

## 10.1 Approximation Algorithms

- An **algorithm** is TM decider.
- Formal languages represent **decision problems**, most of which can be solved by Turing machines.
- Some problems look for minimum or maximum values, called **optimization problems**, most of which can be computed by Turing machines.
- Finding an optimal value may be too expensive; a nearly optimal may be good enough and easier to find. Such Turing machines are called **approximate algorithms**.

## 22c:131 Limits of Computation

Hantao Zhang

<http://www.cs.uiowa.edu/~hzhang/c131>

The University of Iowa  
Department of Computer Science

Ch.10 ADVANCED TOPICS

## Proof of Theorem 10.1

**Theorem 10.1** There is a polynomial time algorithm that produces a vertex cover of  $G$  that is no more than twice as large as a smallest vertex cover.

$A =$  "On input  $\langle G \rangle$ , where  $G$  is an undirected graph:

1. Repeat the following until all edges in  $G$  have a marked endpoint:
2. Find an edge in  $G$  that have no marked endpoints.
3. Mark the two endpoints of that edge.
4. Output all marked nodes.

Algorithm  $A$  satisfies the conditions in Theorem 10.1.

## Vertex Cover Problem

- **Decision version:**  $Vertex-Cover = \{ \langle G, k \rangle \mid G = (V, E), \text{ an undirected graph, } k \text{ an integer, } G \text{ has a vertex cover whose size is no more than } k \}$ .
- **Optimization version:** Given  $G$ , find a vertex cover of minimum size in  $G$ .
- **Theorem 7.44**  $Vertex-Cover$  is NP-complete.
- **Theorem 10.1** There is a polynomial time algorithm that produces a vertex cover of  $G$  that is no more than twice as large as a smallest vertex cover.

## Max-Cut

- **Decision version:**  $Max-Cut = \{ \langle G, k \rangle \mid G = (V, E), \text{ an undirected graph, } k \text{ an integer, } G \text{ has a cut } (S, T) \text{ such that } |E \cap (S \times T)| \geq k \}$ .
- **Optimization version:** Given  $G$ , find a cut whose cut edges are maximum in  $G$ .
- **Theorem** *Max-Cut* is NP-complete.
- **Note:** The Min-Cut problem is in  $P$ .

## $k$ -Optimal

- **Minimization problem:** An approximation algorithm is  $k$ -optimal if it always finds a solution that is not more than  $k$  times the size of the optimal.
- **Maximization problem:** An approximation algorithm is  $k$ -optimal if it always finds a solution that is at least  $1/k$  times the size of the optimal.

## 10.2 Probabilistic Algorithms

- A **probabilistic algorithm** is an algorithm designed to use the outcome of a random process.
- **Definition 10.3** A **probabilistic Turing machine**  $M$  is a type of nondeterministic Turing machine in which each nondeterministic step is called a **coin-flip step** and has two legal next moves. We assign probability to each branch  $b$  of  $M$ 's computation on input  $w$  as follows. Define the probability of branch  $b$  to be

$$\Pr[b] = 2^{-k},$$

where  $k$  is the number of coin-flip steps that occur on branch  $b$ .

Define the probability that  $M$  accepts  $w$  to be

$$\Pr[M \text{ accepts } w] = \sum \Pr[b], b \text{ is an accepting branch.}$$

## Theorem 10.2

**Theorem 10.2** There is a polynomial time algorithm, 2-optimal approximate algorithm for *Max-Cut*.

$B =$  "On input  $\langle G \rangle$ , where  $G = (V, E)$  is an undirected graph:

1. Let  $S = \emptyset, T = V$ .
2. If moving a single node, either from  $S$  to  $T$  or from  $T$  to  $S$ , increase the size of the cut, make that move and repeat this stage.
3. If no such node exists, output  $(S, T)$ .

Algorithm  $B$  satisfies the conditions in Theorem 10.2.

## Lemma 10.5

Let  $\epsilon$  be a fixed constant strictly between 0 and 0.5. Then for any polynomial  $p(n)$ , a probabilistic polynomial time Turing machine  $M_1$  that operates with error probability  $\epsilon$  has an equivalent probabilistic polynomial time Turing machine  $M_2$  that operates with an error probability of  $2^{-p(n)}$ .

