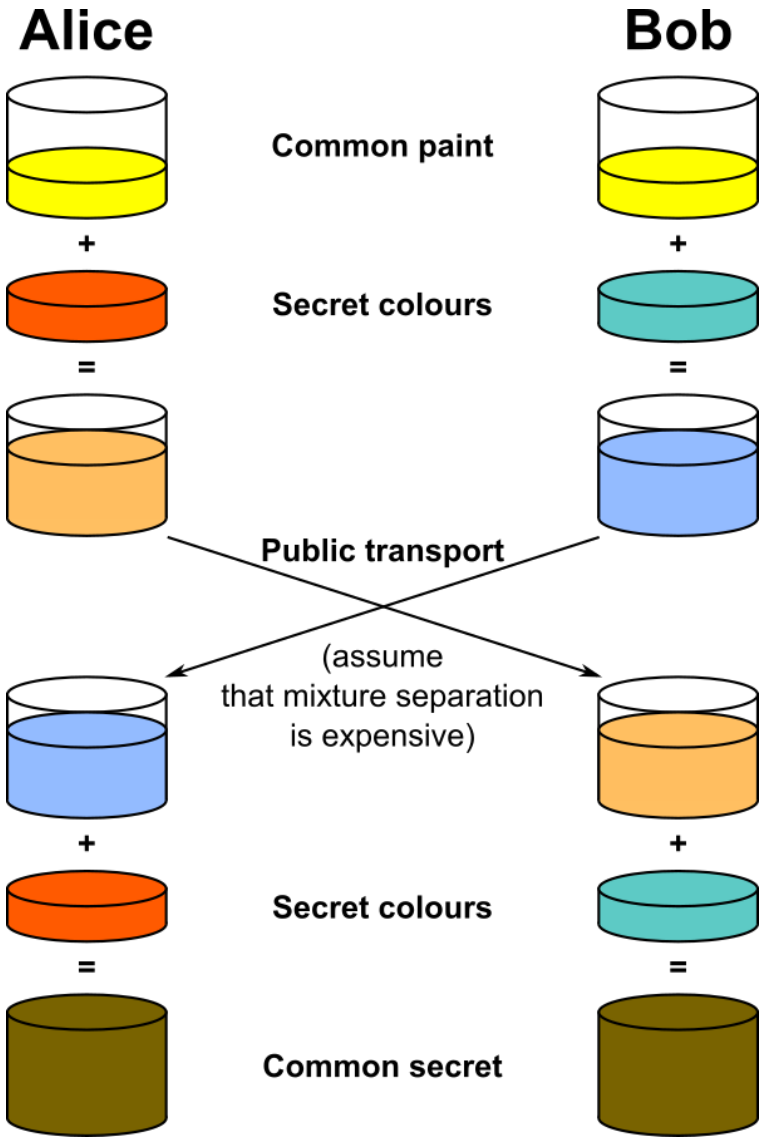
SPCS Cryptography Class Lecture 6

June 29, 2015

Diffie-Hellman Key Exchange

Recall the idea of Diffie-Hellman,



Diffie-Hellman Key Exchange

Source: https://upload.wikimedia.org/wikipedia/commons/4/46/Diffie-Hellman_Key_Exchange.svg

An example with colors:

http://web.stanford.edu/~soarer/spcs/dh_colors.html

Diffie-Hellman Key Exchange

The algorithm with numbers:

- Alice and Bob agree on a large prime p and a primitive root g mod p.
- Alice chooses a private key a (secret color), and Bob chooses a private key b.
- Alice sends Bob $A \equiv g^a \bmod p$ (Alice's mixed color), and Bob sends Alice $B \equiv g^b \bmod p$.
- Alice now has $B \equiv g^b \bmod p$ (Bob's mixed color) and her chosen key a. She can then compute $g^{ab} \equiv B^a \bmod p$ (Alice's secret color + Bob's mixed color).
- Similarly, Bob can compute $g^{ab} \equiv A^b \bmod p$.
- $K = g^{ab} \bmod p$ would be their shared secret key.

Diffie-Hellman Key Exchange

An example with numbers:
`http://web.stanford.edu/~soarer/spcs/dh_numbers.html`

Diffie-Hellman Key Exchange

Summary of Diffie-Hellman: What it does

- Both Alice and Bob would choose a secret key.
- They would both try to calculate $g^{ab} \bmod p$. Alice would get $g^b \bmod p$ from Bob, and she can compute $g^{ab} \bmod p$ afterwards. Similarly for Bob.

For safety against Eve:

- Hardness assumption - hard to separate colors.
- Given

$$p, g, g^a \bmod p, g^b \bmod p,$$

it SHOULD BE hard for Eve to find $g^{ab} \bmod p$. (No one knows!)

