10 Randomized algorithms

Summer Term 2011

Jan-Georg Smaus

Randomized algorithms

Overview

- Classes of randomized algorithms
  - Quicksort
  - Randomized Quicksort
  - Randomized primality test
  - Cryptography

1. Classes of randomized algorithms

- Las Vegas algorithms
  - Always correct; expected running time "probably fast"
  - Example: randomized Quicksort

- Monte Carlo algorithms (mostly correct):
  - Probably correct; guaranteed running time
  - Example: randomized primality test

2. Quicksort

**Algorithm:** Quicksort

**Input:** unsorted range \([l, r]\) in array \(A\)

**Output:** sorted range \([l, r]\) in array \(A\)

1. If \(r > l\)
   2. then choose pivot element \(p = A[r]\)
   3. \(m = \text{divide}(A, l, r)\)
   4. Divide \(A\) according to \(p\):
      \(A[l, \ldots, m - 1] \leq p \leq A[m + 1, \ldots, r]\)
   5. \(\text{Quicksort}(A, l, m - 1)\)
   6. \(\text{Quicksort}(A, m + 1, r)\)

The divide step
3. Randomized Quicksort

**Algorithm: Quicksort**

- **Input:** unsorted range \([l, r]\) in array \(A\)
- **Output:** sorted range \([l, r]\) in array \(A\)

1. if \(r > l\) then
2. randomly choose a pivot element \(p = A[i]\) in range \([l, r]\)
3. swap \(A[i]\) and \(A[r]\)
4. \(m = \text{divide}(A, l, r)\)
5. \(\ast\) Divide \(A\) according to \(p\):
6. \(\ast\) \([A[l], \ldots, A[m - 1]] \leq p \leq [A[m + 1], \ldots, A[r]]\)
7. \(\ast\) Quicksort\((A, l, m - 1)\)
8. \(\ast\) Quicksort\((A, m + 1, r)\)

---

**Analysis**

- Let \(S_i\) be the \(i\)-th smallest element
- \(S_1\) is chosen as pivot with probability \(1/n\):
  - Sub-problems of sizes \(0\) and \(n-1\)
    - \(\ast\)
    - \(\ast\)
    - \(\ast\)
    - \(\ast\)
- \(S_n\) is chosen as pivot with probability \(1/n\):
  - Sub-problems of sizes \(n-1\) and \(0\)
Analysis

Expected running time

\[ E(T(n)) = \frac{1}{n} \sum_{k=0}^{n-1} (E(T(k)) + E(T(n-k-1))) + \Theta(n) \]

\[ = \frac{2}{n} \sum_{k=0}^{n-1} E(T(k)) + \Theta(n) \]

(\Theta(n) because of the divide-procedure)

One can show that \( E(T(n)) = O(n \log n) \).

With high probability, bad inputs cannot spoil the performance.

4. Primality test

Definition:
An integer \( p > 2 \) is prime iff \( a | p \Rightarrow a = 1 \) or \( a = p \).

Algorithm: deterministic primality test (naive)
Input: integer \( n > 2 \)
Output: answer to the question: Is \( n \) prime?

if \( n = 2 \) then return true
if \( n \) even then return false
for \( i = 1 \) to \( \sqrt{\frac{n}{2}} \) do
if \( 2i + 1 \) divides \( n \) then return false
return true

Complexity: \( E(n) \) where \( n \) is the input, but the input size is just \( \log n \).

Primality test

Goal:
Randomized method
- Polynomial time complexity (in the length of the input)
- If answer is "not prime", then \( n \) is not prime
- If answer is "prime", then the probability that \( n \) is not prime is at most \( p > 0 \)

\( k \) iterations: probability that \( n \) is not prime is at most \( p^k \).