## The Class BPP

- $\Pr[M \text{ accepts } w] = \sum \Pr[b]$ ,  $b$  is an accepting branch.
- $\Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w]$ .
- For  $0 \leq \epsilon < 0.5$ , we say  $M$  **recognizes language  $A$  with error probability  $\epsilon$**  if
  1.  $w \in A$  implies  $\Pr[M \text{ accepts } w] \geq 1 - \epsilon$ , and
  2.  $w \notin A$  implies  $\Pr[M \text{ rejects } w] \geq 1 - \epsilon$ .
- **Definition 10.4 BPP** is the class of languages that are recognized by probabilistic polynomial time Turing machines with an error probability of  $1/3$ .

## Some Lemmas

Suppose  $0 \leq \delta \leq \epsilon < 1/2$ .

1.  $\epsilon(1 - \epsilon) \leq 1/4$
2.  $\delta(1 - \delta) \leq \epsilon(1 - \epsilon)$
3. If  $w + c = 2k$ ,  $w > c$ , then  $\epsilon^w(1 - \epsilon)^c \leq \epsilon^k(1 - \epsilon)^k$
4.  $\sum_{i=0}^n \binom{n}{i} = 2^n$

## Lemma 10.5

Let  $\epsilon$  be a fixed constant strictly between 0 and 0.5. Then for any polynomial  $p(n)$ , a probabilistic polynomial time Turing machine  $M_1$  that operates with error probability  $\epsilon$  has an equivalent probabilistic polynomial time Turing machine  $M_2$  that operates with an error probability of  $2^{-p(n)}$ .

**Proof:** Given TM  $M_1$  with error probability  $\epsilon < 1/2$  and a polynomial  $p(n)$ , we construct a TM  $M_2$  that recognizes  $L(M_1)$  with an error probability of  $2^{-p(n)}$ .

$M_2 =$  “On input  $x$ :

1. Calculate  $k = p(n)/\alpha$ , where  $\alpha = \log_2(4\epsilon(1 - \epsilon))$ .
2. Run  $2k$  independent simulations of  $M_1$  on input  $x$ .
3. If most runs of  $M_1$  accept, then *accept*; otherwise, *reject*.”

## Proof of Lemma 10.5

For

$$\begin{aligned} & \Pr[ M_2 \text{ outputs incorrectly on input } x] \\ & \leq (4\epsilon(1 - \epsilon))^k \\ & \leq 2^{-p(n)} \end{aligned}$$

We choose

$$k \geq p(n) / (-\log_2(4\epsilon(1 - \epsilon)))$$

## Proof of Lemma 10.5

$$\begin{aligned} & \Pr[ M_2 \text{ outputs incorrectly on input } x] \\ = & \Pr[ M_1 \text{ outputs on } x \text{ incorrectly } w \text{ times and correctly } c \text{ times}] \\ & \text{where } w + c = 2k, w > c \\ = & \sum_{c=0}^{k-1} \binom{2k}{c} (\epsilon_x)^w (1 - \epsilon_x)^c \\ & \text{where } \epsilon_x = \Pr[ M_1 \text{ outputs incorrectly on } x] \\ \leq & \sum_{c=0}^{k-1} \binom{2k}{c} (\epsilon)^w (1 - \epsilon)^c \\ \leq & \sum_{c=0}^{k-1} \binom{2k}{c} (\epsilon)^k (1 - \epsilon)^k \\ = & (\sum_{c=0}^{k-1} \binom{2k}{c}) (\epsilon)^k (1 - \epsilon)^k \\ \leq & (\sum_{c=0}^{2k} \binom{2k}{c}) (\epsilon)^k (1 - \epsilon)^k \\ = & (2^{2k}) (\epsilon)^k (1 - \epsilon)^k \\ = & (4\epsilon(1 - \epsilon))^k \end{aligned}$$

