# Lecture 1: Introduction

Last updated: Jan 18, 2021

References:

- The Algorithm Design Manual, Skiener, Chapter 1
- Algorithm Design Techniques, Programming Pearls, Jon Bentley, ACM 1984
- Algorithms, Jeff Erickson. Chapter 0

What is an algorithm?

An algorithm is an **explicit**, **precise**, **unambiguous**, **mechanically executable** sequence of **elementary instructions**, intended to accomplish a **specific purpose**.

- Explicit: can be described in words and mathematical notations
- Precise: only one interpretation
- Mechanically executable: only use elementary instructions that machines support
- Specific purpose: what does it accomplish?

# Problem vs Problem Instance





To be interesting, an algorithm must solve a **general** problem. An algorithmic problem is specified by describing the complete set of **instances** it must work on and of its output.

The distinction between problem and problem instance is fundamental.

For example, the algorithmic problem known as *sorting* can be described:

*Problem*: sorting
*Input*: A sequence of $n$ keys $a_1, \ldots, a_n$.
*Output*: The permutation (reordering) of the input sequence such that $a_1' \leqslant a_2' \leqslant \cdots \leqslant a_n'$.

An instance of the sorting problem might be an array of integers {2,3,1}.

Consider the following "procedure":

```
MagicSort(a[1..3]):
    Output {1,2,3}
```

It correctly "sorts" the **problem instance** {2,3,1}, but fails on pretty much every other instances (defined by inputs).

This procedure is not an (correct) algorithm. An algorithm must solve a general problem (all possible instances), instead of one or part.

Most of the time, it's not clear whether the algorithm actually does solve all instances, so we must supply a **proof** of correctness of the algorithm to make it useful. The proof is a certification of the correctness of an algorithm.

# Bad Example





Here's a curious algorithm:

```
BeAMillionareAndNeverPayTaxes():
    Get a million dollars
    if the tax man shows up
        say "I forgot"
```

What's the problem with this "algorithm"?

# Example: Multiplication





We would like to multiply two positive integers. First let's figure out what data representation we use. We mostly use the decimal positional notation: basically 123 means $1 \times 100 + 2 \times 10 + 3$. Formally, we represent an integer $x$,y as array of decimal digits $X[0 \ldots m-1]$, $Y[0 \ldots n-1]$

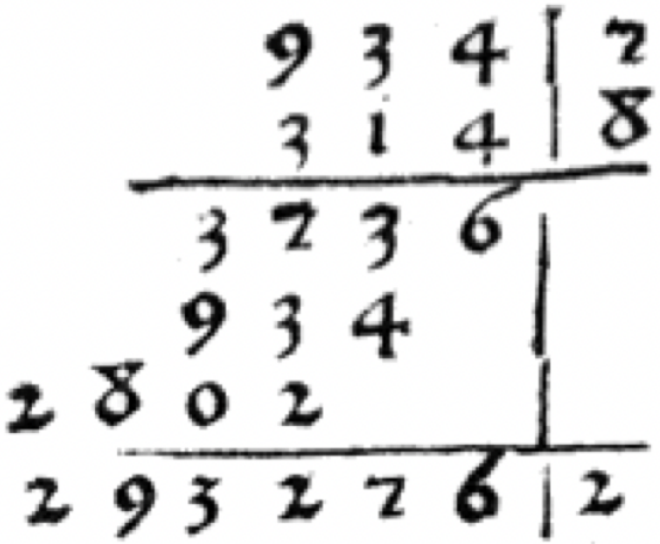
$$x = \sum_{i=0}^{m-1} X[i] \times 10^i, y = \sum_{j=0}^{n-1} Y[j] \times 10^j$$

We would like to compute $z = x \cdot y$ which is represented $Z[0 \ldots m+$

$n-1]$:

$$z = \sum_{k=0}^{m+n-1} Z[k] \times 10^k$$

I heard that Americans are most familiar with the Lattice algorithm, which is illustrated as




Why is it correct? Can we give a proof of the Lattice multiplication

algorithm? Hint:

$$z = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} X[i]Y[j] \times 10^{i+j}$$

The algorithm is based on **elementary operations**: single digit multiplication (can be done by looking up in a table, from memory of a computer, etc) and addition.

Now we know it's correct, the next question is: is it efficient?

First we derive the time complexity in terms of the elementray operations. By some accounting, we see that the Lattice multiplication algorithm takes $O(mn)$ steps (single digit multiplication/addition).

There's an even older and maybe simpler algorithm goes by many names including **peasant multiplication** which reduces to four operations; 1) determining parity; 2) addition; 3) duplation (doubling); 4) mediation (halving).

```
PeasantMultiply(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return prod
```

| $x$ | $y$ | | $prod$ |
|---|---|---|---|
| | | | 0 |
| 123 | + 456 | = | 456 |
| 61 | + 912 | = | 1368 |
| 30 | ~~1824~~ | | |
| 15 | + 3648 | = | 5016 |
| 7 | + 7296 | = | 12312 |
| 3 | + 14592 | = | 26904 |
| 1 | + 29184 | = | **56088** |

Now why is this algorithm correct? It's based on the following recursive identity:

$$xy = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \, (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \, (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

This is an recursive algorithm! (implemented as iterations).

