COMP S265F Unit 1: Analysis of Algorithms

Dr. Keith Lee

School of Science and Technology

The Open University of Hong Kong

Overview

- What is an algorithm?
- Euclid’s algorithm
- Different types of analysis of algorithms
- Proof of correctness
- Asymptotic notation
- Time complexity analysis
  - Best case analysis
  - Worst case analysis
  - Average case analysis
- Improving Euclid’s algorithm
- Time complexity of the improved Euclid’s algorithm

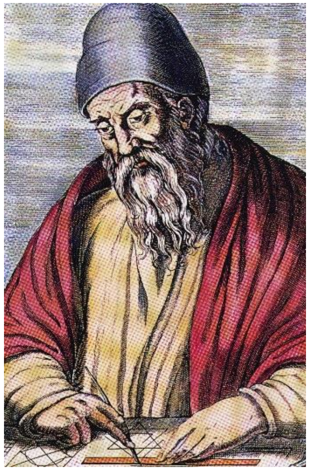
What is an algorithm?

- An algorithm is a sequence of precise and concise instructions that guide you (or a computer) to solve some specific problem.
- **Examples:** cooking recipe, smartphone trouble shooting instructions, furniture assembly directions.

Note:

- Unlike programs, algorithms are free from any grammar rules (of a programming language).
- **For programs:** form is more important than content; i.e., although your program has the right ideas to solve the problem, it is still wrong if it has syntax error.
- **For algorithm:** content is more important than form; i.e., it is acceptable as long as you can guide me to find the solution correctly; I don't care what languages (or in whatever ways) you use. It is still okay even if there are some trivial details missing.

Example: Euclid's Algorithm



- First appeared in

  Euclid, Elements (c. 300BC), Book 7.
- It solves the following specific problem:
  - **input:** two positive integers **a**, **b**
  - **output:** the **greatest common divisor (gcd) of a, b**.
- **Example:**
  - **input:** 100, 92
  - **output:** gcd(100, 92) = 4
- **Algorithm (mutual subtraction)**
  - Replace the larger number by the difference of the two numbers until both are equal; then the answer is this common value.

Example: Euclid’s Algorithm (cont’)

- Example
  {18, 42} → {18, 24} → {18, 6} → {12, 6} → {6,6}
  Answer: gcd(18, 42) = 6.

- A **Python** implementation:

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Pseudo-code:

```
gcd(a, b):
    while a ≠ b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

- You can describe an algorithm in whatever form you like, as long as the description is concise and precise.
  - E.g., **Pseudo-code**: In your test, assignment and exam, you only need to describe to us clearly how the algorithm works.

Analysis of algorithms

After designing an algorithm, we analyze it. More precisely, we do the followings:

- **Proof of correctness:** whether it is correct, i.e., it returns the correct answer for any possible input.
- **Time complexity analysis:** find out how fast your algorithm runs
  - in the best case,
  - in the worst case,
  - on average.
- **Space complexity analysis:** decide how much memory space your algorithm requires.
- **Look for improvement (or proof of optimality):** decide whether your algorithm is best possible; can we improve it such that it runs faster or use less memory space.

Proof of Correctness (for Euclid)

Prove that given any positive integers a and b, the Euclid's algorithm always returns the greatest common divisor of a and b.

Fact 1: If d divides integers x and y; then d divides x+y and x-y.

Lemma 1: For any +ve integers $a > b$, we have $\gcd(a, b) = \gcd(b, a-b)$.

Proof.

- Let $g = \gcd(a, b)$ and $g' = \gcd(b, a-b)$
  [Here, gcd(x,y) denotes the actual gcd of x and y, not our function]
- By definition, g divides a and b. Thus, by Fact 1, g divides a-b.
- Hence, g is a common divisor of b and a-b, and it is no greater than $\gcd(b, a-b)$. In other words, $g \le g'$.

Proof of Correctness (for Euclid)

Prove that given any positive integers a and b, the Euclid's algorithm always returns the greatest common divisor of a and b.

Fact 1: If d divides integers x and y; then d divides x+y and x-y.

Lemma 1: For any +ve integers a > b, we have gcd(a, b) = gcd(b, a-b).

Proof.

- Let g = gcd(a, b) and g' = gcd(b, a-b).
- By definition, g divides a and b. Thus, by Fact 1, g divides a-b.
- Hence, g is a common divisor of b and a-b, and it is no greater than gcd(b, a-b). In other words, g ≤ g'.
- By definition, g' divides b and **x**=a-b, and from Fact 1, g' divides b+**x** = b+(a-b) = a.
- Hence, g' is a common divisor of a and b, and is no greater than gcd(a, b). In other words, g' ≤ g.
- Therefore, g = g', i.e., gcd(a, b) = gcd(b, a-b).

Theorem: Given any +ve integers a and b, Euclid correctly finds gcd(a,b).

Example: gcd(42, 18)=gcd(24, 18)=gcd(18, 6)=gcd(12, 6)=gcd(6, 6)=6.

Proof. By mathematical induction on n = max(a, b).

Base Case:

- When n = 1, then a = b = 1.
- Euclid correctly returns gcd(a, b) = 1.

Theorem: Given any +ve integers a and b, Euclid correctly finds gcd(a,b). (cont')

Example: gcd(42, 18)=gcd(24, 18)=gcd(18, 6)=gcd(12, 6)=gcd(6, 6)=6.

Proof. By mathematical induction on n = max(a, b).

Induction hypothesis:

Suppose Euclid correctly returns gcd(x, y) whenever ***max(x, y) < n***.

Let us consider the three cases when n = max(a, b).

Case 1: a = b. Euclid correctly returns gcd(a, b) = a.

Theorem: Given any +ve integers a and b, Euclid correctly finds gcd(a,b). (cont')

Example: gcd(42, 18)=gcd(24, 18)=gcd(18, 6)=gcd(12, 6)=gcd(6, 6)=6.

Proof. By mathematical induction on n = max(a, b).

Induction hypothesis:

Suppose Euclid correctly returns gcd(x, y) whenever ***max(x, y) < n***.

Case 2: a > b.

- After the first iteration of the while loop, a becomes a-b, and b is still b.
- Thus, the rest of the execution finds gcd(a-b, b).
- Since max(a-b, b) < n, by the Induction Hypothesis,
  Euclid correctly returns gcd(a-b, b).
- By Lemma 1, gcd(a, b) = gcd(a-b, b).
  We conclude that Euclid correctly returns gcd(a, b).

Theorem: Given any +ve integers a and b, Euclid correctly finds gcd(a,b). (cont')

Example: gcd(42, 18)=gcd(24, 18)=gcd(18, 6)=gcd(12, 6)=gcd(6, 6)=6.

Proof. By mathematical induction on n = max(a, b).

Induction hypothesis:

Suppose Euclid correctly returns gcd(x, y) whenever ***max(x, y) < n***.

Case 3: b > a. (Similar to Case 2)

- After the first iteration of the while loop, b becomes b-a, and a is still a.
- Thus, the rest of the execution finds gcd(a, b-a).
- Since max(a, b-a) < n, by the Induction Hypothesis,
  Euclid correctly returns gcd(a, b-a).
- By Lemma 1, gcd(a, b) = gcd(a, b-a).
  We conclude that Euclid correctly returns gcd(a, b).

Theorem: Given any +ve integers a and b, Euclid correctly finds gcd(a,b). (cont')

Example: gcd(42, 18)=gcd(24, 18)=gcd(18, 6)=gcd(12, 6)=gcd(6, 6)=6.

Proof.

- Thus, we have shown in all the three possible cases that when max(a, b) = n, Euclid correctly finds gcd(a, b).
- By the principle of mathematical induction, we conclude that Euclid correctly finds gcd(a, b) for all +ve integers a, b.

Time and Space complexity analysis

- The central goal in such analysis is to make quantitative assessments of the "**goodness**" of the algorithms.
