An outline of the book

1. Analyzing algorithms
2. Data abstraction
3. Recursion and induction
4. Sorting
5. Selection and adversary arguments
6. Dynamic sets and searching
7. Graphs and graph traversals
8. Graph optimisation and greedy algorithms
9. Transitive closure
10. Dynamic programming
11. String matching
12. Polynomials and matrices
13. NP-complete problems
14. Parallel algorithms
1 Chapter 1

1.1 Sections 1, 2, and 3

Read this material on your own. All of this is either background material or else is material analogous to what you have already learned.

1.2 Section 4—What are we measuring?

• Let’s count the right things
• We admit that not all algorithms for solving the same problem necessarily will involve the same basic operations, and that this will complicate the problem of analysis.
• Worst case versus average case analysis
• The constants matter as well as the asymptotics
• Space can matter and is usually not counted in the asymptotics
• Implementation issues can matter and are usually not counted in the asymptotics.
1.3 Measuring the right things is not always trivial

Consider the greatest common divisor function.

**Definition 1.1.** The greatest common divisor $g$ of two integers $a$ and $b$, written $g = \gcd(a, b)$, is the largest positive integer that divides both $a$ and $b$ evenly.

**Examples:** $\gcd(0, n) = n$, $\gcd(1, n) = 1$, $\gcd(21, 35) = 7$

**Convention:** The gcd is always taken to be positive, even if $a$ and $b$ are both negative. The issue of signs is dealt with as a separate matter.

**Proposition 1.2.** If $a = \prod p_i^{e_i}$ and $b = \prod p_i^{f_i}$ are the canonical factorings of $a$ and $b$ into products of prime numbers, then $\gcd(a, b) = \prod p_i^{\min(e_i, f_i)}$

**Proposition 1.3 (The major method).** For any integers $a$, $b$, and $k$, we have

\[
\gcd(a, b) = \gcd(b, a) = \gcd(-a, b) = \gcd(-a, -b) = \gcd(a + b, b) = \gcd(a - b, b) = \gcd(a, a \ast k + b)
\]
Euclid’s algorithm

Input a and b
Preprocess signs and zeros to guarantee $a > 0$ and $b > 0$

while $(0 != \text{remainder})$
  \[ \text{quotient} = \text{IntegerPart}(a/b) \]
  \[ \text{remainder} = a - \text{quotient} \times b \]
  \[ a = b \]
  \[ b = \text{remainder} \]
At end, the gcd is $b$

Example:

\[
gcd(114, 90) = gcd(90, 114 - 1 \times 90) = gcd(90, 24) \\
= gcd(24, 90 - 3 \times 24) = gcd(24, 18) \\
= gcd(18, 24 - 1 \times 18) = gcd(18, 6) \\
= gcd(6, 18 - 3 \times 6) = gcd(6, 0) = 6
\]

Measuring the cost: The costly part of this algorithm is the division, especially if the operands are multiprecise. Less costly, but not zero compared to the divisions, are the multiplications.

Analysis (handwaving version): The worst case for Euclid is the gcd of two successive Fibonacci numbers, for which the quotients will all be 1 and the operands will decrease as slowly as possible. Since Fibonacci numbers basically double in size with every iteration, if we have $a = F_n$ and $b = F_{n-1}$ we will have operands essentially (i.e., to within a first-order wave of the hands) of $n$ bits and thus $n = \lg(F_n)$ divisions.
An Improved (?) algorithm

Input a and b
Preprocess signs and zeros to guarantee a >= b > 0
while(0 != remainder)
  remainder = a - b
  if(remainder > b)  // meaning quotient > 1
    remainder = remainder - b
  if(remainder > b)  // meaning quotient > 2
    remainder = remainder - b
  if(remainder > b)  // meaning quotient > 3
    remainder = a - b * IntegerPart(a/b)  // PUNT
  endif
  endif
  a = b
  b = remainder
endwhile
At end, the gcd is b

Example:

\[\gcd(114, 90) = \gcd(24, 90) = \gcd(90, 24)\]
\[= \gcd(66, 24) = \gcd(42, 24) = \gcd(18, 24) = \gcd(24, 18)\]
\[= \gcd(6, 18) = \gcd(18, 6)\]
\[= \gcd(12, 6) = \gcd(6, 6) = \gcd(0, 6) = 6\]
Theorem 1.4. Given random integers \(a\) and \(b\) of size bounded by \(N\), then the quotients in Euclid’s algorithm are 1, 2, 3, and 4 with frequencies about 0.41504, 0.16992, 0.09311, and 0.05890. Thus, about \(0.41504 + 0.16992 + 0.09311 \approx 2/3\) of the time the quotients are \(\leq 3\).

