

**Course:** c6924@csis.hku.hk, <http://www.csis.hku.hk/~c6924/>

**Lecturer:** Isaac K. K. To, [kkto@csis.hku.hk](mailto:kkto@csis.hku.hk), CB407, Sat 4pm–6pm.  
**Tutor:** Wang Lian, [wlian@csis.hku.hk](mailto:wlian@csis.hku.hk), HW506.

**Text:** Introduction to Algorithms, Cormen, Leiserson and Rivest. MIT Press.

**Course aim:**

- To learn some methodologies that are widely used in designing algorithms.
- To learn some metrics in which algorithms are designed for.
- To understand that some problems are inherently difficult, and what to do to deal with them.

ALG/TITCS L01-1

For both courses:

- 1 quiz (Oct 28). (15% for CSIS6924, 8% for CSIS0324.)
- 3 assignments. (15% for CSIS6924, 12% for CSIS0324.)
- Final examination. (70% for both courses.)

For students taking CSIS0324:

- Reading project. (10%) Late October, 1/2 hour presentation at last 2 weeks.

ALG/TITCS L01-2

Lecture today

Have a quick overview of some neat ideas in theoretical computer science.

Our course will cover a few of these ideas. A quick overview at least allow you to think about whether to read it at home.

- Interesting algorithms: FFT, Primality Testing.
- Design methodology: Greedy algorithms, Dynamic Programming.
- Problem difficulty: Uncomputability, NP-completeness.
- Alternative metrics: Approximation algorithm, Randomized algorithms.
- Alternative setting: Amortized analysis, On-line computation, Parallel Algorithms.

ALG/TITCS L01-4

Fast Fourier Transform (FFT)

**The Polynomial Multiplication Problem:** multiply two polynomials of degree  $n$ , e.g.  $4x^3 + 2x^2 - 3x - 1$  and  $0.5x^3 + x^2 - 3x + 1$ . ( $n = 4$ .)

Grade-school method:

$$\begin{array}{r}
 \phantom{x)} \phantom{0.5x^3} \phantom{+x^2} \phantom{+3x} \phantom{+1} \\
 \phantom{x)} \phantom{0.5x^3} \phantom{+x^2} \phantom{+3x} \phantom{+1} \\
 \hline
 \phantom{x)} \phantom{0.5x^3} \phantom{+x^2} \phantom{+3x} \phantom{+1} \\
 \phantom{x)} \phantom{0.5x^3} \phantom{+x^2} \phantom{+3x} \phantom{+1} \\
 \phantom{x)} \phantom{0.5x^3} \phantom{+x^2} \phantom{+3x} \phantom{+1} \\
 \phantom{x)} \phantom{0.5x^3} \phantom{+x^2} \phantom{+3x} \phantom{+1} \\
 \phantom{x)} \phantom{0.5x^3} \phantom{+x^2} \phantom{+3x} \phantom{+1} \\
 \hline
 2x^6 \phantom{+x^5} \phantom{-1.5x^4} \phantom{-0.5x^3} \\
 \hline
 2x^6 \phantom{+x^5} \phantom{-1.5x^4} \phantom{-0.5x^3} \phantom{-8x^2} \phantom{-6x} \phantom{-1}
 \end{array}$$

This method needs  $O(n^2)$  time. Any faster method?

How about:  
 If we represent polynomials by  $(x_0, f(x_0)), \dots, (x_{2n-1}, f(x_{2n-1}))$ ?

Then the problem becomes trivial. We just need to compute the product of  $2n$  numbers, needing  $O(n)$  time.

ALG/TITCS L01-5

Fast Fourier Transform (cont.)

**Problem:** Most of the time, we want our polynomial in the first form. Can we convert from/to this new form quickly?

What this requires is actually to evaluate the polynomial at  $2n$  points, and then solve a set of  $2n$  equations, both of them seems to require  $\Omega(n^2)$  time.

But if we have good choices of  $x_0, x_1, \dots, x_{2n-1} \dots$

**FFT:** Suppose  $n$  is a power of 2. We can then take the numbers  $x_i$  to be the  $n$ -th complex root of unity. Then the results of most computations can be reused a large number of times, and the running time can get down to just  $\Omega(n \lg n)$  for both the conversions.

