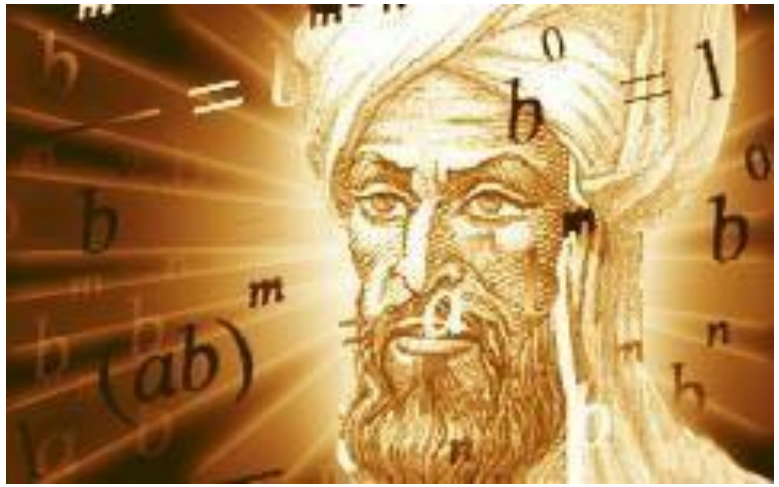
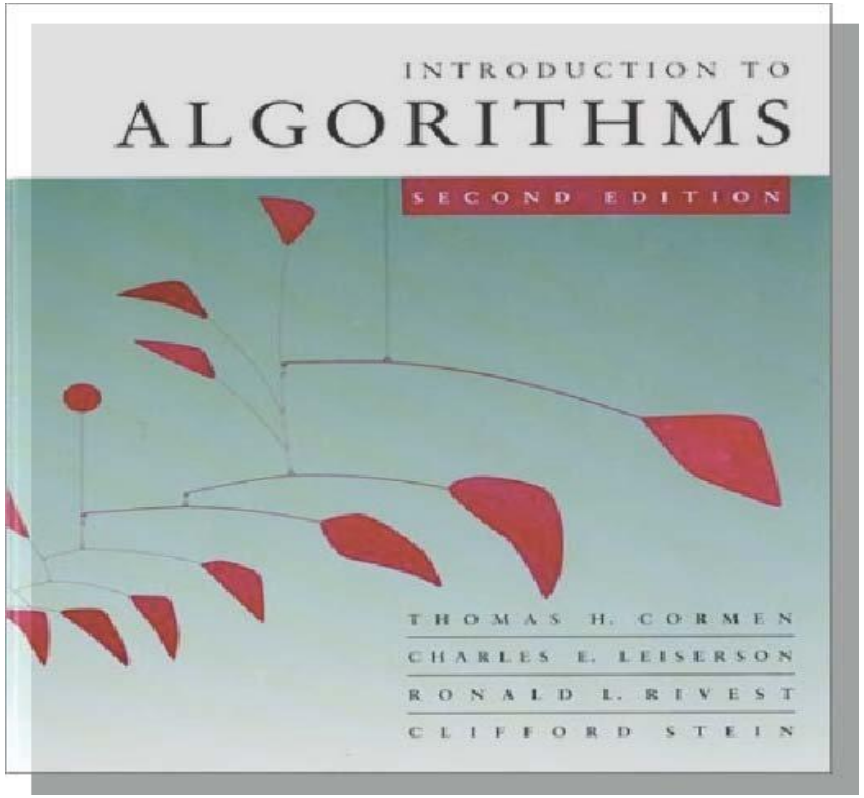
Father of Algorithms



Material

- Lecture slides.
- Text Book by Cormen et. al.
- Tutorials on Web.



Material Outline

- Introduction.
- Sorting Algorithms and Analysis Techniques.
- Greedy Algorithms.
- Branch and Bound Technique.
- Graph Algorithms.
- Hashing.
- Dynamic Programming Paradigm.

Lecture #1

- Introduction.
- Mathematical Background.
- A simple Example.

What is an algorithm?