## Fermat Test

- **Fermat's Little Theorem:** If  $p$  is prime, and  $a \in \mathcal{Z}_p^+ = \{1, 2, \dots, p-1\}$ , then  $a^{p-1} \equiv 1 \pmod{p}$ .
- **Fermat Test:** If  $a^{n-1} \not\equiv 1 \pmod{n}$ , then  $n$  is not a prime.
- **PSEUDOPRIME** = "On input  $n$ :
  1. Select  $a_1, a_2, \dots, a_k$  randomly in  $\mathcal{Z}_n^+$ .
  2. Compute  $a_i^{n-1} \pmod{n}$  for each  $i$ .
  3. If one of the computed values in step 2 is not 1, *reject*; otherwise, *accept*."
- **Example:**  $n = 6$ ,  $a_1 = 2$ ,  $2^{6-1} = 32$ , and  $32 \pmod{6} = 2$ .

## Example: Primality Testing

- Testing if a number is prime has been extensively studied.
- There exists a complex polynomial time algorithm.
- We present a practical approximation algorithm for this problem.
- Some concepts:
  - Two numbers  $x$  and  $y$  are **equivalent modulo**  $p$ ,  $x \equiv y \pmod{p}$ , if  $x \pmod{p} = y \pmod{p}$ .
  - All the remainders by  $p$  is  $\mathcal{Z}_p = \{0, 1, 2, \dots, p-1\}$ .
  - **Fermat's Little Theorem:** If  $p$  is prime, and  $a \in \mathcal{Z}_p^+ = \{1, 2, \dots, p-1\}$ , then  $a^{p-1} \equiv 1 \pmod{p}$ .

## Korselt's Criterion

- **Theorem** (Korselt 1899) A positive composite integer  $n$  is a Carmichael number if and only if  $n$  is square-free, and for all prime divisors  $p$  of  $n$ ,  $p - 1$  divides  $n - 1$  (i.e.,  $p - 1 \mid n - 1$ ).
- **Ex:**  $561 = 3 \cdot 11 \cdot 17$ ,  $2 \mid 560$ ,  $10 \mid 560$ , and  $16 \mid 560$ .
- **Theorem** If  $n$  is neither prime nor Carmichael number, then there exist at least  $(n - 1)/2$   $a$ 's in  $Z_n^+$  such that  $a^{n-1} \not\equiv 1 \pmod{n}$ .
- **PSEUDOPRIME's** error probability is  $2^{-k}$  for pseudo-primes (primes + Carmichael numbers).

## Carmichael Numbers

- Some composite numbers  $n$  pass Fermat Test for any  $a$  where  $\gcd(n, a) = 1$ .
- Such numbers are called **Carmichael numbers**.
- The first three Carmichael numbers are 561, 1105, 1729.
- Let  $C(n)$  denote the number of Carmichael numbers less than or equal to  $n$ .

$i$	3	4	5	6	7	8	9	10	11
$C(10^i)$	1	7	16	43	105	255	646	1547	3605

## Square-root Test

- A number  $x \in Z_n^+$  is said to be a **square root of 1 modulo  $n$**  if  $x^2 \equiv 1 \pmod{n}$ .
- **Lemma:** For any prime  $p$ , it has only two square roots of 1: 1 and  $p - 1$ .
- **Ex1:** Four square roots of 1 modulo 21 are 1, 20, 8, 13.
- **Ex2:** One square root of 1 modulo Carmichael number 561 is 67:  $67^2 = 4489 = 8 \cdot 561 + 1$ .
- **Square-root Test:** If  $x$  is a square root of one modulo  $n$ ,  $x \not\equiv 1$  and  $x \not\equiv n - 1$ , then  $n$  is composite.

## Fermat Test on Carmichael Numbers

**Ex:**  $561 = 3 \cdot 11 \cdot 17$ . If

- $\gcd(561, a) = 3 \cdot 11$ , then  $a^{560} = 528 \pmod{561}$ ;
- $\gcd(561, a) = 3 \cdot 17$ , then  $a^{560} = 408 \pmod{561}$ ;
- $\gcd(561, a) = 11 \cdot 17$ , then  $a^{560} = 187 \pmod{561}$ ;
- $\gcd(561, a) = 3$ , then  $a^{560} = 375 \pmod{561}$ ;
- $\gcd(561, a) = 11$ , then  $a^{560} = 154 \pmod{561}$ ;
- $\gcd(561, a) = 17$ , then  $a^{560} = 34 \pmod{561}$ ;
- for other  $a$ ,  $a^{560} = 1 \pmod{561}$  (320 cases).

