1 Analysis of Algorithms

- how long your code takes to run, with "how long" referring to the number of steps. Then the question remains: what is a step?
- look for rough bounds, depend on size of the problem but don't depend on what computer you have

We say $f(n)$ is $O(g(n))$ if there exist constants C, N such that $\forall n \geq N$, we have $f(n) \leq Cg(n)$. Or a lower bound Ω, then $f(n) \geq Cg(n)$.

We say $f(x)$ is $\theta(g(x))$ if both $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.

Example 1.1 (Bubble Sort Algorithm)**.** Input: list of numbers $L = [\ell_0, \ldots \ell_n]$.

Output: L, but sorted. Repeat the following:

```
for i from 0 to n-1:
    if ℓ_i > ℓ_{i+1}:
        swap ℓ_i with ℓ_{i+1} in L
```

Until we get through L without swapping anything.

Best case scenario: L is already sorted: then we have 0 swaps and n comparisons.

Worst case scenario: L is reverse sorted. Then this algorithm takes $O(n^2)$ steps. Note the $n-1$ in the algorithm: we have a possible optimization by decreasing this value by 1 at each pass. Without this optimization, the n passes take n steps each. With this optimization, however, $n+(n-1)+(n-2)+\ldots+2+1$ steps $= \frac{n(n+1)}{2} = O(n^2)$ steps.

In the average case number of swaps: Imagine L is n uniform random numbers in $[0,1]$. i.e. the ranking of elements of L gives a uniform $\pi \in S\pi$. The average case number of swaps is $E(inv(\pi)) = \frac{n(n+1)}{4} = O(n^2)$. Note that the best sorting algorithms are $\theta(n \log n)$.

The Euclidean algorithm for the greatest common divisors of two integers has input of two numbers, $q_0, q_1 \in \mathbb{N}$ with $q_0 > q_1$.

Then we write:

$$
\begin{aligned}
q_0 &= a_1q_1 + q_2 \\
q_1 &= a_2q_2 + q_3 \\
q_2 &= a_3q_3 + q_4 \\
&\vdots \\
q_{k-1} &= a_kq_k + q_{k+1} \\
q_k &= a_{k+1}q_{k+1}
\end{aligned}
$$

So q_{k+1} is the greatest common divisor of (q_0, q_1).

Then the question remains: How long does this algorithm run for? The run time in the worst case situation should be when all of the a_is are 1, and when $q_{k+1} = 1$. Therefore we look at solutions to the Fibonacci numbers.

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, n \geq 2$$

So how big is n compared to F_n?

If $G(x) = \sum_{n\geq 0} F_nx^n = 1 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + \ldots$, then check: $G(x) = \frac{x}{1-x-x^2}$, then the roots are $\frac{1+\sqrt{5}}{2} = \varphi$ and $\frac{1-\sqrt{5}}{2}$. So $F_n is \theta(\varphi^n)$. i.e. n is $\theta(\log_\varphi(F_n))$.

So the Euclidean Algorithm runs in time $O(\log(q_0))$.

Example 1.2. $\begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & \ldots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \ldots & b_{nn} \end{bmatrix} = \begin{bmatrix} \quad \sum_k a_{ik}b_{kj} \quad \end{bmatrix}$ takes $\theta(n^3)$ multiplications. A faster matrix multiplication algorithm is given by the Strassen ALgorithm. Then normally for two matrices $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and $B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$ multiplying naively takes 8 multiplications (2^3). But Strassen found a way to do it with 7 multiplications and more additions, which is faster. See wikipedia for more information.

Note that this also works for block matrices. Recursively we can do this on a $2^k \times 2^k$ matrix, then the number of steps is $7^k = 2^{\log_2(7)k}$. Then if n is 2^k, then this is $O(n^{\log_2 7}) \approx O(n^{2.8\ldots})$.

In general, how much faith should we put into analysis of algorithms? Only as much as will help you optimize your programs when necessary.