Extra - Probabilistic Algorithms

Luke Mathieson

May 24, 2019

Outline

- Deterministic Algorithms and Introducing Randomness
- Las Vegas and Monte Carlo Algorithms
- Examples:
    - Matrix Mutliplication Checking
    - String Matching
    - Primality Testing

Part I

Basics, Las Vegas and Monte Carlo Algorithms & Quick Sort

Deterministic Algorithms

So far we have seen *deterministic* algorithms:

- The execution of the algorithm depends only on the input.
- A given input produces exactly one output.
- The worst case complexity gives an upper bound for the number of steps a solution will be produced in.

A Trade Off

What if we are willing to trade *certainty* for *speed*?

This leads to the notion of *probabilistic* algorithms.

- The execution can depend on a random number ("coin toss").
- The output may not be correct.
- The output may not even be produced.
- But we might save a lot of time.

Deterministic Quick Sort

Recall the Quick Sort algorithm's properties:

- $O(n^2)$ worst case complexity, but $O(n \log n)$ on average.
- If the input happens to be bad - *i.e.* we pick a bad pivot value - then we take about n^2 steps.
- In most cases though the pivot splits the list roughly in half, and we get the $O(n \log n)$ behaviour.

Adding Some Randomness

Almost all lists are *typical* and Quick Sort performs well, only a few are *atypical*.

What can we do about the *atypical* lists?
We can randomise!

- There's a lot more *typical* lists than *atypical*.
- Thus if we randomly permute our list, we'll get a *typical* list with high probability. (It suffices to randomly choose the pivot)

Randomised Quick Sort

Now no input is *typical* or *atypical* anymore.

We get the $O(n \log n)$ behaviour with a probability that's very close to 1.

Note also that the algorithm always produces a correct output - it's just the exact time it takes that's a bit uncertain.

Probabilistic Algorithms

Probabilistic algorithms have two cases:

- The running time is always at most $T(n)$, but the answer may not always be correct (but is only wrong with low probability).
- The answer is always correct, but may not be produced in time $T(n)$ (but only fails to terminate with low probability).

Proposition

The second case can be converted to the first case.

Proof.

Run the algorithm for $T(n)$ steps, if it terminates, output the answer, otherwise output a random answer.

If the algorithm terminates, we know the answer is correct. If it doesn't, we get an answer that may be wrong, but this case only happens with low probability. $\square$

So we only have to consider algorithms that have definite running times, but may give incorrect answers with low probability.

Two Variants of Probabilistic Algorithms

There are two major classes of probabilistic algorithm:

Las Vegas Algorithms

Always give correct answers, but may sometimes give up.

Monte Carlo Algorithms

Always give an answer, but they might be lying.

Two Variants of Probabilistic Algorithms

Definition

A *Las Vegas* algorithm terminates after $T(n)$ steps and either outputs a correct answer or with low probability outputs a message indicating that no answer was found.

Definition

A *Monte Carlo* algorithm terminates after $T(n)$ steps and outputs an answer that is correct with high probability.

Back to Quick Sort

We can now give two randomised versions of Quick Sort:

Monte Carlo Quick Sort

- Run the sorting algorithm for $10n \log n$ steps.
- Output the list at this point.

With high probability the list has already be sorted.

Back to Quick Sort

Las Vegas Quick Sort

- Run the sorting algorithm for $10n \log n$ steps.
- Check if the list has been sorted.
- If it's sorted, output the answer, otherwise give up.

With high probability we get an answer, and if we do, it's definitely correct.
If the algorithm does give up, we can run it again and we have a good chance that the algorithm will succeed with a different permutation.

Part II

Matrix Multiplication Verification

Matrix Multiplication Verification I

Say we have 3 $n \times n$ matrices A, B and C, and we want to check whether

$$AB = C$$

We can compute this in $O(n^3)$ time (if we're clever, $O(n^{2.376})$ [CW81]).

With a bit of guessing, we can do this faster!

Matrix Multiplication Verification II

Freivalds [F77] made the following simple observation:

For any $n \times 1$ vector $\vec{x}$,

$$AB = C \Rightarrow AB\vec{x} = C\vec{x}$$

More importantly, if $AB \neq C$ then for a randomly chosen $\vec{x}$ the probability that $AB\vec{x} = C\vec{x}$ is small.

Matrix Multiplication Verification III

We make the speed saving because we can compute $AB\vec{x}$ as two *vector* multiplications, rather than one matrix multiplication ($AB\vec{x} = A\vec{y}$ where $\vec{y} = B\vec{x}$).

This can be done in $O(n^2)$ time.

