Chapter 5: Probabilistic and Parallel Algorithms

Contents

- Probabilistic algorithms
  - Monte Carlo algorithm - Primality testing
  - Las Vegas algorithm - N-queen problem
- Parallel algorithms
  - Finding connected components in a graph
  - Parallel sorting

Probabilistic Algorithms

Different types of algorithms

Suppose an algorithm produces X as the maximum value, when the true maximum is Y.

- An **exact** or **deterministic** algorithm guarantees X = Y.
- An **approximation** algorithm guarantees X‘s rank is “close to” Y‘s rank
- A **probabilistic** algorithm gives up its guarantee of getting X = Y and may say “X is usually Y” in exchange for an improved running time. They sacrifice reliability for speed.

Disadvantages of deterministic algorithms

Some problems may be too complex to solve. Deterministic algorithms to solve such problems may have high resource (time and memory) consumption.

If there is more than one correct answer, deterministic algorithms always come up with the same answer.

Some applications require nondeterminism (unpredictability), e.g., in card games to avoid cheating.

Probabilistic/Randomized Algorithms

Probabilistic or randomised algorithms employ some form of random element in an attempt to obtain improved performance.

- Sometimes, improvement can be dramatic, from intractable to tractable
- Some loss in reliability of results can occur
- Different runs may produce different results for the same input, reliability may be improved by running several times
- Two main categories
  - **Monte Carlo algorithms** have good running time, their result is not guaranteed
  - **Las Vegas algorithms** have a guaranteed result, but do not guarantee fast running time

Monte Carlo Algorithms

These are randomized algorithms which may produce incorrect results with some small probability, but whose execution time is deterministic.

If such an algorithm is run multiple times with independent random choices each time, the failure probability can be made arbitrarily small at the cost of a slight increase in computing time .

Primality Test

Prime number is a natural number greater than 1 number whose only factors are 1 and itself.

Primality test is an algorithm for determining whether an input number is prime.

No known algorithm can solve this problem with certainty in a reasonable time when the number to be tested has more than a few hundred decimal digits.

The Simplest Primality Test

Given an input number, n, check whether it is divisible by all numbers from 2 to n-1. If we find any number that divides, n is composite, otherwise it is prime.

Time complexity: $O(n)$

Optimized way: Instead of checking till n-1, we can check till $\sqrt{n}$ because a larger factor of n must be a multiple of a smaller factor that has been already checked, i.e. for any divisor $p > \sqrt{n}$, there must be another divisor $n/p < \sqrt{n}$.

Time Complexity: $O(\sqrt{n})$

Probabilistic Primality Test

- Many popular probabilistic primality tests use, apart from the input number `n`, some other numbers `a` which are chosen at random from some sample space.
- They never report a prime number as composite, but it is possible for a composite number to be reported as prime.
- The probability of error can be reduced by repeating the test with several independently chosen values of a.
- Some popular probabilistic primality tests:
  - Fermat Primality Test
  - Miller-Rabin Primality Test

Fermat Primality Test

Fermat primality test is based on the **Fermat's Little Theorem**, which states that

> If n is a prime number, then for any integer a, $1 < a < n-1$, $a^{n-1} \equiv 1 \pmod{n}$
> *read it as a^{n-1} and 1 are congruent modulo n* [i.e., $a^{n-1} \% n = 1$]
>
> Examples:
>
> - Since 5 is prime,
> $2^4 \equiv 1 \pmod{5}$ [or $2^4 \% 5 = 1$], $3^4 \equiv 1 \pmod{5}$ and $4^4 \equiv 1 \pmod{5}$
> - Since 7 is prime,
> $2^6 \equiv 1 \pmod{7}$, $3^6 \equiv 1 \pmod{7}$, $4^6 \equiv 1 \pmod{7}$, $5^6 \equiv 1 \pmod{7}$ and $6^6 \equiv 1 \pmod{7}$

Fermat Primality Test

Given an integer `n`, choose some integer `a` coprime to `n` and calculate a^{n-1} `modulo n`. If the result is different from 1, then `n` is composite. If it is 1, then `n` may be prime.

If n is prime, then this method always returns true. If n is composite (or non-prime), then it may return true or false.

The probability of producing incorrect results for composite is low and can be reduced by doing more iterations.

Fermat Primality Test

Inputs: `n`: a value to test for primality, `n>3;` `k`: a parameter that determines the number of times to test for primality

Output: composite if `n` is composite, otherwise probably prime

Steps:

```
1. Repeat k times:
    a. Pick a randomly in the range [2, n - 2]
    b. If $a^{n-1} \not\equiv 1 \pmod{n}$, then return composite
2. Return probably prime
```

Fermat Primality Test

Example:

n = 17

Step 1: Pick `a` randomly in the range `[2, 15]`. Let's say a = 9.

Step 2: Check $a^{n-1} \neq 1 \pmod{n}$

9^{16} % 17 = 1

Repeat. Let's say a = 12. 12^{16} % 17 = 1

Fermat Primality Test

Example:

n = 239

Step 1: Pick a randomly in the range [2, 237] . Let's say a = 210.