## Square-root Test

- **Square-root Test:** If  $x$  is a square root of one modulo  $n$ ,  $x \neq 1$  and  $x \neq n - 1$ , then  $n$  is composite.
- **SQUAREROOTTEST** = “On  $\langle a, n \rangle$ , where  $a^{n-1} \equiv 1 \pmod n$ ,
  1. Compute  $h$  and  $s$  such that  $n - 1 = 2^h s$ , where  $h \geq 1$  and  $s$  is odd.
  2. Let  $x_0 = a^s \pmod n$ . If  $x_0 = 1$  then *accept*.
  3. For  $j$  from 1 to  $h$  repeat the following.
  4.  $x_j = x_{j-1}^2 \pmod n$
  5. If  $x_j = n - 1$  then *accept*.
  6. If  $x_j = 1$  then *reject*.
  7. If no rejects in step 5, *accept*”.

What's its time complexity?

## Square-root Test

- **Square-root Test:** If  $x$  is a square root of one modulo  $n$ ,  $x \neq 1$  and  $x \neq n - 1$ , then  $n$  is composite.
- **SQUAREROOTTEST** = “On  $\langle a, n \rangle$ , where  $a^{n-1} \equiv 1 \pmod n$ ,
  1. Compute  $h$  and  $s$  such that  $n - 1 = 2^h s$ , where  $h \geq 1$  and  $s$  is odd.
  2. Let  $x_0 = a^s \pmod n$ .
  3. For  $j$  from 1 to  $h$  repeat the following.
  4.  $x_j = x_{j-1}^2 \pmod n$
  5. If  $x_j = 1$  and  $x_{j-1} \neq 1$  and  $x_{j-1} \neq n - 1$ , *reject*.
  6. If no rejects in step 5, *accept*”.

What's its time complexity?

## Square-root Test: $n = 105$

- There are 16 values in  $Z_{105}^+$  such that  $a^{104} \equiv 1 \pmod{105}$ : 1, 8, 13, 22, 29, 34, 41, 43, 62, 64, 71, 76, 83, 92, 97, 104.
- Of them, eight are square roots of 1 modulo 105: 1, 29, 34, 41, 64, 71, 76, 104.
- $104 = 2^3 \cdot 13$ , hence  $h = 3$  and  $s = 13$ .
- For  $a = 8$ ,  $x_0 = 8^{13} \equiv 8 \pmod{105}$ .  
 $x_1 = x_0^2 = 8^2 = 64 \pmod{105}$ .  
 $x_2 = x_1^2 = 64^2 \equiv 1 \pmod{105}$ .
- **Conclusion:** For 104 numbers in  $Z_{105}^+$ , Fermat Test will succeed 88 times and Square Root Test will succeed 14 times. The success rate is 102/104.

## Square-root Test: $n = 15$

- There are four values in  $Z_{15}^+$  such that  $a^{14} \equiv 1 \pmod{15}$ : 1, 14, 4, 11; they are square roots of 1 modulo 15:
- $14 = 2 \cdot 7$ , hence  $h = 1$  and  $s = 7$ .
- For  $a = 4$ ,  $x_0 = 4^7 \equiv 4 \pmod{15}$ .  
 $x_1 = x_0^2 = 4^2 = 16 \equiv 1 \pmod{15}$ .
- For  $a = 11$ ,  $x_0 = 11^7 \equiv 11 \pmod{15}$ .  
 $x_1 = x_0^2 = 11^2 \equiv 1 \pmod{15}$ .
- **Conclusion:** For 14 numbers in  $Z_{15}^+$ , Fermat Test will succeed 10 times and Square Root Test will succeed 2 times. The success rate is 12/14.