Matrix Multiplication Verification IV

- The algorithm returns either `true` or `false`.
- If it says `false`, then it is correct.
    - $(AB = C \Rightarrow AB\vec{x} = C\vec{x}) \Rightarrow (AB\vec{x} \neq C\vec{x} \Rightarrow AB \neq C)$ (contrapositive)
- If it says `true`, it's *probably* correct.

This is a quadratic time Monte Carlo algorithm.

Matrix Multiplication Verification V

We can even improve this!

Lemma

Let $\vec{x}$ be a binary vector chosen uniformly at random.
If $AB \neq C$ then $AB\vec{x} = C\vec{x}$ with probability at most $\frac{1}{2}$.

Matrix Multiplication Verification VI

A 50% error chance doesn't sound so great.

There's a standard trick for this though:

- Run the algorithm k times.
- If any run outputs `false`, return `false`.
- If all runs return `true`, the probability that it guessed poorly k times in a row is at most $\frac{1}{2^k}$.

This is called *probability amplification*. So if we run it 10 times, the probability of saying `true` incorrectly is less than $\frac{1}{1000}$, 20 runs gives less than $\frac{1}{10^6}$.

Part III

String Matching

String Matching

Let Σ be a finite set of symbols called an alphabet.

Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$ be two strings over Σ (i.e. $A \in \Sigma^n$ and $B \in \Sigma^m$) with $m \leq n$.

The *String Matching Problem* is the problem of determining whether B is a substring of A (i.e. whether there is some k such that $b_1 = a_k, b_2 = a_{k+1}, \ldots, b_m = a_{k+m-1}$). ▸ Example

A Naïve Deterministic Solution

We can take a sliding window approach:

- Start with B lined up with the beginning of A.
- Match the characters one by one.
- If it matches, we've found it.
- If there's a mismatch, move B along one character and start again.
- Stop when we find it, or run out of A to check against.

There's $n - m + 1$ positions that B could start at, and we do m comparisons each time, so this takes about $O((n - m + 1)m)$ time.

A Slightly Better Approach

Replace strings of length m with a function $f : \Sigma^m \to \mathbb{N}$ of m variables - then we only need to do one comparison at each step. (It helps to treat Σ as a subset of $\mathbb{N}$).
We compute $\beta = f(b_1, \ldots, b_m)$ and then

$$\begin{array}{ccc} \alpha_1 & = & f(a_1, \ldots, a_m) \\ \alpha_2 & = & f(a_2, \ldots, a_{m+1}) \\ \vdots & \vdots & \vdots \\ \alpha_{n-m+1} & = & f(a_{n-m+1}, \ldots, a_n) \end{array}$$

and compare only the substrings with $\alpha_j = \beta$.

A Slightly Better Approach

The question is what function f to use:

- We could just take $f(x_1, \ldots, x_m) = \sum_{i=1}^{m} x_i$.
  - This is pretty easy to compute ▸ Example.
  - However too many strings have the same value under f - too many "collisions".
- We can pick better f and get a better result.

Rabin-Karp Algorithm I

We can use an algorithm that employs the same basic idea as *hashing* [KR87].

Hashing produces a “fingerprint” for each string. We can pick a hash function such that any two different strings will probably produce a different hash.

Rabin-Karp Algorithm II

Pick a large prime p and randomly select an integer $r \in [1, p-1]$.

Set

$$f(x_1, \ldots, x_m) = \sum_{i \in [m]} x_i r^{m-i} (\text{mod } p)$$

Looks complicated, but we can actually compute this efficiently.

How Many Collisions Do We Have Now?

How often do we have collisions, that is, how often

$$f(c_1, \ldots, c_m) = f(d_1, \ldots, d_m) \ ?$$

Answer: Not too often because the above collision implies that

$$e_1 r^{m-1} + e_2 r^{m-2} + \ldots + e_{m-1} r + e_m \equiv 0 \pmod{p}$$

where $e_i = c_i - d_i$.

Some Analysis

Lagrange Theorem: A polynomial of degree k has at most k roots.

As we have a polynomial of degree $m-1$, for each pair of m-tuples $(c_1, \ldots, c_m) \neq (d_1, \ldots, d_m)$ there are at most $m-1$ “bad” values of r for which a collision is possible.

So if B is not a substring of A, there are at most $(m-1)(n-m+1)$ values of r which might give the same value for f.

So if p is much bigger than $(m-1)(n-m+1)$ and $r \in [1, p-1]$ is selected “at random”, the probability of collision is very small.

Back to the Algorithm

- So we can just compute f for all length m substrings (having chosen p and r).
- If $f(B)$ is the same as any of these, we check if the corresponding substring is equal.
- Otherwise B is not a substring of A.

