

Probabilistic Algorithms

- An algorithm that uses random numbers
 - sometimes better to chose randomly than to spend a lot of time to compute optimal choice
- May sometimes not have a guaranteed correct result
 - but: can, with several tries, solve any problem
 - otherwise: a heuristic algorithm
- Numerical probabilistic algorithms
 - confidence interval: answer is in a range with a certain probability
- Monte Carlo algorithms
 - give rarely wrong answer
- Las Vegas algorithms
 - rarely say: don't know

© 2005-06. 324-1-120

Three Kinds of Probabilistic Algorithms

- Algorithm to answer: “When did Columbus discover America?” — run it 3 times
- Numerical:
 - between 1490 and 1500
 - between 1485 and 1495
 - between 1491 and 1501
- Monte Carlo:
 - 1492, 1492, 357 BC
- Las Vegas:
 - 1492, sorry!, 1492

© 2005-06. 324-1-121

Probabilistic Algorithms

- Expected time
 - average time for a certain instance
- Average expected time
 - average of the expected times of all instances
- Worst-case expected time
 - worst of the expected times of all instances
- Random number generation
 - needed: function that can return uniformly distributed close-to-random numbers
 - typically, in the range between 0 and 1
 - pseudorandom generator: seed + function
 - example:
 - sequence: $x_i = f(x_{i-1})$ $x_0 = \text{seed}$
 - $f(x) = x^2 \bmod p \cdot q$ p, q large primes = 3 mod 4
 - take last bit of result

© 2005-06. 324-1-122

Prime Number Test

- Monte Carlo algorithm
- Needed for public-key cryptography
 - need large prime numbers
 - based on the fact that it's very hard to find factors of large numbers
- Guaranteed algorithm
 - try numbers $< \sqrt{n}$ as factors
 - runtime:
 - for a 100-digit number: $\sqrt{10^{100}} = 10^{50} = \text{forever}$
 - but a Monte Carlo algorithm can do it!
 - for a 100-digit-number: $\lg 10^{100} = 100 * \lg 10 = 332. \dots$
 - with arbitrary error probability

© 2005-06. 324-1-123

Monte Carlo Prime Number Test

- Fermat's Little Theorem:
 - n is prime
 - then: $a^{n-1} \bmod n = 1$ $1 < a < n-1$
 - can use this to test if a number is prime!
- But who says this does not happen if n is not prime??
 - turns out it happens very rarely (3% if $n < 1000$)
 - at least if a is chosen randomly
 - with some numbers it fails most of the time
 - smallest: $561 = 3 \cdot 11 \cdot 17$
- Algorithm can be improved so that there are no numbers where it fails often
 - hence can be repeated to increase reliability

© 2005-06. 324-1-124

Modulo Exponentiation

- Remember: divide-and-conquer exponentiation
 - runtime?
 - problem: overflow
 - modulo: similar, but no overflow!!
 - theorem: $(x \bmod z)^y = x^y \bmod z$

```
// Return: (basisexponent) mod modulo
int expoMod( int base, int exponent, int modulus ) {
    if( exponent == 1 ) {
        return base % modulus;
    }
    if( exponent % 2 == 0 ) {
        int i = expoMod( base, exponent / 2 );
        return ( i * i ) % modulus;
    }
    return ( base * expoMod( base, exponent - 1 ) )
        % modulus;
}
```

© 2005-06. 324-1-125

Monte Carlo Prime Number Test

```
boolean isLikelyPrime( int n ) {  
    int a = randomInteger( 1, n-1 );  
    return expoMod( a, n-1, n ) == 1;  
}
```

- Runtime: $O(\log n)$
 - error probability depends on n
- Miller-Rabin prime test
 - improved version
 - failure probability for one call: 1%
 - error probability does not depend on n
 - can be decreased by repeat calls!
 - runtime: $O(\log^3 n)$

© 2005-06. 324-1-126

RSA Cryptography

- RSA = Rivest, Shamir, Adleman
 - was patented until recently
 - a public-key cryptography system
 - check out GnuPG (freeware implementation)
- Two people A and B want to communicate
 - each needs: a public key
 - each needs: a private key
 - compare to single-key cryptography
 - requires no prior coordination
- Process: send message from A to B
 - `encode(message, privateKey(A), publicKey(B))`
 - `decode(message, publicKey(A), privateKey(B))`

© 2005-06. 324-1-127

RSA Cryptography

- Keys
 - p, q: large, random prime numbers
 - $z = p \cdot q$; $z > \text{message}$
 - impossible to calculate prime factors of z
 - n: random integer $< z$
 - has no common factors with $(p-1)(q-1)$
 - public key: (n, z)
 - private key: $s < z$
 - $n \cdot s \bmod ((p-1)(q-1)) = 1$
 - s is unique number
 - if it does not exist, choose new n
- Encode
 - $\text{encryptedMessage} = \text{message}^n \bmod z$
- Decode
 - $\text{message} = \text{expoMod}(\text{encryptedMessage}, s, z)$
 - $\text{encryptedMessage}^s \bmod z = (\text{message}^n \bmod z)^s \bmod z = (\text{message}^n)^s \bmod z = \text{message}^{n \cdot s} \bmod z = \text{message}$

© 2005-06. 324-1-128

Randomized Caching

- Cache
 - a subset of data needed is stored locally
 - for faster access
 - example: Web browser cache
 - miss: when a requested item is not in the cache
- Problem
 - reducing chance of misses in a cache
- Idea
 - mark recently used items
 - in phases

© 2005-06. 324-1-129

Randomized Caching

- Need to select item to evict from cache
 - how???
 - efficiency depends on selection right item
- Least Recently Used (LRU)
 - common, simple strategy
 - problem:
 - cache size = k
 - if we have loop that requests $k+1$ items each iteration
 - then each request is a miss!
 - number of misses: $O(k)$
- Randomized marking algorithm
 - pick random unmarked item to evict
 - expected number of misses: $O(\log k)$

© 2005-06. 324-1-130

Randomized Caching

```
class Cache { //parts of class removed

    Source source; //where the data come from

    Item request( Key key ) {
        mark( key ); //mark as recently used
        if(contains( key ) ) {
            return get( key );
        } else {
            if( allItemsAreMarked() ) {
                unmarkAllItems();
                return request( key );
            } else {
                evictARandomUnmarkedItem();
                Item i = source. getItem( key );
                put( key, i );
                return i;
            }
        }
    }
}
```

© 2005-06. 324-1-131