## Square-root Test: $n = 221$

- There are 16 values in  $Z_{221}^+$  such that  $a^{220} \equiv 1 \pmod{221}$ : 1, 18, 21, 38, 47, 64, 86, 103, 118, 135, 157, 174, 183, 200, 203, 220.
- Of them, four are square roots of 1 modulo 221: 1, 103, 118, 220.
- $220 = 2^2 \cdot 55$ , hence  $h = 2$  and  $s = 55$ .
- For  $a = 21$ ,  $x_0 = 21^{55} \equiv 200 \pmod{221}$ .  
 $x_1 = x_0^2 = 200^2 \equiv 220 \pmod{221}$ .  
 $x_2 = x_1^2 = 220^2 \equiv 1 \pmod{221}$ .  
Since  $x_1 = 221 - 1$ , the square-root test failed. There are six such cases: 1, 21, 47, 174, 200, 220.
- **Conclusion:** For 220 numbers in  $Z_{221}^+$ , Fermat Test will succeed 204 times and Square Root Test will succeed 10 times. The success rate is  $214/220$ .

## Another Example: $n = 561$

- There are 320 values in  $Z_{561}^+$  such that  $a^{560} \equiv 1 \pmod{561}$ .
- Of them, eight are square roots of 1 modulo 105.
- $560 = 2^4 \cdot 35$ , hence  $h = 4$  and  $s = 35$ .
- For  $a = 2$ ,  $x_0 = 2^{35} \equiv 263 \pmod{561}$ .  
 $x_1 = x_0^2 = 263^2 \equiv 166 \pmod{561}$ .  
 $x_2 = x_1^2 = 166^2 \equiv 67 \pmod{561}$ .  
 $x_3 = x_2^2 = 67^2 \equiv 1 \pmod{561}$ .
- For  $a = 5$ ,  $x_0 = 5^{35} \equiv 23 \pmod{561}$ .  
 $x_1 = x_0^2 = 23^2 \equiv 529 \pmod{561}$ .  
 $x_2 = x_1^2 = 529^2 \equiv 463 \pmod{561}$ .  
 $x_3 = x_2^2 = 463^2 \equiv 67 \pmod{561}$ .  
 $x_4 = x_3^2 = 67^2 \equiv 1 \pmod{561}$ .
- **Conclusion:** For 560 numbers in  $Z_{561}^+$ , Fermat Test will succeed 240 times and Square Root Test will succeed 318 times. The success rate is  $558/560$ .

## Combine Fermat and Square-root Tests

### Miller-Rabin's Algorithm

*PRIME* = "On input  $n$ , where  $n$  is odd:

1. Select  $a_1, a_2, \dots, a_k$  randomly in  $Z_n^+$ .
2. For each  $i$  from 1 to  $k$ :
3. Compute  $a_i^{n-1} \pmod{n}$  and *reject* if the value is not 1.
4. Call *SQUAREROOTTEST*( $a_i, n$ ); if it rejects, *reject*.
5. All tests have passed at this point, so *accept*."

**Theorem** If  $k = 1$ , the error probability of *PRIME* is at most  $\frac{1}{4}$ .  
In general, it is  $4^{-k}$ .

Let *PRIMES* = {  $n$  |  $n$  is a prime number in binary }.

**Theorem 10.9** *PRIMES*  $\in$  BPP.

## Square-root Test

- Let  $A = \{a \in Z_n^+ \mid a^{n-1} \equiv 1 \pmod{n}\}$  and  $m = |A|$ .
- The best theoretic result says that *SQUAREROOTTEST* will succeed at least  $m/2$  times for numbers in  $A$ .
- The experimental results show that *SQUAREROOTTEST* will succeed  $m - 2$  times in many cases.

## 10.6 Cryptography

- Cryptography is a much older field area than computer science.
- Modern cryptography tools use computers and many areas of computer science need cryptographic protection.
  - Passwords protection - war on hikers
  - Secure networks - online business and banking
  - Digital signatures
  - E-voting
- Complexity theory provides a way to gain evidence for a secret code's security, because we know in complexity theory which functions are easy and which are hard to compute.

