2} \geq \frac{1}{5} \text{ for all } n \geq 10.$$

This implies that the probability that the algorithm does not quit in a given iteration is $< 4/5$.

Therefore, the probability that the algorithm does not quit in 10 iterations is $< (4/5)^{10} < 0.1074$.

So, the algorithm will terminate in 10 iterations or less with a probability ≥ 0.8926.

The probability that the algorithm does not terminate in 100 iterations is $< (4/5)^{100} < 2.04 \times 10^{-10}$. So, the algorithm will almost certainly quit in 100 iterations or less.

This is a Las Vegas algorithm.

In contrast, if $n = 2 \times 10^6$, for example, any deterministic algorithm will spend at least one million time steps, as opposed to the 100 iterations of the Las Vegas algorithm.

In general, the probability that the algorithm does not quit in the first $c\alpha \log n$ (c is a constant and α is a probability parameter) iterations is $< (4/5)^{c\alpha \log n} = n^{-c\alpha \log(5/4)}$ which will be $< n^{-\alpha}$ if we choose $c \geq \frac{1}{\log(5/4)}$.

Thus the algorithm terminates in $\frac{1}{\log(5/4)} \alpha \log n$ iterations or less with probability $\geq 1 - n^{-\alpha}$. Since each iteration of the while loop takes $O(1)$ time, the run time of the algorithm is $O(\log n)$.

__/__/____

- Primality testing:

The following algorithm detects primality with a high probability. If the input is prime, the algorithm never gives an incorrect answer. But, if the input is composite, i.e, non-prime, then there is a small probability that the answer may be incorrect. This is a Monte Carlo algorithm with one-sided error.

Theorem 1: Fermat's theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for any integer $a < n$.

Theorem 2: The equation $x^2 \equiv 1 \pmod{n}$ has exactly two solutions, namely 1 and $n-1$, if n is a prime.

Cor: If the equation $x^2 \equiv 1 \pmod{n}$ has roots other than 1 and $n-1$, then n is composite.

Note: Any integer x which is neither 1 nor $n-1$ but which satisfies $x^2 \equiv 1 \pmod{n}$ is said to be a non-trivial square root of 1 mod n.

Algorithm explanation: With inspiration from Fermat's theorem randomly choose an $a < n$ and check whether $a^{n-1} \equiv 1 \pmod{n}$. Call this Fermat's equation. If Fermat's eqn. is not satisfied, n is composite. If the eqn. is satisfied, we try some more random a's. If on each a tried, Fermat's eqn. is satisfied,

we output n is prime; otherwise we output, n is composite.
Here large is a number sufficiently large that ensures a probability of correctness of $\geq 1 - n^{-\alpha}$, a high probability.

```
fn PrimeO(n, α) {
    // Returns true if n is a prime and false otherwise
    // α is the probability parameter
    q = n - 1;
    for i = 1 to large {// Specify large
        m = q; y = 1;
        a = random() mod q + 1;
        z = a;
        // Compute a^(n-1) mod n in the while loop next
        while (m > 0) {
            while (m mod 2 = 0) {
                z = z^2 mod n;
                m = ⌊m/2⌋;
            }
            m = m - 1; y = (y * z) mod n;
        }
        if (y ≠ 1) then return false;
        // If a^(n-1) mod n is not 1, n is not a prime
    }
    return true;
}
```

If the input is prime, the algorithm never outputs an incorrect answer. If n is composite, the output may sometimes be incorrect because Fermat's eqn. may be satisfied for some $1 < a < n$ when n is composite. In fact, there are composite numbers, known as Carmichael numbers, for which every a that is less than and relatively prime to n will satisfy Fermat's eqn, e.g 561 and 1105. There are only 255 Carmichael numbers less than 1000000000.

In the next algorithm, known as Miller-Rabin's algorithm for primality testing, we have an additional section where we look for nontrivial square roots of n. It is assumed n is odd, naturally. A nontrivial square root of 1 leads to compositeness of n. The a in such case is called a witness to the compositeness of n.

Theorem: There are at least $\frac{n-1}{2}$ witnesses to the compositeness of n, if n is composite and odd.

```
fn Prime(n, α) {
    // Returns true if n is a prime and false otherwise
    // α is the probability parameter
    q = n-1;
    for i = 1 to α * log(n) {
        m = q; y = 1;
        a = Random() mod q+1;
        z = a;
        // Compute a^(n-1) mod n in the while loop next
        while (m > 0) {
            while (m mod 2 = 0) {
                x = z; z = z^2 mod n;
                // If x is a nontrivial sq root of 1, then
                // n is not a prime
                if ((z = 1) and (x ≠ 1) and (x ≠ q)) then
                    return false;
                m = ⌊m/2⌋;
            }
            m = m-1; y = (y * z) mod n;
        }
        if (y ≠ 1) return false;
        // If a^(n-1) mod n is not 1, n is not a prime
    }
    return true;
}
```

Assume that n is composite, since the algorithm will be correct when n is prime. The probability that a randomly chosen a will be a witness is $\geq \frac{n-1}{2n}$, which is very nearly equal to $1/2$. This means that a randomly chosen a will fail to a witness with probability $\leq 1/2$.

Therefore, the probability that none of the first $\alpha \log n$ a's chosen is a witness is $\leq \left(\frac{1}{2}\right)^{\alpha \log n} = \frac{1}{n^{\alpha}}$. So, the algorithm Prime will give an incorrect answer with only probability $\leq n^{-\alpha}$.

The run time of the outermost while loop is nearly the same as that of an exponentiation algorithm and equal $O(\log n)$. Since the while loop is executed $O(\log n)$ times, the run time of the whole algorithm is $O(\log^2 n)$.