Measuring the cost:

Now what do we measure to compare this against straight Euclid? We have eliminated divisions, but added a nontrivial number of subtractions.
Yet Another Algorithm

Input a and b
Preprocess signs and zeros to guarantee a > 0 and b > 0
Compute $g_1$ as the 2-part of the gcd (*)
Preprocess to guarantee a and b both odd and $a \geq b$ (**) 
while (0 != difference) 
  difference = a - b 
  difference = difference shifted right until odd (***)
  \{a, b\} = {max(b, difference), min(b, difference)} (****)
endwhile
At end, the gcd is $g_1 \times b$ (*****)

Example:

$$\text{gcd}(1110010_2, 1011010_2) = 2 \cdot \text{gcd}(111001_2, 101101_2) = 2 \cdot \text{gcd}(101101_2, 1100_2)$$
$$= 2 \cdot \text{gcd}(101101_2, 11_2) = 2 \cdot \text{gcd}(101010_2, 11_2) = 2 \cdot \text{gcd}(10101_2, 11_2)$$
$$= 2 \cdot \text{gcd}(10010_2, 11_2) = 2 \cdot \text{gcd}(1001_2, 11_2) = 2 \cdot \text{gcd}(110_2, 11_2)$$
$$= 2 \cdot \text{gcd}(11_2, 11_2) = 2 \cdot \text{gcd}(0, 11_2) = 2 \cdot 11_2 = 6$$

(*) If $a$ written in binary ends in $n$ zeros, then $2^n$ exactly divides $a$. Similarly $b$ for $m$ terminating zeros. The 2-part of the gcd is $2^e$, where $e = \min(n, m)$.

(**) After we get the 2-part, we shift $a$ and $b$ right to make them both odd; we don’t need the 2s any more.

(***) If $a$ and $b$ are odd, their gcd is odd. But their difference is even, and the even part of the difference is irrelevant. We shift it away to get smaller numbers.
(****) These two assignments are done in parallel.

(***** ) The final gcd is the product of the odd part and the even part.

**Measuring the cost:** Now we're really in trouble. This is algorithmically incommensurable with either previous algorithm unless we can accurately count bit-operations.

**Worst case:** We lose only one bit in the (*** ) shift. Fibonacci numbers again, and \( \lg N \) iterations of the loop.
Section 5–Asymptotics

Definitions

• **Big Oh**: (Landau, 1890s) We write

\[
f(x) = O(g(x))
\]

if \( \exists \) constants \( c \) and \( C \) such that \( x > c \implies |f(x)| \leq C \cdot g(x) \).

**NOTE**: The text omits the absolute value symbol. This is a rare lapse in mathematical rigor. The correct notation requires the absolute value. HOWEVER, in the context of algorithm analysis, the functions we’re looking at are counting the amount of work involved and are therefore always positive. So in this context the text won’t be in error. Things that are asserted to be \( O(.) \) in the text will in fact be \( O(.) \) in a rigorous sense.

• **Little Oh**: (Landau, 1890s) We write

\[
f(x) = o(g(x))
\]

if \( |f(x)/g(x)| \to 0 \) as \( \to \infty \).

• **Big Omega**: (Hardy, about 1915) We write

\[
f(x) = \Omega(g(x))
\]

if \( f(x) \) is not \( o(g(x)) \), that is: there exists a sequence \( x_1, x_2, \ldots, x_n, \ldots \), tending to \( \infty \), such that for any fixed constant \( C \), there exists a constant \( c \) such that \( x_i > c \implies |f(x_i)| \geq C \cdot g(x_i) \).
• **Knuth’s Big Omega**: (Knuth’s corrupted notation, about 1975) We write

\[ f(x) = \Omega_{K}(g(x)) \]

if for any fixed constant \( C \) there exists a constant \( c \) such that

\[ x > c \implies |f(x)| \geq C \cdot g(x). \]

**NOTE**: Note the difference between the traditional definition of \( \Omega(.) \) and the modern corruption of Knuth. The text is one of the few places in the “computer science” literature in which the correct notation is acknowledged to exist. The difference between the two can lead to some confusion.