- Our text defines an ***algorithm*** to be any well-defined computational procedure that takes some values as ***input*** and produces some values as ***output***. Like a cooking recipe, **an algorithm provides a step-by-step method for solving a computational problem.**
- Unlike programs, **algorithms are not dependent on a particular programming language, machine, system, or compiler.** They are mathematical entities, which can be thought of as running on some sort of *idealized computer* with an infinite random access memory and an unlimited word-size.
- **Algorithm design is all about the mathematical theory behind the design of good programs.**

Why study algorithm design?

- **Programming is a very complex task**, and there are a number of aspects of programming that make it so complex.
- The first is that most programming projects are very **large**, requiring the coordinated efforts of **many people.**
- The next is that many programming projects involve storing and accessing **large quantities of data** efficiently.
- The last is that many programming projects involve solving **complex computational problems**, for which simplistic or naive solutions may not be efficient enough.
- The complex problems may involve numerical data but often they involve **discrete data.** This is where the topic of algorithm design and analysis is important.

Mathematical Basics

- **Polynomial:** Powers of n, such as n^5 and $\sqrt{n} = n^{1/2}$.
- **Polylogarithmic:** Powers of log n, such as $(\log n)^7$. We will usually write this as $\log^7 n$.
- **Exponential:** A constant (not 1) raised to the power n, such as 3^n.
- An important fact is that **polylogarithmic functions are strictly asymptotically smaller than polynomial** function, which are strictly asymptotically smaller than exponential functions (assuming the base of the exponent is bigger than 1).
- For example, if we let mean "asymptotically smaller" then,

$$\log^a n < n^b < c^n$$

Mathematical Basics

- **Logarithm Simplification:** It is a good idea to first simplify terms involving logarithms. For example, the following formulas are useful.

  Here *a; b; c* are constants:

$$\begin{aligned} \log_b n &= \frac{\log_a n}{\log_a b} = \Theta(\log_a n) \\ \log_a(n^c) &= c \log_a n = \Theta(\log_a n) \\ b^{\log_a n} &= n^{\log_a b}. \end{aligned}$$

- **Avoid using log *n* in exponents**. The last rule above can be used to achieve this. For example, rather than saying $3^{\log_2 n}$,

  express this as $n^{\log_2 3} \approx n^{1.585}$.

Mathematical Basics

- **Summations:** they naturally arise in the **analysis of iterative algorithms.** Also, more complex forms of analysis, such as **recurrences,** are often solved by reducing them to summations.
- Solving a summation means reducing it to a ***closed form formula***, that is, one having no summations, recurrences, integrals, or other complex operators.
- In algorithm design it is often not necessary to solve a summation exactly, since an **asymptotic approximation** or close upper bound is usually good enough. Here are some common summations and some tips to use in solving summations.

Mathematical Basics

- **Constant Series:** For integers *a* and *b*,

$$\sum_{i=a}^{b} 1 = \max(b - a + 1, 0).$$

- Notice that when $b = a - 1$, there are no terms in the summation (since the index is assumed to count upwards only), and the result is 0. Be careful to check that $b \geq a-1$ before applying this formula blindly.

- **Arithmetic Series:** For $n > 0$,

$$\sum_{i=0}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

- This is **$\Theta(n^2)$.** As we will see later in more details!

Mathematical Basics

- **Geometric Series:** Let $x \neq 1$ be any constant (independent of n), then for $n \geq 0$,

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}.$$

- If **$0 < x < 1$ then this is $\Theta(1)$.**
- **If $x > 1$, then this is $O(x^n)$,** that is, the entire sum is proportional to the last element of the series.