## Definition 10.10

- **RP** is the class of languages that are recognized by probabilistic polynomial time Turing machines where inputs in the language are accepted with a probability of at least  $\frac{1}{2}$  and inputs not in the language are rejected with a probability of 1.
- Let  $COMPOSITES = \{ n \mid n \text{ is a composite number in binary} \}$ .  
Then  $COMPOSITES \in RP$ .

## One-Way Functions (Formal)

- A function  $f : \Sigma^* \rightarrow \Sigma^*$  is **length-preserving** if  $|w| = |f(w)|$ .
- A one-to-one length-preserving function is called a **permutation**.
- Suppose a probabilistic Turing machine  $M$  computes a **probabilistic function**  $f_M : \Sigma^* \rightarrow \Sigma^*$ , or simply  $f_M : \Sigma^* \rightarrow \Sigma^*$ . Define the probability that  $M(w) = x$  to be

$$\Pr[M(w) = x] = \sum \Pr[b], b \text{ is a branch where } M \text{ halts in accept state with } x \text{ on the tape for input } w.$$

- Since  $M$  may sometimes fail to accept an input  $w$ , so

$$\sum_{x \in \Sigma^*} \Pr[M(w) = x] \leq 1.$$

## One-Way Functions

- Informally, a **one-way** function is a function  $f$  such that  $f(x)$  is easy to compute but its inverse is hard to compute.
- A function is **one-to-one** if  $f(x) \neq f(y)$  whenever  $x \neq y$ .
- Given a one-to-one function  $f$ , the **inverse** of  $f$  is the function  $g = f^{-1}$  such that  $g(y) = x$  if  $f(x) = y$ .
- The concept of **inverse** can be extended to non-one-to-one functions: the **inverse** of  $f$  is the function  $g$  such that  $g(y) = z$  if  $f(x) = y = f(z)$  for some  $z$ .

## Do We Have One-Way Functions?

**Theorem** If there exists a one-way function  $f$  satisfying Def. 10.45, then  $\mathbf{P} \neq \mathbf{NP}$ .

**Proof:**

- For every length-preserving  $f$ , we have an inverse of  $f$  which can be computed by a TM  $R$  in  $\mathbf{NP}$ .
- $R =$  "On input  $\langle M_f, x \rangle$ 
  1. Nondeterministically choose a string  $w$ ,  $|w| = |x|$ .
  2. Call  $M_f$  on  $w$ . If  $M_f(w) = x$ ,  $R$  accepts with output  $w$ ; otherwise reject.
- If  $\mathbf{P} = \mathbf{NP}$ , then there exists a deterministic TM  $E$  equivalent  $R$ ,  $E \in \mathbf{P}$  and  $\Pr_{E,w}[E(f(w)) = w] = 1$ . Condition 2 is violated and we cannot have any one-way functions.

**Note:** If  $\mathbf{P} \neq \mathbf{NP}$ , we still do not know if there exist one-way functions. Limits of Computation – p.34/41

## Definition 10.45

- A **one-way permutation** is a permutation  $f$  with the following two properties.
  1. It is computable in polynomial time.
  2. For every probabilistic polynomial time TM  $M$ , every  $k$ , and sufficiently large  $n$ , if we pick a random  $w$  of length  $n$  and run  $M$  on input  $w$ ,

$$\Pr_{M,w}[M(f(w)) = w] \leq n^{-k}.$$

- A **one-way function** is a length-preserving  $f$  with the following two properties.
  1. It is computable in polynomial time.
  2. For every probabilistic polynomial time TM  $M$ , every  $k$ , and sufficiently large  $n$ , if we pick a random  $w$  of length  $n$  and run  $M$  on input  $w$ ,

$$\Pr_{M,w}[M(f(w)) = y \text{ where } f(y) = f(w)] \leq n^{-k}.$$
Limits of Computation – p.33/41

## Uses of One-Way Functions

- Easy to encode but hard to decode.
- Useful for secure password systems.
- Difficult to use alone for general cryptosystems (requires decode).
- Another kind of one-way functions are called *trapdoor functions* which can easily compute inverses (decoding) with special information (i.e., keys).

## Candidates for One-Way Functions

- The multiplication function *mult*.