▸ Example

But...

How do we compute f efficiently?
We can adapt the sliding window approach: for overlapping substrings, we can reuse previous results - Dynamic Programming!

$$\begin{aligned}
f(a_{j+1}, \ldots, a_{j+m+1}) &= a_{j+1} r^{m-1} + \ldots + a_{j+m+1} \\
&= r(a_{j+1} r^{m-2} + \ldots + a_{j+m}) + a_{j+m+1} \\
&= r(f(a_j, \ldots, a_{j+m}) - a_j r^{m-1}) + a_{j+m+1}
\end{aligned}$$

What We Get

So we can compute all the f values for A in *linear* time!

With high probability we only have to do m actual comparisons.

This is a $O(n)$ Las Vegas algorithm!
It is still possible to get a lot of collisions, but if we start with too many, we can just guess a different r and start again.

Part IV

Primality Testing

Primality Testing

- Prime numbers play an important role in a lot of security related programs. (and string matching!)
- To make this work we need them to be quite large.
- So we need a decent way of finding large prime numbers.

Theorem (Prime Number Theorem)

Let $\pi(n)$ be the number of primes less than or equal to n. Then:

$$\pi(n) \approx \frac{n}{\log n}$$

As a consequence the probability that any integer of size ℓ is prime is $O(\frac{1}{\ell})$. So they're pretty scarce.

The Naïve Way

The basic idea is to take a large number N and test if it's prime.

We know that if N is prime, it has no factor between 2 and $\sqrt{N}$

We also know that if d is not a factor of N, neither is any multiple of d.

A Simple Approach: We could test 2, 3 and any number that is not a multiple of 6 up to $\sqrt{N}$. If no factor is found, then we know that N is prime.

Naïvety isn't cheap

This takes $O(\sqrt{N})$ time.

This is exponential in the size of N (i.e. $\log N$).
Interesting sizes of N for modern applications are at least 768 bits (the exact number isn't so important). This would give about 10^{116} steps to check primality. We'd need to be able to do about 10^{100} steps every second to compute this within the lifespan of the universe so far (about 10^{17} seconds).

Thinking Harder

So we want to a property that holds for primes, doesn't (seem to) hold for composites and is easy to check.

Theorem ((almost) Fermat's Little Theorem)

If p is prime them for every $a \in \mathbb{Z}$ that is coprime with p we have

$$a^{p-1} \equiv 1 (mod\ p)$$

This gives the "Fermat Primality Test":

1. Choose a random $a \in [2, N-1]$.
2. If $a^{N-1} \equiv 1 (\text{mod } N)$, return `Prime`.
3. Otherwise return `Composite`.

Fermat & Primality

Fast exponentiation makes this test very quick.
But:

Theorem (Euler)

Given coprime a and N,

$$a^{\phi(N-1)} \equiv 1 (mod\ N)$$

where ϕ is Euler's totient function.

So there are some values of a that will give us a false positive using the Fermat test (i.e. N can still be composite, just without sharing factors with a).

Example

If $N = 4087$, both 841 and 1905 would suggest N is prime by Fermat's test, but $N = 61 \times 67$.

How Often Can this Happen?

Quite a bit actually, 1105 has 768 false witnesses - more than two thirds of possible testing candidates.
In fact, for $N = 1105$, any coprime a will fail the test.

Fortunately numbers like 1105 are rare, most have very few false witnesses, so the Fermat Test is very reliable and fast. But we can do better...

Refining Fermat's Test

We can also check

$$a^{\frac{N-1}{2}} \mod N$$

If N is prime, then

$$X^2 - 1 \equiv 0 (\bmod N)$$

has only 2 solutions (± 1).

If N is odd but not prime, then $X^2 - 1 \equiv 0 (\bmod N)$ has at least 4 solutions.

If $a^{N-1} \equiv 1$, then $a^{\frac{N-1}{2}}$ is a root of 1 and maybe different to ± 1.

How useful is this?

Given our original example of $N = 4087$, there are 36 false witness, only 18 of which survive the second test.

For $N = 1105$, out of the 768 false witness, only 384 pass the new test!.

We can keep going, if we pass the second test, and $N - 1$ is a multiple of 4, we can try $a^{\frac{N-1}{4}}$ and see if we get something different from ± 1.