**NOTE**: Note also that in the context of algorithm analysis, sometimes the two notations can agree in a restricted sense. Graph algorithms, for example, usually have running times measured in terms of the number of nodes, \( n \), in the graph. If for every \( n \) there exists a pathological worst case graph, then it could be that the worst case example for each \( n \) would serve as the infinite sequence of points with \( f(n) \geq C \cdot g(n) \). Even though not all the running times on \( n \) nodes had this bad running time, one could, by phrasing the statement properly, say that \( \Omega(.) = \Omega_{K}(.) \) for this algorithm.

• **Theta**: (??) We write

\[ f(x) = \Theta(g(x)) \]

if there exist constants \( c, C_1, \) and \( C_2 \) such that

\[ x > c \implies C_1 \cdot g(x) \leq |f(x)| \leq C_2 \cdot g(x). \]
1.4 The Meaning of Asymptotics

Assume a computer doing “ops” at 1GHz, that is, at $10^9$ per second. There are $28 \cdot 24 \cdot 3600 = 2419200$ seconds in a 28-day month, so one month is $2419200000 \cdot 10^6$ ops. How big a problem is this? The following numbers (times $10^6$) represent the size $n$ solvable with an algorithm whose running time is the first column, assuming “one op per value of $n$.” The second part of the table normalizes against an exponential algorithm by dividing by the boldface entry.

<table>
<thead>
<tr>
<th></th>
<th>1 week</th>
<th>2 weeks</th>
<th>3 weeks</th>
<th>4 weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>604800000.0</td>
<td>1209600000.0</td>
<td>1814400000.0</td>
<td>2419200000.0</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>33913481.7</td>
<td>66923668.3</td>
<td>97931484.8</td>
<td>128938074.4</td>
</tr>
<tr>
<td>$n^2$</td>
<td>24592.7</td>
<td>34779.3</td>
<td>42595.8</td>
<td>49185.4</td>
</tr>
<tr>
<td>$n^3$</td>
<td>845.6</td>
<td>1065.4</td>
<td>1219.6</td>
<td>1342.3</td>
</tr>
<tr>
<td>$n^4$</td>
<td>156.8</td>
<td>186.5</td>
<td>206.4</td>
<td>221.8</td>
</tr>
<tr>
<td>$n^5$</td>
<td>57.1</td>
<td>65.5</td>
<td>71.1</td>
<td>75.3</td>
</tr>
<tr>
<td>$2^n$</td>
<td>29.2</td>
<td>30.2</td>
<td>30.8</td>
<td>31.2</td>
</tr>
<tr>
<td>$e^n$</td>
<td>20.2</td>
<td>20.9</td>
<td>21.3</td>
<td>21.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1 week</th>
<th>2 weeks</th>
<th>3 weeks</th>
<th>4 weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>20732292.2</td>
<td>41464584.4</td>
<td>62196876.6</td>
<td>82929168.8</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>1162540.0</td>
<td>2294115.5</td>
<td>3357050.5</td>
<td>4419943.5</td>
</tr>
<tr>
<td>$n^2$</td>
<td>843.0</td>
<td>1192.2</td>
<td>1460.2</td>
<td>1686.1</td>
</tr>
<tr>
<td>$n^3$</td>
<td>29.0</td>
<td>36.5</td>
<td>41.8</td>
<td>46.0</td>
</tr>
<tr>
<td>$n^4$</td>
<td>5.4</td>
<td>6.4</td>
<td>7.1</td>
<td>7.6</td>
</tr>
<tr>
<td>$n^5$</td>
<td>2.0</td>
<td>2.2</td>
<td>2.4</td>
<td>2.6</td>
</tr>
<tr>
<td>$2^n$</td>
<td>1.0</td>
<td>1.0</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>$e^n$</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
</tr>
</tbody>
</table>
2 Chapter 2

Read this material on your own.