Mathematical Basics

- **Quadratic Series:** For $n \geq 0$,

$$\sum_{i=0}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{2n^3 + 3n^2 + n}{6}.$$

- **Harmonic Series:** This arises often in probabilistic analyses of algorithms. It does not have an exact closed form solution, but it can be closely approximated. For $n \geq 0$,

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = (\ln n) + O(1).$$

Mathematical Basics

- **Summations with general bounds:** When a summation does not start at the 1 or 0, as most of the above formulas assume, you can just split it up into the difference of two summations. For example, for $1 \leq a \leq b$

$$\sum_{i=a}^{b} f(i) = \sum_{i=0}^{b} f(i) - \sum_{i=0}^{a-1} f(i).$$

- **Linearity of Summation:** Constant factors and added terms can be split out to make summations simpler.

$$\sum (4 + 3i(i-2)) = \sum 4 + 3i^2 - 6i = \sum 4 + 3\sum i^2 - 6\sum i.$$

Mathematical Basics

- **Approximate using integrals:** Integration and summation are closely related. (Integration is in some sense a continuous form of summation.) Here is a handy formula. Let $f(x)$ be any ***monotonically*** *increasing function* (the function increases as x increases).

$$\int_0^n f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)dx.$$

Example: Right Dominant Elements

- As an example of the use of summations in algorithm analysis, consider the following simple problem.
- We **are given a list *L* of numeric values**. We say that an element of *L* is *right dominant* if it is strictly larger than all the elements that follow it in the list. Note that the last element of the list is always right dominant, as is the last occurrence of the maximum element of the array.
- For example, consider the list **L = (10; 9; 5; 13; 2; 7; 1; 8; 4; 6; 3)**
- The sequence of **right dominant** elements are **(13; 8; 6; 3).**

Example: Right Dominant Elements

- In order to make this more concrete, we should think about **how *L* is represented**. It will make a difference whether *L* is represented as an **array (allowing for random access),** a **doubly linked** list (**allowing for sequential access in both directions**), or a **singly linked list (allowing for sequential access in only one direction).**
- Among the three possible representations, the **array representation** seems to yield the simplest and clearest algorithm.
- However, we will design the algorithm in such a way that it only performs **sequential scans**, so it could also be implemented using a **singly linked** or **doubly linked list.**

Example: Right Dominant Elements

- This is common in algorithms. Chose your representation to make the algorithm as simple and clear as possible, but give thought to how it may actually be implemented.
- Remember that **algorithms are read by humans, not compilers**. We will assume here that the array *L* of size *n* is indexed from 1 to *n*.
- The most straight forward solution is introduced next slide.

Example: Right Dominant Elements

- A basic solution:

```
// Input: List L of numbers given as an array L[1..n]
// Returns: List D containing the right dominant elements of L
RightDominant(L) {
    D = empty list
    for (i = 1 to n)
        isDominant = true
        for (j = i+1 to n)
            if (A[i] <= A[j]) isDominant = false
        if (isDominant) append A[i] to D
    }
    return D
}
```

Example: Right Dominant Elements

- The time spent in this algorithm is dominated (no pun intended) by the time spent in the inner (*j*) loop.
- On the *i*th iteration of the outer loop, the inner loop is executed from *i* + 1 to *n*, for a total of *n* − (*i* + 1) + 1 = *n*−*i* times. (Recall the rule for the constant series above.) Each iteration of the inner loop takes constant time.
- Thus, up to a constant factor, the running time, as a function of *n*, is given by the following summation:

$$T(n) = \sum_{i=1}^{n} (n - i).$$

Example: Right Dominant Elements

- To solve this summation, let us expand it, and put it into a form such that the above formulas can be used.

$$\begin{aligned} T(n) &= (n-1) + (n-2) + \ldots + 2 + 1 + 0 \\ &= 0 + 1 + 2 + \ldots + (n-2) + (n-1) \\ &= \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}. \end{aligned}$$

- This **running time is of order n^2** (review the arithmetic series: is $\Theta(n^2)$)

Example: Right Dominant Elements

- One more faster solution: there is a simple ***O*(*n*)** time algorithm for this problem.
- In a **one single scan for the list from right–to-left** and finding the maximum found values (called **records**).
- In other words, recording the best (maximum) element found so far…
- The list of the best elements are the solution.
- Can you write this code?
- As you see, for one problem there are many solutions that are different in speed of execution and occupied space.