Let  $\Sigma = \{0, 1\}$  and for any  $w \in \Sigma^*$  let

$$\text{mult}(w) = w_1 \times w_2,$$

where  $w = w_1 w_2$  such that either  $|w_1| = |w_2|$  or  $|w_1| = |w_2| + 1$ . Each binary string is treated as binary numbers. Leading zeroes are padded to  $\text{mult}(w)$  if  $|\text{mult}(w)| < |w|$ .

E.g.,  $w = 1100101001$ ,  $w_1 = 11001$  and  $w_2 = 01001$ .  
 $w_1 \times w_2 = 11100001$  and  $\text{mult}(w) = 0011100001$ .

# Digital Signatures

- Public-Key Cryptosystems make it easy.
- Suppose Bob wants to send his signature to Alice.
  1. Let  $M$  be the document to be signed. Bob applies his own private key,  $p_{Bob}$ , to  $M$ :  $C = f(p_{Bob}, M)$ .
  2. Bob sends both  $M$  and  $C$  to Alice.
  3. Alice finds Bob's public key,  $k_{Bob}$ , and decrypt  $C$  using  $k_{Bob}$ :  $X = g(k_{Bob}, C)$ . If  $X = M$ , then Alice knew it's Bob's signature.

# Cryptosystems

- **Private-Key Cryptosystems:** Two parties who want to communicate over an insecure channel share a private key for encoding and decoding.  
**Problem:** Hard to change keys
- **Public-Key Cryptosystems:** Each party has two keys: one is in the public domain and the other is kept private.
- Suppose Alice wants to send a message  $M$  to Bob.
  1. Alice finds Bob's public key,  $k_{Bob}$ , and compute  $C = f(k_{Bob}, M)$  using  $k_{Bob}$ .
  2. Alice sends the encrypted message  $C$  to Bob.
  3. Bob uses his private key,  $p_{Bob}$  to decrypt the message:  $M = g(p_{Bob}, C)$ .

# Trapdoor Functions

- A **trapdoor function**  $f : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$  is a length-preserving indexing function that has an auxiliary probabilistic polynomial time TM  $G$  and an auxiliary function  $h : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ . The trio  $f, G$ , and  $h$  satisfy the following three conditions.

1.  $f$  and  $h$  are computable in polynomial time.
2. For every probabilistic polynomial time TM  $E$ , every  $k$ , and sufficiently large  $n$ , if we pick a random output  $\langle i, t \rangle$  of  $G$  on  $1^n$  and a random  $w$  of length  $n$  and run  $E$  on input  $i$  and  $w$ , then

$$\Pr_{E,w}[E(i, f_i(w)) = y, \text{ where } f_i(y) = f_i(w)] \leq n^{-k}.$$

3. For every  $n$ , every  $w$  of length  $n$ , and every output  $\langle i, t \rangle$  of  $G$  that occurs with nonzero probability for some input to  $G$

$$h(t, f_i(w)) = y, \text{ where } f_i(y) = f_i(w).$$

# Indexing Functions

- If we have a family of functions  $\{f_i\}$  for  $i \in \Sigma^*$ , we represent them by the single function  $f : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ , where  $f(i, w) = f_i(w)$  for any  $i$  and  $w$ . This  $f$  is called an **indexing function**.  $f$  is said to be length-preserving if every  $f_i$  is length-preserving.

## Use of Trapdoor Functions

- TM  $G$ , functions  $f$  and  $h$  are cryptosystem dependent.
  - RSA cryptosystem
    1.  $G$  finds two large primes,  $p$  and  $q$ , a small integer  $e$ , and computes  $N = pq$ , and the multiplicative inverse  $d$  of  $e$  modulo  $\phi(N) = (p-1)(q-1)$ , i.e.,  $ed \equiv 1 \pmod{\phi(N)}$ .  $G_1$  returns  $t = (N, e, d)$ .
    2.  $(N, e)$  is the public key and  $(N, d)$  is the private key.
    3.  $f_{N,e}(w) = w^e \pmod{N}$ .
    4.  $h(t, f(w)) = h((N, e, d), f(w)) = (f(w))^d \pmod{N}$ .
  - Rabin cryptosystem
  - G1Gamal cryptosystem