- We express the time/space complexity of algorithms in terms of some functions (usually polynomial) of the input size.

Example:

- The bubble-sort algorithm takes $n^2/2 + n + 3$ steps to sort n numbers.
- We say that the time complexity of the bubble-sort algorithm is **$O(n^2)$**.

Asymptotic notations

- Key notion: **Asymptotic analysis** (i.e., rough analysis)
- Given two functions **f(n)** and **g(n)** on integer **n**, we say that
  - **f(n) = O(g(n))**
    if there is some constant $c > 0$ such that **f(n)** $\leq$ c**g(n)** for all sufficiently large n.
  - **f(n) = Ω(g(n))**
    if there is some constant $d > 0$ such that **f(n)** $\geq$ d**g(n)** for all sufficiently large n.
  - **f(n) = Θ(g(n))**
    if we have both *(i)* f(n)=O(g(n)) and *(ii)* f(n)= Ω(g(n)).

Asymptotic notations (cont')

Intuitively,

- **$f(n) = O(g(n))$** is a formal way to say $f(n) \le g(n)$ *asymptotically*.

- **$f(n) = \Omega(g(n))$** is a formal way to say $f(n) \ge g(n)$ *asymptotically*.

- **$f(n) = \Theta(g(n))$** is a formal way to say $f(n)=g(n)$ *asymptotically*.

Asymptotically means roughly, i.e.,

- the relationship may not hold for some integers n, but it holds for all **large enough** n, and
- we don't mind the constant factor of the functions.

Asymptotic notations: Examples

- **200 n = O(n)**
  because when c = 201, 200n ≤ c n for large n.

- **1000 n = O(n log n)**
  because

$$\lim_{n\to\infty} \frac{1000n}{n \log n} = \lim_{n\to\infty} \frac{1000}{\log n} = 0 \leq 1 \ ,$$

which implies that when c=1, $1000n \leq cn \log n$ for sufficiently large n.

- Thus, for the time complexity of bubble-sort, we don't need to say its time complexity is $n^2/2 + n + 3$. It suffices to say it is $O(n^2)$ or $\Theta(n^2)$.

Time complexity analysis

Identify some important operations/steps in the algorithms and estimate how many times these operations/steps needs to be executed.

- For example, when we analyze algorithms for sorting, we count the number of comparisons.

Time complexity analysis of Euclid’s algorithm:

- Operation: subtractions
- How many subtractions are needed to find gcd(a ,b)?

Euclid: Best case analysis

- O(1) subtraction.
- Best case analysis is often useless (except maybe when you are “selling” this algorithm).

Euclid: Worst case analysis

- A moment reflection reveals that the worst case occurs when a >> b.
- E.g., when a = 2,000,000 and b = 2,
  Euclid needs to make around 1,000,000 subtractions.
- In other words, the worst case time complexity of Euclid is O(a), or more precisely, O(max(a, b)).

Euclid: Average case analysis

- Knuth and Yao proved that if a and b (a > b) are given "randomly", then **on average**, Euclid needs to make

$$\frac{6(\ln a)^2}{\pi^2} + O(\ln a(\ln \ln a)^2) \text{ subtractions.}$$

- Here, $\ln(a) = \log_e(a)$.

- *Average case analysis usually requires difficult usage of probability theory.*

Look for improvement

Examine your complexity analysis carefully to find out

- What is the worst case?
- Where is the bottleneck in the computation?
- Can we reduce the number of steps used in this bottleneck?

Can we improve Euclid's algorithm for finding gcd(a, b)?

- **Worst case:** a >> b
- **Bottleneck:** we need to subtract b from a many times before we can get to the point where a < b.
- **Key question:** how can we reduce the number of subtractions?

Look for improvement: Euclid

Idea: finding **shortcut**

$$(a,b) \rightarrow (a\text{-}b,b) \rightarrow (a\text{-}2b,b) \rightarrow \ldots \rightarrow (a\text{-}mb,\ b) \rightarrow \ldots$$

We want to find the jump here, from a to a - mb where a - mb < b.

- To find **r = a - mb** with **r < b**, note that a = mb + r where r < b.
- Thus, r must be the remainder of a / b, i.e., r = a mod b.
  - Python: `r = a % b`
- Hence, we can do the jump as follows:

$$(a,b) \longrightarrow (a \bmod b,\ b) \longrightarrow \ldots$$

We can do similar jump for the remaining subtractions.

When should we stop?

The condition for "jumping":

- As long as $a > b$ and $b \neq 0$, we can do the jump:

  $(a,b) \longrightarrow (a \bmod b, b)$.

What happens when $a > b$ and $b = 0$?

- **Example:**

$$\gcd(42, 18) = \gcd(42 \bmod 18, 18) = \gcd(18, 6)$$
$$= \gcd(18 \bmod 6, 6) = \gcd(0, 6)$$

From $18 \bmod 6 = 0$, we know that $\gcd(18, 6) = 6$, hence $\gcd(42, 18) = 6$.

When should we stop? (cont')

The condition for "jumping":

- As long as $a > b$ and $b \neq 0$, we can do the jump:

  $(a,b) \longrightarrow (a \bmod b, b)$.

What happens when a > b and b = 0?

- In general,

  $\gcd(a,b) \longrightarrow \gcd(a \bmod b, b) \longrightarrow \dots \longrightarrow \gcd(x, c) \longrightarrow \gcd(\underbrace{\mathbf{0}}_{x \bmod c}, c)$

- We conclude $\gcd(x, c) = c$.
- Hence, $\gcd(a, b) = \gcd(a \bmod b, b) = \dots = c$.

An improved Euclid's algorithm

- A Python implementation

