Algorithms (IV)

Yu Yu

Shanghai Jiaotong University

Review of the Previous Lecture

Cryptography

The typical setting for cryptography

- **Alice** and **Bob**, who wish to communicate in private.
- **Eve**, an eavesdropper, will go to great lengths to find out what Alice and Bob are saying.

Alice wants to send a *specific message* x, written in binary, to her friend Bob.

1. Alice encodes it as $e(x)$, sends it over.
2. Bob applies his *decryption function* $d\cdot)$ to decode it: $d(e(x)) = x$.
3. Eve, will intercept $e(x)$: for instance, she might be a sniffer on the network.

Ideally the encryption function $e(\cdot)$ is so chosen that without knowing $d(\cdot)$, Eve cannot do anything with the information she has picked up.
In other words, knowing $e(x)$ tells her little or nothing about what x might be.

The Rivest-Shamir-Adelman (RSA) cryptosystem

An example of **public-key cryptography:**

- Anybody can send a message to anybody else using publicly available information, rather like addresses or phone numbers.
- Each person has a public key known to the whole world and a secret key known only to him- or herself.
- When Alice wants to send message x to Bob, she encodes it using his *public key*.
- Bob decrypts it using his *secret key*, to retrieve x.
- Eve is welcome to see as many encrypted messages for Bob as she likes, but she will not be able to decode them, *under certain simple assumptions*.

RSA in a nutshell

Pick any two primes p and q and let $N = pq$. For any e relatively prime to $(p-1)(q-1)$:

1. The mapping $x \mapsto x^e \pmod{N}$ is a bijection on $\{0, 1, \ldots, N-1\}$.
2. The inverse mapping is easily realized: let d be the inverse of e modulo $(p-1)(q-1)$. Then for all $x \in \{0, 1, \ldots, N-1\}$,

$$(x^e)^d \equiv x \pmod{N}.$$

- The mapping $x \mapsto x^e \pmod{N}$ is a reasonable way to encode messages x; no information is lost. So, if Bob publishes (N, e) as his **public key**, everyone else can use it to send him encrypted messages.
- Bob should retain the value d as his **secret key**, with which he can decode all messages that come to him by simply raising them to the dth power modulo N.

Security assumption for RSA

The security of RSA hinges upon a simple assumption:
Given N, e, and $y = x^e \pmod{N}$, it is computationally intractable to determine x.

Universal Hashing

Motivation

We will give a short "nickname" to each of the 2^{32} possible IP addresses.

You can think of this short name as just a number between 1 and 250 (we will later adjust this range very slightly).

Thus many IP addresses will inevitably have the same nickname; however, we hope that most of the 250 IP addresses of our particular customers are assigned distinct names, and we will store their records in an array of size 250 indexed by these names.

What if there is more than one record associated with the same name?

Easy: each entry of the array points to a linked list containing all records with that name. So the total amount of storage is proportional to 250, the number of customers, and is independent of the total number of possible IP addresses.

Moreover, if not too many customer IP addresses are assigned the same name, lookup is fast, because the average size of the linked list we have to scan through is small.

Hash tables

How do we assign a short name to each IP address?

This is the role of a hash function: A function h that maps IP addresses to positions in a table of length about 250 (the expected number of data items).

The name assigned to an IP address x is thus $h(x)$, and the record for x is stored in position $h(x)$ of the table.
As described before, each position of the table is in fact a *bucket*, a linked list that contains all current IP addresses that map to it.
Hopefully, there will be very few buckets that contain more than a handful of IP addresses.

How to choose a hash function?

- There is nothing *inherently wrong* with any single function. If our 250 IP addresses were uniformly drawn from among all $N = 2^{32}$ possibilities, then these functions would behave well.
  The problem is we have no guarantee that the distribution of IP addresses is *uniform*.
- Conversely, there is no *single hash function*, no matter how sophisticated, that behaves well on all sets of data.
  Since a hash function maps 2^{32} IP addresses to just 250 names, there must be a collection of at least

$$2^{32}/250 \approx 2^{24} \approx 16,000,000$$

  IP addresses that are assigned the same name (or, in hashing terminology, **collide**).