Now what's the time complexity of this algorithm?

Without loss of generality, assume $x \leqslant y$. Clearly, the algorithm does $\log x$ parity, addition, and mediation. What's the cost of each operation?

Assuming any reasonable place-value (positional) representation of numbers (binary, decimal, Roman numeral, bead positions on abacus, etc)

each operation requires $O(\log x + \log y) = O(\log y)$ (because $x$ has $O(\log x)$ digits). Therefore the total time complexity is $O(\log x \cdot \log y) = O(mn)$ time, the same as the Lattice algorithm!

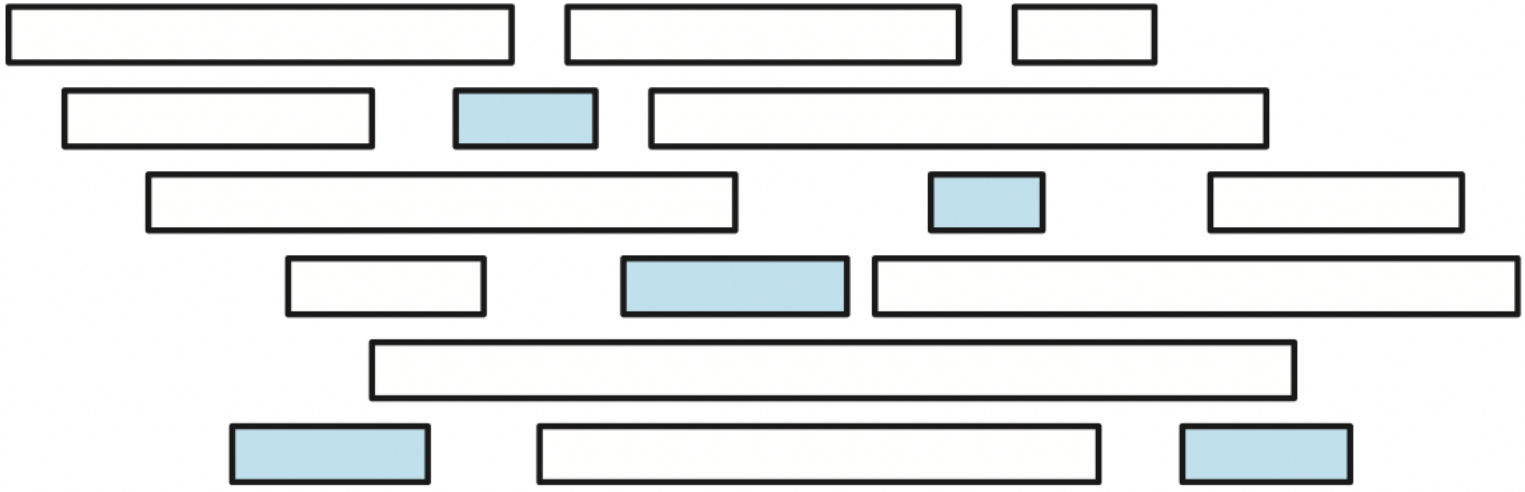
This algorithm is arguably easier for humans to execute, because the basic operations are simpler (if you can't remember single digit multiplication table!).

In fact, for binary representation, the peasant algorithm is identical as lattice mulitplication algorithm.

# Scheduling Classes







Suppose we are given $n$ classes with potentially overlapping lecture time. Class $i$ starts at time $S[i]$ and finishes at $F[i]$. Find the maximum number of non-overlapping classes.

We can visualize the class as blocks on time axis. The goal is to find the largest subset of blocks with no vertical overlap.

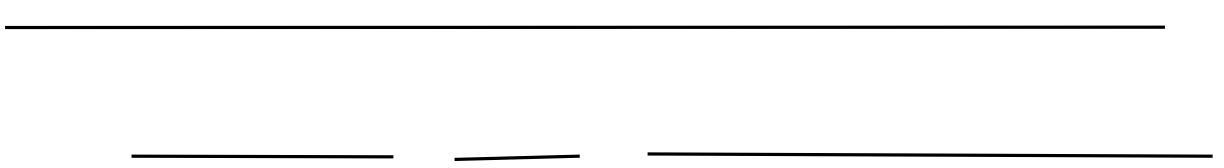
Think of an algorithm to solve it!

Let's try some ideas. Suppose the inputs are given by a set $I$ of intervals ([start,finish]).

```
EarliestClassFirst(I):
    P ← ∅
    while I is not empty:
        Put the earliest starting class j into set P
        Remove class j from set I
        Remove classes that overlaps class j from set I
    return P.size()
```

Is this correct? If not, can you give an example?