Randomized primality test

Theorem: (Fermat’s little theorem)
If \( p \) prime and \( 0 < a < p \), then \( a^{p-1} \mod p = 1 \). (I.e., \( p \) divides \( a^{p-1} – 1 \))

Definition:
\( n \) is pseudoprime to base \( a \), if \( n \) not prime and \( a^{n-1} \mod n = 1 \).

Example:
\( n = 11 \cdot 31 = 341 \), \( a = 2 \):
\( 2^{340} \mod 341 = 1 \).
\( n = 341 \), \( a = 3 \):
\( 3^{340} \mod 341 = 56 \neq 1 \)

Randomized primality test 1

1 Randomly choose \( a \in [2, n-1] \)
2 if \( a^{n-1} \mod n = 1 \) then \( n \) is possibly prime
3 else \( n \) is definitely not prime

Advantage: This only takes polynomial time.

Examples:
\( n = 17 \), \( a = 2 \):
\( 2^{16} = 65536 \), \( 65536 \mod 17 = 1 \).
\( n = 23 \), \( a = 2 \):
\( 2^{22} = 4194304 \), \( 4194304 \mod 23 = 1 \).
\( n = 341 \), \( a = 2 \):
\( 2^{340} \mod 341 = 1 \).
17 and 23 are indeed prime, 341 is not!

Prob(\( n \) is not prim, but \( a^{n-1} \mod n = 1 \)) ?

Carmichael numbers

Problem: Carmichael numbers

Definition: An integer \( n \) is called Carmichael number if
for all \( a \) with GCD(a, n) = 1.

Example:
Smallest Carmichael number: \( 561 = 3 \cdot 11 \cdot 17 \)
561 is pseudoprime to any base \( a \) that is not divisible by 3 or 11 or 17.
To show that 561 is not prime, we hence need a base \( a \) that is divisible by 3 or 11 or 17. This is still quite likely to find, but there are worse examples.
Randomized primality test 2

Theorem:
If \( p \) prime and \( 0 < a < p \), then the only solutions to the equation
\( a^2 \mod p = 1 \)
are \( a = 1 \) and \( a = p - 1 \).

Definition:
\( a \) is called non-trivial square root of 1 mod \( n \), if
\( a^2 \mod n = 1 \) and \( a \neq 1, n - 1 \).

Example:
\( n = 35 \)
\( 6^2 \mod 35 = 1 \)

Fast exponentiation

Idea:
During the computation of \( a^p \) (\( 0 < a < n \) randomly chosen), which we need
for the first primality test, test as a byproduct whether there is a non-
trivial square root 1 mod \( n \).

Method for the computation of \( a^p \):

Case 1: \( n \) is even
\( a^p = a^{p/2} * a^{p/2} \)

Case 2: \( n \) is odd
\( a^p = a^{p-1} * a^{p-1} * a \)

Fast exponentiation

Example:
\( a^6 = (a^{31})^2 \)
\( a^{31} = (a^{15})^2 * a \)
\( a^{15} = (a^7)^2 * a \)
\( a^7 = (a^3)^2 * a \)

Fast exponentiation + squares

boolean isProbablyPrime;

power(int a, int p, int n) {
/* computes \( a^p \mod n \) and checks during the
computation whether there is an \( x \) with
\( x^2 \mod n = 1 \) and \( x \neq 1, n-1 \) */
    if (p == 0) return 1;
    x = power(a, p/2, n);
    result = (x * x) % n;
    if (result == 1 && x != 1 && x != n - 1) isProbablyPrime = false;
    if (p % 2 == 1)
        result = (a * result) % n;
    return result;
}

Fast exponentiation + squares

Complexity: \( O(\log^2 n \log p) \)

Combined Procedure Miller-Rabin

primalityTest(int n) {
/* carries out the randomized primality test for
a randomly selected \( a \ )* /
    a = random(2, n -1);
    isProbablyPrime = true;
    result = power(a, n-1, n);
    if (result != 1 || !isProbablyPrime) return false;
    else return true;
}
**Combined Procedure Miller-Rabin**

**Theorem:**
If \( n \) is not prime, there are at most \( \frac{n-1}{4} \) integers \( 0 < a < n \) for which the algorithm \texttt{primalityTest} fails. Hence the probability of failure is \( \frac{n-1}{4} < \frac{n}{n} = \frac{1}{4} \).