Solution: *let us pick a hash function at random from some class of functions.*

Families of hash functions

Let us take the number of buckets to be not 250 but n = 257. a prime number!
We consider every IP address x as a quadruple x =

$$(x_1, x_2, x_3, x_4)$$

of integers modulo n.

We can define a function h from IP addresses to a number mod n as follows:

Fix any four numbers mod $n = 257$, say 87, 23, 125, and 4. Now map the IP address $(x_1, \ldots, x_4)$ to $h(x_1; \ldots, x_4) = (87x_1 + 23x_2 + 125x_3 + 4x_4) \pmod{257}$.

In general For any four coefficients $a_1, \ldots, a_4 \in \{0, 1, \ldots, n-1\}$ write $a = (a_1, a_2, a_3, a_4)$ and define h_a to be the following hash function:

$$h_a(x_1, \ldots, x_4) = (a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + a_4 \cdot x_4) \pmod{n}.$$

Property

Consider any pair of distinct IP addresses $x = (x_1, \ldots, x_4)$ and $y = (y_1, \ldots, y_4)$. If the coefficients $a = (a_1, \ldots, a_4)$ are chosen uniformly at random from $\{0, 1, \ldots, n-1\}$, then

$$\Pr\left[h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)\right] = \frac{1}{n}.$$

Universal families of hash functions

Let

$$\mathcal{H} = \left\{ h_a \mid a \in \{0, 1, \ldots, n-1\}^4 \right\}.$$

It is **universal**:

> *For any two distinct data items x and y, exactly $|\mathcal{H}|/n$ of all the hash functions in $\mathcal{H}$ map x and y to the same bucket, where n is the number of buckets.*

Chapter 2. Divide-and-conquer algorithms

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

Multiplication

Johann Carl Friedrich Gauss

1777 - 1855

$$1 + 2 + \cdots + 100 = \frac{100 \cdot (1 + 100)}{2} = 5050.$$

Gauss once noticed that although the product of two complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a + b)(c + d)$, since

$$bc + ad = (a + b)(c + d) - ac - bd.$$

In our big-O way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity. But this modest improvement becomes *very significant when applied recursively*.

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma

For every n there exists an n' with $n \leq n' \leq 2n$ such that n' a power of 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R.$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

The additions take linear time, as do the multiplications by powers of 2. The significant operations are the four $n/2$-**bit multiplications**; these we can handle by *four recursive calls*.

Our method for multiplying n-bit numbers starts by making recursive calls to multiply these four pairs of $n/2$-bit numbers, and then evaluates the preceding expression in $O(n)$ time.

Writing $T(n)$ for the overall running time on n-bit inputs, we get **the recurrence relation**:

$$T(n) = 4T(n/2) + O(n)$$

Solution: $O(n^2)$.

By **Gauss**'s trick, three multiplications, $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$, suffice, as

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

A divide-and-conquer algorithm for integer multiplication

MULTIPLY(x, y)
// Input: positive integers x and y, in binary
// Output: their product

1. $n =$ max(size of x, size of y) rounded as a power of 2.
2. **if** $n = 1$ **then** return xy.
3. $x_L, x_R =$ leftmost $n/2$, rightmost $n/2$ bits of x
4. $y_L, y_R =$ leftmost $n/2$, rightmost $n/2$ bits of y
5. $P_1 =$ MULTIPLY(x_L, y_L)
6. $P_2 =$ MULTIPLY(x_R, y_R)
7. $P_3 =$ MULTIPLY$(x_L + x_R, y_L + y_R)$
8. return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$.

The time analysis

The recurrence relation:

$$T(n) = 3T(n/2) + O(n)$$

- The algorithm's recursive calls form a **tree structure**.
- At each successive level of recursion the subproblems get halved in size.
- At the $(\log_2 n)^{\text{th}}$ level, the subproblems get down to size 1, and so the recursion ends.
- The **height** of the tree is $\log_2 n$.
- The **branching factor** is 3: each problem recursively produces three smaller ones, with the result that at depth k in the tree there are 3^k subproblems, each of *size* $n/2^k$.