ALG/TITCS L01-6

Primality Testing

**Problem Primality:** given a number  $N$  of  $n$ -bits, test whether it is a prime. This is a problem needed in a large number of cryptographic algorithm and protocols.

If we do it in the naive way and try to test for factors...

- We will be testing until around  $\sqrt{N}$ .
- A constant fraction of numbers can be skipped (e.g. since they are multiples of 2 or 3).
- This ends up with an algorithm of running time  $O(\sqrt{N})$ .
- Since  $N$  is of size  $\Omega(2^n)$ , this means the algorithm runs at  $O(2^{n/2})$  running time: very slow.

ALG/TITCS L01-7

## Primality Testing (cont.)

Idea 1:

### Fermat's theorem:

If  $N$  is a prime, then  $a^{N-1} \equiv 1 \pmod{N}$  unless  $a$  is a multiple of  $N$ .

Performing this "Fermat's test" requires only  $O(\lg N \lg \lg N)$  time to check, i.e.  $O(n \lg n)$ -time!

Difficulty: there are some numbers that  $N$  is not a prime but yet  $a^{N-1} \equiv 1 \pmod{N}$ . These numbers are called "base- $a$  pseudo-primes".

Idea 2: we can choose  $a$ . If Fermat's test fails for most  $a$ 's for composite numbers, we can use a random  $a$  and hope that we choose the right  $a$ .

Difficulty: there are some composite numbers that pass Fermat's test for all choices of  $a$ 's (e.g. 561). These numbers are called *Carmichael numbers*. This difficulty can be side-stepped.

ALG/TITCS L01-8

## Greedy Algorithms

**The Activity-Selection Problem:** Given a set of  $n$  jobs, each with a starting time and an finishing time. Find the largest set of jobs that do not overlap each other.

Intuitively, there are a large number of algorithms that seems "make sense". But proving that the algorithm really work and find the best set of jobs seems not easy.

In fact, one algorithm really work, and it requires only  $O(n \lg n)$  time:

**Algorithm:** Sort the jobs according to the finishing time. Consider the jobs from the earliest finishing job to the latest. Always accept the first job. Then accept a job whenever it does not overlap with the last accepted job.

Why this is optimal? What other problems can be solved this way?

ALG/TITCS L01-9

## Dynamic Programming

**Edit steps problem:** We have two strings,  $w_1$  and  $w_2$ , over a small alphabet. We want to use three editing operations to turn  $w_1$  to  $w_2$ . The allowed operations include Inserting a character, Deleting a character, and Changing a character to another character. What is the minimum number of steps required,  $E(w_1, w_2)$ ?

Reduction: let  $w_1 = u_1c_1$  and  $w_2 = u_2c_2$ , where both  $c_1$  and  $c_2$  are single characters. If  $c_1 = c_2$ , then  $E(w_1, w_2) = E(u_1, u_2)$ . Otherwise, we can do one of the three steps at the end to make them equal. So  $E(w_1, w_2)$  is one of  $E(u_1, u_2) + 1$ ,  $E(u_1, w_2) + 1$  or  $E(w_1, u_2) + 1$ .

Solve it by divide-and-conquer: too many subproblems!

Idea: but there are not so many possible subproblems. Let's do all of them!

ALG/TITCS L01-10

## Uncomputability

Some seemingly reasonable questions could be unexpectedly difficult.

**The Post Correspondence Problem:** Given a finite list of pairs of strings,  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ . Is there a finite sequence  $i_1, i_2, \dots, i_m$  such that the concatenation  $x_{i_1}x_{i_2}\dots x_{i_m}$  is the same as the concatenation  $y_{i_1}y_{i_2}\dots y_{i_m}$ ?

This problem seems to be reasonable: everything is finite, and everything is well-defined.

But actually, largely because we have no bound on  $m$ , the problem is provably unsolvable. That is, there is no algorithm that guarantee termination and when it terminates, it decides correctly whether it can be done or not.

How to show that it is unsolvable, and what other problems are unsolvable?

ALG/TITCS L01-11

## NP-completeness

**Vertex Cover Problem:** We have a graph  $G(V, E)$ . Is there a set of vertices  $C$  of at most  $k$  vertices such that, for any edge  $E$ , at least one end-points of  $E$  is in  $C$ . (We say such sets is a vertex cover.)