Step 2: Check $a^{n-1} \not\equiv 1 \pmod{n}$

210^{238} % 239 = 1

Repeat. Let's say a = 25. 25^{238} % 239 = 1

Fermat Primality Test

Example:

n = 250

Step 1: Pick `a` randomly in the range `[2, 250]`. Let’s say a = 30.

Step 2: Check $a^{n-1} \not\equiv 1 \pmod{n}$

30^{249} % 250 = 0

Thus 250 is composite.

Fermat Primality Test

Time complexity: O(k log n). Note that the modular exponentiation of step 1.b takes O(log n) time.

Note that the Fermat Primality Test may fail even if we increase the number of iterations (higher k). There exist some composite numbers (e.g., the number 561) with the property that for every $a < n$ and coprime to n, we have $a^{n-1} \equiv 1 \pmod{n}$. Such numbers are called **Carmichael numbers.**

Fermat's primality test is often used if a rapid method is needed for filtering, for example in the key generation phase of the RSA public key cryptographic algorithm.

Miller-Rabin Primality Test

- It is also based on the Fermat's Little Theorem
- The result of this test will be whether the given number is a composite number or a probable prime number
- The probable prime numbers in the Miller-Rabin Primality Test are called strong probable numbers, as they have a higher chance of being a prime number than in the Fermat's Primality Test.
- A number n is said to be strong probable prime number to base a if one of these congruence relations hold:
  - $a^d \equiv \pm 1 \pmod{n}$
  - $a^{2^r d} \equiv -1 \pmod{n}$ for all $0 \leq r \leq s-1$

where n-1 $= 2^s$d

Miller-Rabin Primality Test

Steps to check whether n is prime or not:

1. Find $n-1=2^s*d$, such that d is an odd number
2. Choose a such that $1<a<n-1$
3. Compute $b_0=a^d \pmod{n}$. If b_o is +-1, n is probably prime else

   Compute $b_1=b^2_0 \pmod{n}$ …. $b_i=b^2_{i-1} \pmod{n}$ where $0 \le i \le s-1$
   until b_i is +1 or -1

   If b_i is !(-1) the number is composite

   If b_i is -1 the number is probably prime

Simulate for **561**. Fermat's returns it as a Prime, MR returns it as a composite

Check for 23 and 27.

Miller-Rabin Primality Test

Input #1: $n > 2$, an odd integer to be tested for primality
Input #2: k, the number of rounds of testing to perform
Output: "*composite*" if n is found to be composite, "*probably prime*" otherwise

```
let $s > 0$ and $d$ odd $> 0$ such that $n - 1 = 2^s d$
repeat $k$ times:
    $a \leftarrow$ random(2, $n - 2$)
    $x \leftarrow a^d \bmod n$
    repeat $s$ times:
        $y \leftarrow x^2 \bmod n$
        if $y = 1$ and $x \neq 1$ and $x \neq n - 1$ then
            return "composite"
        $x \leftarrow y$
    if $y \neq 1$ then
        return "composite"
return "probably prime"
```

Las Vegas algorithms - Characteristics

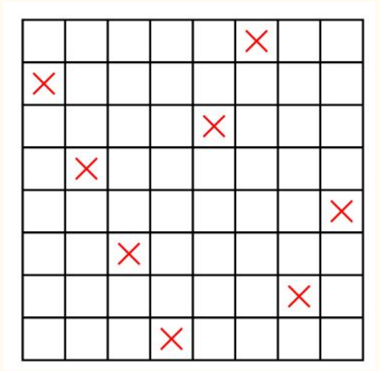
- Las Vegas algorithms make probabilistic choices to help guide them more quickly to a correct solution.
- Unlike Monte Carlo algorithms, they never return a wrong answer.
- Two main categories
    - Those which guarantee a correct solution though they may take longer if unfortunate choices are made
    - Those that may fail to produce a result, but if they return a result, it is always correct

Eight queens problem

Place 8 queens on a chess board so that no one attacks another.

Remember: a queen attacks other pieces on the same row, same column and same diagonals.

A possible solution:



Backtracking algorithm

Place first queen in top-left corner.

Assume now the situation with some non-attacking queens in place.

If < 8 queens on board, place a queen in the next row — if it attacks a queen, move along the row one square at a time. If no position is possible, move the queen in the immediately preceding row on one square. If no position is suitable, move the queen in the next row back, etc..

This algorithm actually returns the first solution after examining 114 of the 2057 possible states.

A Las Vegas approach – non-backtracking

- Place queens randomly on successive rows
- No attempt is made to relocate previous queens when there is no possibility left for the next queen
- The algorithm either ends successfully or fails if there is no square in which the next queen can be placed
- The process can be repeated if a failure is detected, and will consider a probably different placement

A Las Vegas approach – non-backtracking

The expected number of states explored can be calculated to be around 56, compared with 114 for the deterministic algorithm.

Further improvements can be made by combining random placement with some backtracking, first fixing some queens at random and then completing the arrangement via backtracking.

When the first two rows are occupied at random, then the expected number of states explored to find a solution becomes just 29.

References

- Brassard, G. & Bratley, P. Fundamentals of Algorithmics.
- Barringer H. Randomized algorithms - a brief introduction
- https://www.geeksforgeeks.org/fermat-method-of-primality-test/
- https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/
- https://www.baeldung.com/cs/fermat-primality-test