For each subproblem, a linear amount of work is done in identifying further subproblems and combining their answers. Therefore the total time spent at depth k in the tree is

$$3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n).$$

The time analysis (cont'd)

$$3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n).$$

At the very top level, when $k = 0$, we need $O(n)$.

At the bottom, when $k = \log_2 n$, it is

$$O\left(3^{\log_2 n}\right) = O\left(n^{\log_2 3}\right)$$

Between these two endpoints, the work done increases *geometrically* from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level.

The sum of any increasing geometric series is, within a constant factor, simply the last term of the series
Therefore the overall running time is

$$O(n^{\log_2 3}) \approx O(n^{1.59}).$$

We can do even better!

Recurrence relations

Master theorem

If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

Proof of Master Theorem

Assume that n is a power of b. This will not influence the final bound in any important way: n is at most a multiplicative factor of b away from some power of b.

Next, notice that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. This is the height of the recursion tree.

The branching factor of the recursion tree is a, so the kth level of the tree is made up of a^k subproblems, each of size $n = b^k$.

The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

Proof of Master Theorem

The total work done is

$$\sum_{k=0}^{\log_b^n} \left(a^k \times O\left(\frac{n}{b^k}\right)^d \right) = \sum_{k=0}^{\log_b^n} \left(O(n^d) \times \left(\frac{a}{b^d}\right)^k \right).$$

It's the sum of a geometric series with ratio a/b^d.

1. The ratio is less than 1. Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
2. The ratio is greater than 1. The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$:

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

3. The ratio is exactly 1. In this case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

□

Merge sort

The algorithm

```
MERGESORT(a[1 ... n])
// Input: an array of numbers a[1 ... n]
// Output: A sorted version of this array
1.  if n > 1 then
2.    return MERGE(MERGESORT(a[1 ... ⌊n/2⌋]),
3.                 MERGESORT(a[⌊n/2⌋ + 1 ... n]),
4.          else return a.
```

```
MERGE(x[1 ... k], y[1 ... ℓ])
// Input: two sorted arrays x and y
// Output: A sorted version of the union of x and y
1.  if k = 0 then return y[1 ... ℓ]
2.  if ℓ = 0 then return x[1 ... k]
3.  if x[1] ≤ y[1]
4.    then return x[1]∘MERGE(x[2 ... k], y[1 ... ℓ])
5.    else return y[1]∘MERGE(x[1 ... k], y[2 ... ℓ]).
```

The time analysis

The recurrence relation:

$$T(n) = 2T(n/2) + O(n);$$

By Master Theorem

$$T(n) = O(n \log n).$$

An $n \log n$ lower bound for sorting

Sorting algorithms can be depicted as **trees**.

The **depth** of the tree – the number of comparisons on the longest path from root to leaf, is exactly the worst-case time complexity of the algorithm.

Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a *permutation* of $\{1, 2, \ldots, n\}$.

every permutation must appear as the label of a leaf.

This is a binary tree with $n!$ leaves. So, the depth of our tree – and the complexity of our algorithm – must be at least

$$\log(n!) \approx \log\left(\sqrt{\pi\,(2n + 1/3)} \cdot n^n \cdot e^{-n}\right) = \Omega(n \log n),$$

where we use Stirling's formula.

Median

Median

The **median** of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller.
If the list has *even length*, there are two choices for what the middle element could be, in which case we pick the smaller of the two, say.

The purpose of the median is to summarize a set of numbers by a single, typical value.

Computing the median of n numbers is easy: just sort them. The drawback is that this takes $O(n \log n)$ time, whereas we would ideally like something *linear*.

We have reason to be hopeful, because sorting is doing far more work than we really need – we just want the middle element and don't care about the relative ordering of the rest of them.

Selection

Input: A list of numbers S; an integer K.
Output: The kth smallest element of S.

A randomized divide-and-conquer algorithm for selection

For any number v, imagine splitting list S into three categories:

- elements smaller than v, i.e., S_L;
- those equal to v, i.e., S_v (there might be duplicates);
- and those greater than v, i.e., S_R respectively.