Apart from the actual material in this chapter, the major lesson to be taken from this chapter is to be cognizant of what it is that we are counting when we do an analysis of an algorithm. If everything is described using elementary structures and operations, then we will know exactly what we’re counting, but we will also be forced to be more inelegant and detailed than might be necessary. Using the more fancy structures can improve the description, but if we aren’t careful, the encapsulating data structure can hide some of the computational complexity and our analysis will be flawed.
3 Chapter 3 (and parts of chapter 1 §5)

3.1 Some basic formulas

\[
\sum_{i=1}^{n} i = \frac{n(n + 1)}{2} = O(n^2)
\]

\[
\sum_{i=1}^{n} i^2 = \frac{n(n + 1)(2n + 1)}{6} = O(n^3)
\]

\[
\sum_{i=1}^{n} i^3 = \frac{n^2(n + 1)^2}{4} = O(n^4)
\]
3.2 Some basic truths

- Assume an algorithm starts with \( n \) active items. With each iteration of an outer loop all the active items are scanned and the number of active items is diminished by 1. Then the algorithm will take at least \( O(n^2) \) time.

The canonical example of such an algorithm is a bubble sort or insertion sort. With each iteration of the outer loop one item is put in its proper place and the number of elements remaining to be sorted drops by one. Thus \( n \) iterations (one for each of the items to be sorted), and the \( i \)-th iteration takes (worst case) \( n - i \) steps.

```cpp
for(i = 1; i <= n-1; i++)
{
    for(j = i+1; j <= n; j++)
    {
        if(a[i] > a[j])
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
}
```
Note that the time could actually be greater than $O(n^2)$. The actual running time is

$$
\sum_i (n \text{ iterations}) \cdot \frac{n-i \text{ steps}}{\text{per iteration}} \cdot \frac{f(n) \text{ ops}}{\text{per step}}
$$

In the case of the bubble sort, each item is scanned in constant time and $f(n) = \text{constant}$. But if for some reason the time depends on $n$, then the total time will be even worse than this.

- Assume an algorithm starts with $n$ active items. Each iteration of the outer loop takes $f(n)$ time and recursively cuts the active list into $k$ pieces each of size $n/k$. Then the running time of the overall algorithm looks like this.

$$
T(n) = k \cdot T\left(\frac{n}{k}\right) + f(n)
$$

$$
= k \cdot \left(k \cdot T\left(\frac{n}{k^2}\right) + f(n/k)\right) + f(n)
$$

$$
= k^2 \cdot T\left(\frac{n}{k^2}\right) + f(n) + k f(n/k)
$$

$$
= k^3 \cdot T\left(\frac{n}{k^3}\right) + f(n) + k f(n/k) + k^2 f(n/k^2)
$$

$$
= \ldots
$$

$$
= k^{\log_3 n} T(1) + \sum_{i=1}^{\log_3 n} k^i f\left(\frac{n}{k^i}\right)
$$

$$
= n \cdot T(1) + \sum_{i=1}^{\log_3 n} k^i f\left(\frac{n}{k^i}\right)
$$

Now, if $f(n) = n$, this comes up $n \log n$ because each term is $n$ and there are $\log n$ terms.
3.3 More generally, some basic truths

- (Pages 137-140)
- In a recursive situation, we have

\[ T(n) = bT(n/c) + f(n) = \sum_{d=0}^{\log n / \log c} \theta^d f(n/c^d) \]

If we let \( E = \log b / \log c \), then
- If \( f(n) = O(n^{E-\varepsilon}) \), then \( T(n) = \Theta(n^E) \)
- If \( f(n) = \Theta(n^E) \), then \( T(n) = \Theta(f(n) \log n) \)
- If \( f(n) = \Omega_k(n^{E+\varepsilon}) \), and \( f(n) = O(n^{E+\delta}) \), for some \( \delta \geq \varepsilon \), then

\[ T(n) = \Theta(f(n)). \]

- In the “chip” approach, we have

\[ T(n) = bT(n - c) + f(n) \]

then if \( b = 1 \), we have

\[ T(n) = T(n - c) + f(n) = T(n - 2c) + f(n) + f(n - c) \]

\[ = \sum_{d=0}^{n/c} f(n - cd) = \sum_{d=0}^{n/c} f(ch) \approx \frac{1}{c} \int_0^n f(x)dx \]

So this could in fact work.
- But if in the chip approach, we have \( b \geq 2 \), then this is at least exponential even if \( f(k) = 1 \).