```
def gcd2(a, b):
    if a < b:
        a, b = b, a  # swap a & b
    while a % b > 0:
        a, b = b, a % b
    return b
```

Time complexity of improved Euclid

Example:

$$
\begin{aligned}
\gcd(89, 55) &= \gcd(55, 34) = \gcd(34, 21) = \gcd(21, 13) \\
&= \gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) \\
&= \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1
\end{aligned}
$$

Idea: Consider the product of the two arguments.

$$
\begin{aligned}
\gcd(\underbrace{89, 55}_{4895}) &= \gcd(\underbrace{55, 34}_{1870}) = \gcd(\underbrace{34, 21}_{714}) = \gcd(\underbrace{21, 13}_{273}) \\
&= \gcd(\underbrace{13, 8}_{104}) = \gcd(\underbrace{8, 5}_{40}) = \gcd(\underbrace{5, 3}_{15}) \\
&= \gcd(\underbrace{3, 2}_{6}) = \gcd(\underbrace{2, 1}_{2}) = \underbrace{1}_{1}
\end{aligned}
$$

Time complexity of improved Euclid (cont')

Idea: Consider the product of the two arguments.

$$\text{gcd}(\underbrace{89, 55}_{4895}) = \text{gcd}(\underbrace{55, 34}_{1870}) = \text{gcd}(\underbrace{34, 21}_{714}) = \text{gcd}(\underbrace{21, 13}_{273})$$

$$= \text{gcd}(\underbrace{13, 8}_{104}) = \text{gcd}(\underbrace{8, 5}_{40}) = \text{gcd}(\underbrace{5, 3}_{15})$$

$$= \text{gcd}(\underbrace{3, 2}_{6}) = \text{gcd}(\underbrace{2, 1}_{2}) = \underbrace{1}_{1}$$

Observation: After each iteration, the product of the arguments is reduced by at least 1 / 2.

- If this is indeed true for all inputs, then we can conclude that on input a and b, the algorithm iterated at most log(ab) = log a + log b times, and hence the running time of the algorithm is O(log a + log b).
- *Here, log(x) = $\log_2$(x) by convention of theoretical computer science.*

Observation: Why?

- Suppose that given input a_0 and b_0, the algorithm iterations k times before it stops.
- Let a_i and b_i be the values stored at variable a and b at the i-th iterations.
- Let $p_0 = a_0 b_0$, $p_1 = a_1 b_1$, …, $p_k = a_k b_k$.

- If our observation is true, then we have

$$p_1 < p_0 / 2; \quad p_2 < p_1 / 2; \quad \ldots \quad p_k < p_{k-1} / 2.$$

- Rearranging gives

$$p_0 > 2\ p_1 > 2 \times 2\ p_2 = 2^2 p_2 > \ldots > 2^i\ p_i > \ldots > 2^k p_k > 2^k,$$

- Thus,

$$\log (a_0\ b_0) > k \qquad \text{or} \qquad k < \log a_0 + \log b_0.$$

Proof of the Observation

Observation: For $1 \le i \le k$, $p_i = a_i\, b_i < \frac{a_{i-1}\, b_{i-1}}{2} = \frac{p_{i-1}}{2}$.

Proof. Consider the following two cases:

Case 1: $b_{i-1} > \frac{a_{i-1}}{\mathbf{2}}$.

- $a_{i-1} \bmod b_{i-1} \le a_{i-1} - b_{i-1} < a_{i-1} - \frac{a_{i-1}}{\mathbf{2}} = \frac{a_{i-1}}{2}$
- $p_i = (a_{i-1} \bmod b_{i-1}) \times b_{i-1} < \frac{a_{i-1}}{2} \times b_{i-1} = \frac{a_{i-1}\, b_{i-1}}{2} = \frac{p_{i-1}}{2}$

Case 2: $b_{i-1} \le \frac{a_{i-1}}{\mathbf{2}}$.

- $a_{i-1} \bmod b_{i-1} < b_{i-1} \le \frac{a_{i-1}}{\mathbf{2}}$
- $p_i = (a_{i-1} \bmod b_{i-1}) \times b_{i-1} < \frac{a_{i-1}}{2} \times b_{i-1} = \frac{a_{i-1}\, b_{i-1}}{2} = \frac{p_{i-1}}{2}$