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \leq |S_L| \\ v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

How to choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$|S_L|, |S_R| \approx \frac{|S|}{2}.$$

If we could always guarantee this situation, we would get a running time of

$$T(n) = T(n/2) + O(n) = O(n).$$

But this requires picking v to be the median, which is our ultimate goal!

Instead, we follow a much simpler alternative: we pick v randomly from S.

How to choose v? (cont'd)

Worst-case scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \cdots + 1 = \Theta(n^2)$$

Best-case scenario: $O(n)$.

Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the average running time lie? Fortunately, it lies very close to the best-case time.

The efficiency analysis

v is *good* if it lies within the 25th to 75th percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good,

Lemma
On average a fair coin needs to be tossed two times before a "heads" is seen.

Proof.

$$E := \text{expected number of tosses before head is seen}.$$

We need at least one toss, and it's heads, we're done.
If it's tail (with probability $1/2$), we need to repeat. Hence

$$E = 1 + \frac{1}{2}E,$$

whose solution is $E = 2$ □

The efficiency analysis (cont'd)

Let $T(n)$ be the **expected running time** on an array of size n, we get

$$T(n) \leq T(3n/4) + O(n) = O(n).$$

Matrix multiplication

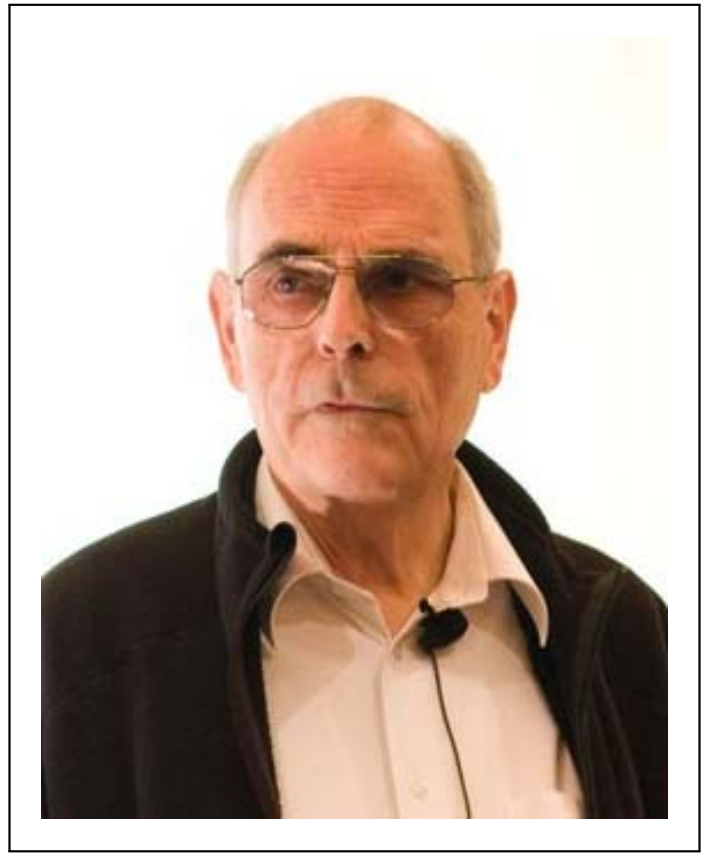
The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j)th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

That is, Z_{ij} is the dot product of the ith row of X with the jth column of Y.

In general, XY is not the same as YX; *matrix multiplication is not commutative*.

The preceding formula implies an $O(n^3)$ algorithm for matrix multiplication.



Volker Strassen (1936 –)

In 1969, the German mathematician **Volker Strassen** announced a surprising $O(n^{2.81})$ algorithm.

Divide and conquer

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Then

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

To compute the size-n product XY, recursively compute eight size-$n/2$ products AE, BG, AF, BH, CE, DG, CF, DH and then do some $O(n^2)$-time addition.

The recurrence is

$$T(n) = 8T(n/2) + O(n^2)$$

with solution $O(n^3)$.

Strassen's trick

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 = P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) & \qquad P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & \qquad P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & \qquad P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

The recurrence is

$$T(n) = 7T(n/2) + O(n^2)$$

with solution $O(n^{\log_2 7}) \approx O(n^{2.81})$.