Diffie-Hellman Key Exchange

Some issues:

- Security: Hardness of Diffie-Hellman Problem

Computational Diffie Hellman Problem (CDHP)

Given prime p, primitive root g, $g^a, g^b \bmod p$, can one find $g^{ab} \bmod p$ quickly?

The most efficient method right now still goes through Discrete Log Problem,

Discrete Log Problem (DLP)

Given prime p, primitive root g, $g^a \bmod p$, can one find a quickly?

Diffie-Hellman Key Exchange

- Security: Man in the middle attack. We need to make sure we are sending to the right person (authentication) and the message is not tampered with (integrity).
- Implementation: We need
  - Large primes with known primitive roots.
  - Fast exponentiation.

El Gamal Cryptosystem

- Diffie-Hellman allows you to *exchange keys*, but it does not encrypt a random message; it only establishes the key for you to use *symmetric-key cryptography*.
- El Gamal is a *public-key cryptosystem* that allows you to send messages.

El Gamal Cryptosystem

Recall how ElGamal works:

- Bob chooses a large prime p, primitive root g mod p, and a private key a mod p. He publishes p, g, and the public key $A \equiv g^a \bmod p$.
- Alice wants to send Bob the message m. She would choose a random key k, and compute

$$c_2 \equiv mA^k \bmod p \text{ (message with a twist)}$$

$$c_1 \equiv g^k \bmod p \text{ (info you need to untwist it)}$$

- Bob computes $(c_1^a)^{-1} c_2 \bmod p$ to get the message m.

El Gamal Cryptosystem

Why does it work?

- The twisted message was

$$c_2 \equiv mA^k \bmod p$$

  In particular, the "twist" is the $A^k \equiv g^{ak} \bmod p$ part.
- With $c_1 \equiv g^k \bmod p$ (and Bob's private key a!), we can *untwist* the messageby

$$m \equiv (mA^k) \cdot A^{-k} \bmod p \equiv (mA^k) \cdot g^{-ak} \bmod p \equiv \underbrace{(mA^k)}_{c_2} \cdot \underbrace{(g^k)^{-a}}_{c_1^{-a}} \bmod p$$

- `http://web.stanford.edu/~soarer/spcs/elgamal.html`

El Gamal Cryptosystem

Summary for ElGamal:

- Security (i.e. How hard for Eve to crack the message) again depends on Diffie-Hellman problem.
- As stated this system is also prone to Man-in-the-middle attack.
- To implement the system, also need to do fast exponentiation, and have large primes ready.

Analysis of Algorithms

Why analyze algorithms?

- We mentioned that some problems are hard to solve. It means that even with computer we cannot solve the problem in a reasonable amount of time.
- So it makes sense to analyze how much time an algorithm takes (at least roughly) to run.
- Very often we break algorithms into many steps, and we will just calculate (roughly) how many steps it takes for the algorithm to run.

Analysis of Algorithms

But computers are powerful. Why bother?

- Suppose we have an algorithm that solves a problem with n^3 operations for an input n.
- Suppose our computer can do 1000 operations per second. (This is way lower than what computers now can do, but would illustrate the point)
- In one second, we can solve the problem for $n = 10$.
- In one minute, $n^3 \sim 60 \cdot 1000$, so $n \sim 39$.
- In one hour, $n^3 \sim 60 \cdot 60 \cdot 1000$, so $n \sim 153$.

Analysis of Algorithms

- But if we have an algorithm that takes only n^2 operations, things improve a lot.
- In one second, $n^2 \sim 1000$, so $n \sim 31$.
- In one minute, $n \sim 244$.
- In one hour, $n \sim 1897$.

Analysis of Algorithms

Here are some more examples,

Operations	1 second	1 minute	1 hour
2^n	9	15	21
n^3	10	39	153
$3n^2$	18	144	1096
$n^2 + 10n$	27	240	1892
n^2	31	244	1897
$2n \log_2 n$	79	2639	107679
$n \log_2 n$	140	4893	200000
n	1000	600000	3600000