If for a number \( n \) we do \( \log n \) tests we get a probability of \( \left(\frac{1}{4}\right)^{\log n} = \frac{1}{n^2} \) of failure. E.g. we might take \( n \) around \( 2^{100} \).

**Application: cryptosystems**

**Traditional encryption of messages with secret keys**

Disadvantages:
1. The key \( k \) has to be exchanged between A and B before the transmission of the message.
2. For messages between \( n \) parties \( n(n-1)/2 \) keys are required.

Advantage:
Encryption and decryption can be computed very efficiently.

**Desired properties of cryptographic systems**

- confidential transmission
- integrity of data
- authenticity of the sender
- reliable transmission

**Public-key cryptosystems**

Diffie and Hellman (1976)

Idea: Each participant A has two keys:
1. a public key \( P_A \) accessible to every other participant
2. a private (or: secret) key \( S_A \) only known to A.

**Encryption in a public-key cryptosystem**

A sends a message \( M \) to B.

\[ P_A(S_A(M)) = M \]

\( P_A \) is the inverse function of \( S_A \) and vice-versa.

3. \( S_A \) cannot be computed from \( P_A \) with reasonable effort.
Encryption in a public-key cryptosystem

1. A accesses B’s public key \( PB \) (from a public directory or directly from B).
2. A computes the encrypted message \( C = PB(M) \) and sends \( C \) to B.
3. After B has received message \( C \), B decrypts the message with his own private key \( SB : M = SB(C) \).

Generating a digital signature

1. A sends a digitally signed message \( M' \) to B:
2. A computes the digital signature \( \sigma \) for \( M' \) with her own private key: \( \sigma = SA(M') \)
3. A sends the pair \( (M', \sigma) \) to B.
4. After receiving \( (M', \sigma) \), B verifies the digital signature: \( PA(\sigma) = M' \)

\( \sigma \) can be verified by anybody via the public \( PA \).

RSA cryptosystems

R. Rivest, A. Shamir, L. Adleman

Generating the public and private keys:

1. Randomly select two primes \( p \) and \( q \) of similar size, each with \( l+1 \) bits \((l \geq 500)\).
2. Let \( n = p \cdot q \)
3. Let \( e \) be a (small) integer that does not divide \((p - 1)(q - 1)\).
4. Calculate \( d = e^{-1} \mod (p - 1)(q - 1) \)
   i.e.:
   \[ d \cdot e \equiv 1 \mod (p - 1)(q - 1) \]
5. Publish \( P = (e, n) \) as public key
6. Keep \( S = (d, p, q) \) as private key

Divide message (described in a binary string) in blocks of size \( 2^l \).
Interpret each block \( M \) as a binary number: \( 0 \leq M < 2^l \cdot l^2 \)

\[ P(M) = Me \mod n \]
\[ S(M) = M^d \mod n \]

P and S are inverses

We have (some basic math ...)

\[ M^e \equiv 1 \mod p \]
\[ M^e \equiv 1 \mod q \]
\[ M^e \equiv 1 \mod p \cdot q \]

and hence

\[ S(M) = (M^e)^d \mod n \]
\[ = M \mod n \]

The other direction is analogous.
S is hard to compute

This is unproven!

According to current knowledge, to compute $d$ from $e$ one would need to know $p$ and $q$.

Also according to current knowledge, computing $p$ and $q$ from $n$ is hard.

Even the fastest computers have never cracked RSA!

Summary

- We have seen two randomised algorithms:
  - Quicksort
  - Prime test
- We have also seen an application of big prime numbers: cryptography.