Unlike PCP, we have a simple algorithm that works: for each  $k$ -combination of  $V$ , check whether it is a vertex cover.

Time complexity:  $O(|V|^k |E|)$ .

Can we do better? Yes. But can we make the running time a polynomial of  $|V|$  and  $k$ ?

**Well...** We have just asked the most significant problem in theoretical computer science.

The minimum vertex cover problem is "NP-complete". There are a lot of other problems that are "NP-complete", and none of them have any known polynomial-time algorithm. But if one of them admits polynomial-time algorithm, all will.

ALG/TITCS L01-12

## Approximation Algorithms

**The Minimum Vertex Cover Problem:** We have a graph  $G(V, E)$ . Find a vertex cover  $C$  with the minimum number of vertices.

Clearly, it is an extension of the Vertex Cover Problem. If you can solve the Minimum Vertex Cover problem, you can also solve the Vertex Cover Problem. Since the Vertex Cover Problem is already difficult, the Minimum Vertex Cover Problem is more difficult.

But if we can accept algorithms that give you a vertex cover which has a bit more vertices than the minimum, it is a very different story:

**Algorithm:** Initialize a empty set of vertices for return. While there is any edge left, add both end-points to the set, and delete all edges adjacent to these two vertices.

It turns out that this simple algorithm, with running time  $O(|V| + |E|)$ , never chooses more than twice the required number of vertices. We say the approximation factor of the algorithm is 2.

ALG/TITCS L01-13

### Randomized Algorithms

**Problem:** Given  $n$  numbers, find a number which is no less than the medium.

- A simple solution is to return the maximum. This requires  $\Theta(n)$  time.
- The best deterministic solution is to return the maximum of the first  $\lceil (n+1)/2 \rceil$  numbers. This is still  $\Theta(n)$  time.

But if we can make error...

**Randomized algorithm:** Randomly pick a number among the  $n$ . Return it.

This algorithm runs in constant time. But it can make error, and the error probability is 0.5. (Las Vegas algorithm.)

This algorithm can be repeated arbitrary number of times, taking the maximum of the answers. This makes the error rate arbitrarily low.

ALG/TITCS L01-14

### Amortized Analysis

Alternative model: We want a data structure to hold some data. We don't need every operation to be done efficiently, but "overall" it must perform well. In particular, after a sequence of operations, the aggregated running time must be bounded by a good function of the number of operations.

**Dictionary problem:** Find a data structure which support insertion, deletion and searching.

Classic solution: using red-black tree, we need worst-case  $O(\lg n)$  time for each operation when there are  $n$  items in the dictionary. This requires some extra storage (for the color), and a complicated case analysis for each insertion/deletion.

**Splay tree:** Use no extra storage at all except the tree pointers. We perform a constant time splay operation everytime a node is accessed. Amortized over all operations, each requires  $O(\lg n)$  time.

ALG/TITCS L01-15

### On-line Algorithms

Alternative model: output must start before input is complete, and we cannot "correct" the output.

**The Ski-rental problem:** To buy a pair of ski needs \$1000. To rent it, each time it costs \$50. You have no idea at all about for how many times you'll go skiing. Should you buy?

On-line algorithm criteria: to do reasonably good even in the worst scenario.

**Example algorithm:** If you wait until the 21st time before you buy: you always get the optimal if you eventually don't ski 21 times, and if you quit at the 21st time, you just spend double the amount of the minimum money to spend.

We say the competitive ratio of the algorithm is 2. Better algorithms have smaller competitive ratio.

ALG/TITCS L01-16

### Parallel Computations

Alternative model: we have  $p$  processors instead of 1, and they share the same clock. Can we use the processors to make the computation real fast?

**Sorting problem:** Given  $n$  numbers in an array, rearrange them to ascending order using  $n$  processors.

Naive solution: use merge sort, break the input into  $p$  equal-sized arrays. Let each processor sort each part, and then ask one processor to deal with the final  $p$ -way merging.

Not so good: the final merging will take majority of time.

A simple parallel algorithm exist (bitonic sorting), which actually make sure that the problem can be solved with  $n$  processors in  $O(\lg^2 n)$  time.

ALG/TITCS L01-17