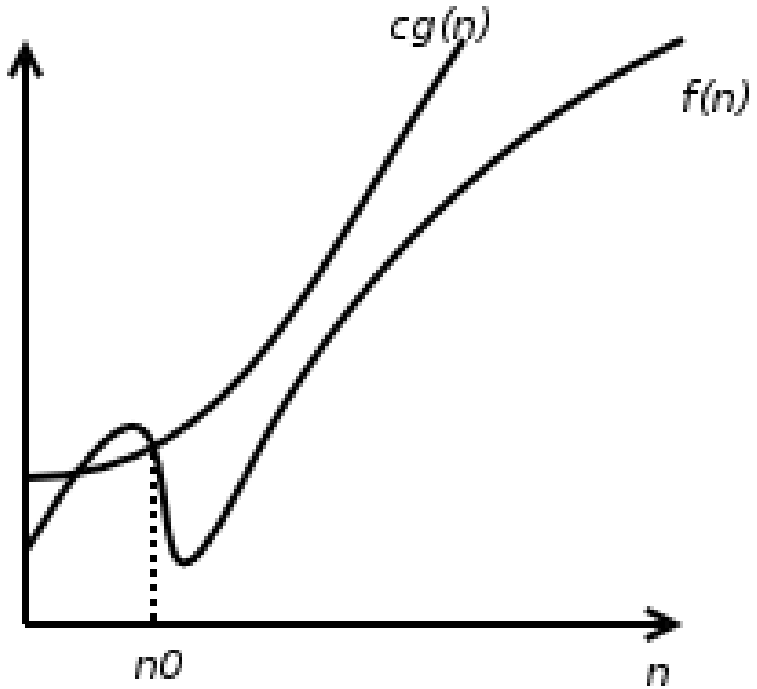
So you see that there is a huge difference in the feasible input size allowed if your algorithm takes n^3 steps rather than n steps.

- $n^2 + 10n$ vs n^2.
- $2n \log_2 n$, $n \log_2 n$ vs n^2 or n^3.

Big O notation

Definition

Let $f(n), g(n)$ be positive functions(Think of them as number of operations). We say that $f(n) = O(g(n))$, if there is some constant C such that $f(n) \leq Cg(n)$ for all n.



Picture from `http://stackoverflow.com/questions/9243691/negative-coefficients-in-polynomial-time-complexity`

Big O notation

- Can replace "for all n" by "for all large enough n", as long as we adjust the constant C.
- See `http://stackoverflow.com/questions/487258/plain-english-explanation-of-big-o` for more explanation.

Example

Is $3n^2 = O(n^2)$ true?
Is $n^2 = O(3n^2)$ true?
So $3n^2$ and n^2 roughly have the same order of magnitude for large n.

Big O notation

Example

Is $n^2 = O(n^2 + 10n)$ true?
Is $n^2 + 10n = O(n^2)$ true?

Example

- $\log n = O(n)$.
- $n = O(2^n)$.
- Is it true that $n^3 + 4n^2 + 5n + 6 = O(n^3)$? What about $O(n^4)$? $O(n^2)$?

Big O notation

We can even do usual addition/multiplication to big O.

Example

- We knew that $n^2 + 10n = O(n^2)$ with constant $C = 10$.
- We also know that $3n^2 = O(n^2)$, with constant $C = 3$.
- We can then check that

$$4n^2 + 10n = (n^2 + 10n) + 3n^2 = O(n^2)$$

  with constant 13.
- This 13 is not the smallest possible number, but it doesn't matter what the exact figure is.
- This says that $O(n^2) + O(n^2) = O(n^2)$.

Big O notation

Example

- What is $O(n) + O(n^2)$?
- $O(n) + O(\log n)$?
- $O(n^2) \cdot O(n)$?
- $O(n)/O(n)$?

- With our notation of Big O-notation, we can talk about running time of algorithms.
- EXACT running time is generally not important. Only the order of magnitudes matter!

Running time of algorithms

Example (Addition)

Given two numbers with at most n digits, how many steps do we need to add them?
(A step = addition of two single digit number)

Solution

- *In order to add them, we have to add the 1's digits, which takes a constant number of steps (i.e., it doesn't depend on the lengths of the numbers).*
- *Then we must add the 10's digits, plus a possible carry, which also takes a constant number of steps.*
- *And so forth. Hence, it takes a constant number of steps for each digit, and there are n digits, so it takes $O(n)$ operations.*
- *So the running time for this algorithm is $O(n)$.*

Running time of algorithms

Example (Multiplication)

How many steps do we need to multiply two numbers with at most n digits?
(A step = multiplication/addition of two one-digit number)

Solution

The Naive way:

- *Multiply every pair of digits.*
- *Every digit multiplication takes one step, and there are n^2 multiplications we need to do, so this takes n^2 steps.*
- *Add up all our results, which takes another n^2 or so steps.*
- *So, multiplication runs in $O(n^2) + O(n^2) = O(n^2)$ steps.*

$$
\begin{array}{rcl}
 & & 1\,0\,2\,3\,4\,5\,6\,7\,8\,9 \\
 & \times & 1\,2\,3\,4\,5\,6\,7\,8\,4\,6 \\
\hline
 & & 6\,1\,4\,0\,7\,4\,0\,7\,3\,4 \\
 & & 4\,0\,9\,3\,8\,2\,7\,1\,5\,6 \\
 & & 8\,1\,8\,7\,6\,5\,4\,3\,1\,2 \\
 & & 7\,1\,6\,4\,1\,9\,7\,5\,2\,3 \\
 & & 6\,1\,4\,0\,7\,4\,0\,7\,3\,4 \\
 & & 5\,1\,1\,7\,2\,8\,3\,9\,4\,5 \\
 & & 4\,0\,9\,3\,8\,2\,7\,1\,5\,6 \\
 & & 3\,0\,7\,0\,3\,7\,0\,3\,6\,7 \\
 & & 2\,0\,4\,6\,9\,1\,3\,5\,7\,8 \\
 & & 1\,0\,2\,3\,4\,5\,6\,7\,8\,9 \\
\hline
 & & 1\,2\,6\,3\,5\,2\,6\,8\,4\,3\,4\,6\,9\,8\,0\,6\,4\,9\,4
\end{array}
$$

Can we do better?

Running time of algorithms

Example (Sorting)

Suppose we have a list of n numbers in random order, and we want to rearrange it in numerical order. How many comparisons does it take to do the job?

Solution

Here is a simple way of doing it, called the selection sort. Suppose we want to sort the list

$$22, 52, 31, 14, 8, 9, 75$$

- *First try to find the smallest number. We go from left to right. Our initial guess is 22.*
  - *If it meets a number which is larger than the current guess, keeps going.*
  - *If it meets a number which is smaller than the current guess, replace guess by that number.*

Running time of algorithms

Solution

- *For our list*

$$22, 52, 31, 14, 8, 9, 75$$

We make the comparisons,

$$22 < 52, 22 < 31, 22 > 14, 14 > 8, 8 < 9, 8 < 75$$

to find the minimum of the list is 8.

- *This way, one can find the minimum in $O(n)$ steps.*
- *Now put this number away, and find the minimum for the remaining $n-1$ numbers again. Keep repeating, we see that the number of comparisons needed is*

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

Can we do better?

Running time of algorithms

What about the actual time complexity/running time?

- It depends on the size of the numbers and your efficiency of comparing numbers.
- Sometimes to separate the problems, we will use units - for example, one unit of time here may be time needed to do one comparison.
- In this course we distinguish them by time vs steps.

Running time of algorithms

Example (Searching)

Suppose we have a list of n SORTED numbers, and we want to find one particular number in the list. How many comparisons does it take to do the job?

The simplest way to do it is to just compare your wanted number with each of the numbers in the list. This takes $O(n)$ steps.

Can we do better?

Running time of algorithms

What is a good algorithm?

- Roughly speaking, a polynomial time algorithm (i.e. one with algorithm complexity a polynomial in n if the input is n) is good.
- For example, $O(n^2)$ is good, $O(n)$ is even better.
- An exponential time algorithm (something of the form $O(c^n)$) is considered too slow.
- For example, $O(2^n)$ is bad, $O(n!)$ is really bad.
- Of course, a polynomial time algorithm with large exponent may still be impractical in practice.

Fast exponentiation

In Diffie-Hellman, one often needs to compute $g^k \bmod p$ for some large k. We surely want to compute this quickly! The naive way:

- One can simply multiply g k times - thus $O(k)$ multiplications are involved.
- Note that this is NOT the running time - since the cost of each multiplication depends on the number of digits, and also the time for modulo p.

Can we reduce the number of multiplications involved?

Fast exponentiation

One basic strategy is repeated squaring.

Example

Find $3^9 \bmod 101$.

The naive way involves 9 multiplications. However, we can also proceed as follows.

Solution

- *First we square to get* $3^2 \bmod 101$.
- *Square again to get* $3^4 \bmod 101$.
- *Square again to get* $3^8 \bmod 101$.
- *Finally,* $3^9 \equiv 3^8 \cdot 3 \bmod 101$. *Only 4 multiplications are involved in this case!*

Fast exponentiation

- In general, write k in terms of base 2, and compute g^{2^i} for all i such that $2^i \leq k$. This takes $\log_2 k$ multiplications.
- The final step is to multiply the 2^i-th power involved in the binary expansion of k. This takes at most $\log_2 k$ multiplications as well.
- In total this algorithm needs only $O(\log_2 k)$ multiplications.

Fast exponentiation

What about actual time complexity?

- This would depend on your algorithm to multiply two numbers, and doing modulo arithmetic.
- For a large prime p, there are roughly $\log_2 p$ digits in base 2.
- Even with our naive algorithms these operations (multiplication or mod p) would take at most $O((\log p)^2) = O(p^\epsilon)$ time.
- The point is as long as k is large enough compared to our modulus p, these are insignificant.