The Miller-Rabin Test

Introduced by Miller [M76] as a deterministic test relying on the unproven generalised Riemann hypothesis, modified by Rabin [R80] to be probabilistic.
Assume $N \geq 3$ is odd (otherwise we're done):

- There exists an $s \geq 1$ and odd t such that $N - 1 = 2^s t$.
- Let $B(N) \subseteq [1, N-1]$ be the set of integers a such that $a^t \equiv 1$ or there is an $i \in [0, s-1]$ such that $a^{2^i t} \equiv -1$ (all modulo N of course).
- Then by definition $a \in B(N)$ means that one of $a^t, a^{2t}, a^{2^2 t}, \ldots, a^{2^s t}$ must be 1.
- If it's not a^t then it's $a^{2^i t}$ for some i and therefore $a^{2^{i-1} t} \equiv -1$.

How big can $B(N)$ be?

We have:

Theorem

If N is prime then $B(N) = [1, N-1]$.
If N is composite then $|B(N)| < \frac{N}{4}$.

So we can use this to efficiently test with high probability that N is prime:

1. Repeat k times:
    1. Randomly pick $a \in [1, N-1]$.
    2. If $a \notin B(N)$, return `Composite`.
2. Otherwise return `Prime`.

Analysis

- If the algorithm returns `Composite`, it's correct.
- If the algorithm returns `Prime`, it's wrong with a probability of $\frac{1}{4^k}$.
- We can perform these computations in polynomial time.

For $k = 20$, this is an error probability of less than 9.09×10^{-13}.

This gives a polynomial time Monte Carlo algorithm!

Pushing the Envelope

Of course there have been improvements since 1980:

- Mersenne Primes are of a specific form, $2^n - 1$, and thus have extra properties we can test.
  - $2^{43112609} - 1$ is the largest know prime with 12978189 decimal digits.
- Adleman *et al.* [APR83] give an *almost* polynomial time algorithm that certifies a number is prime running in $O((\log n)^{C \log\log n})$ (it's a Las Vegas algorithm, if it answers, it's correct, but it may not answer).
- Agrawal *et al.* [AKS04] give a *deterministic*, *polynomial-time* algorithm for primality testing (AKS test).

Nonetheless, the Miller-Rabin test is still heavily used as the error is so low and it is significantly faster to run repeatedly than running the AKS test once.

Bibliography I

- Leonard M. Adleman, Carl Pomerance and Robert S. Rumely, "On distinguishing prime numbers from composite numbers", *Annals of Mathematics*, **117**(1):173206, 1983.
- Manindra Agrawal, Neeraj Kayal and Nitin Saxena, "PRIMES is in P", *Annals of Mathematics*,**160**(2):781793, 2004.
- D. Coppersmith and S. Winograd, "On the asymptotic complexity of matrix multiplication", In *Proc. SFCS*, pages 8290, 1981.
- Rusins Freivalds, "Probabilistic Machines Can Use Less Running Time", *IFIP Congress*, pp. 839842, 1977.
- Richard M. Karp and Michael O. Rabin,"Efficient randomized pattern-matching algorithms", *IBM Journal of Research and Development*, **31**, 1987.

Bibliography II

Gary L. Miller, “Riemann’s Hypothesis and Tests for Primality”, *Journal of Computer and System Sciences*, **13**(3):300317, 1976.

Michael O. Rabin, “Probabilistic algorithm for testing primality”, *Journal of Number Theory*, **12**(1):128138, 1980.

String Matching Example

Example

Given the alphabet $\Sigma = \{*, \&, \%\}$
and the string
$A = \& * \&\% * \% * * \& * \& * \%\% * \% * * \&\% * \& * * \%\& *$

If $B = \& * * \%$ then Yes!
If $B = \% * * \%$ then No!

Back

Using a Function to Match Strings

Example

$\Sigma = \{*, \&, \%\} \quad \longrightarrow \quad \Sigma = \{0, 1, 2\}$

$$A = \& * \&\% * \% * * \& * \& * \%\% * \% * * \&\% * \& * * \%\& *$$

$$A = 101202001010220200120100210$$

$$B = \& * * \% \quad \longrightarrow \quad B = 1002$$

$$\beta = 3$$

$$\begin{aligned} \alpha_j \quad = \quad & 4, 3, 5, 4, 2, 3, 1, 2, 2, 3, 5, 4, \\ & 6, 4, 2, 3, 3, 3, 4, 3, 1, 3, 3, 3 \end{aligned}$$

Example

$$A = 101202001010220200120100210 \quad B = 1002$$

$m = 4,\ n = 27$
Choose $p = 9973,\ r = 5347,$ Then

$$\beta = 1258$$

$$\begin{aligned}\alpha_j \quad = \quad & 6605, 8512, 6867, 3233, 5609, 2513, \\ & 5347, 7792, 6603, 7793, 1979, 6330, \\ & 8123, 3233, 5609, 2513, 5349, 8512, \\ & 6866, 7859, 7791, 1258, 722, 983\end{aligned}$$

▸ Back