Counterexample



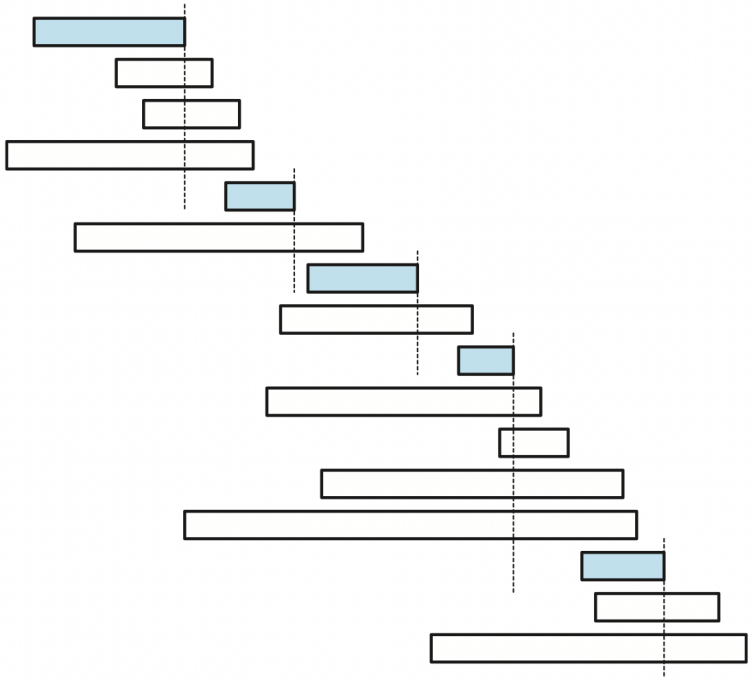
OK, let's try another one...

ShortestClassFirst($I$):
    $P \leftarrow \emptyset$
    while $I$ is not empty:
        Put the shortest class $j$ into set $P$
        Remove class $j$ from set $I$
        Remove classes that overlaps class $j$ from set $I$
    return $P$.size()

This is still not correct...

How about this one:

EarliestFinishClassFirst($I$):
    $P \leftarrow \emptyset$
    while $I$ is not empty:
        Put the earliest finishing class $j$ into set $P$
        Remove class $j$ from set $I$
        Remove classes that overlaps class $j$ from set $I$
    return $P$.size()



Can you find any counter example?

But how do you know this algorithm is correct? In later lectures, we are going to **prove** that the EarliestFinishClassFirst() algorithm is guarantteed to give an optimal solution. This example shows that algorithms are not obvious to be correct. Finding counterexample

proves the algorithm is incorrect; but absence of counterexample does not prove it. We need much stronger argument.

Basic proof techniques include:

- Mathematical induction and recursion. They go like this:
  1. Basic case is obvious correct: n=0, n=1, for example.
  2. If the statement is true for all $k \leqslant n-1$, then we show that the statement is also true for $k = n$.
  3. We proved that the statement is true for any $n$.

# Describing Algorithms





There are three primary ways to descreibe algorithms:

- English words
- Pseudocode
- Computer programming language

in the order of increasing precision, but in decreasing conciseness and generality. In this course, we are going to primarily use combination of English and pseudocode, according to this rule:

Our description of algorithm should *include* every detail necessary to fully specify the algorithm, prove its correctness, and analyze its running time. At the same time, it should *execlude* any details that are NOT necessary to fully specify the algorithm, prove its correcetness, or analyzing its running time.

Practically, **never** describe repeated operations informally, as in "Do this first, and do that, and **so on**...", or "repeat this process until [something]".

If it's a loop, write a loop with the initialization, loop body, and conditions. If it's recursion, write recursive function calls, and the base case for termination.

# Contents of This Course





This course consists of the following contents:

- Algorithm design techniques: recursion, divide and conquer, backtracking, greedy, randomized, ...
- Algorithm analysis: the proof of correctness, runtime/space complexity
- Selected illustrative or important algorithms & data structures: combinatoric problems, games, tree, graphs, networks, hash, disjoint sets...
- Problem solving: how to solve a problem? From distilling a specification of problem, to designing algorithm, to analyzing its correctness and performance, and to turn that into computer programs.

# The RAM Model of Computation





Why can we analyze the performance of algorithms independent of software systems and the machine? That's because we rely on the **RAM model of computation**, a simple abstract machine that captures the first-order performance characteristics of real machines:

- Each simple operation (+,-,*,/,if,call) takes 1 step. In reality, not all steps are equal; / is usually much slower than others.
- Loops and subroutine calls are not simple operation.
- Each random memory access takes 1 step. In reality, there's cache, data locality, and memory is not really random access.

By abstracting the machine, we can simply count the number of "steps" an algorithm needs. It captures the asymptotic behavior of an

algorithm very well. (faster algorithm is definitely faster on machine if problem size is sufficiently big).

## Worst case Complexity

Unless otherwise stated, we assume in this course that we are always referring to the worst case complexity. Why prefer worst case than average case, or best case?

- Worst case complexity is easy to analyze
- It's easy to use—no need to assume any specialness of input
- It's conservative—guaranttee to be working as described, if not better.

Average case complexity is sometimes more appropriate, especially in randomized algorithms. But it's more involved in analysis, because it needs assumptions on input probablistic distribution. We'll consider average case complexity on as-needed basis.

**Asymptotic notations** (review):

Big-O: upper bound of the functions, when problem size $n$ is big (asymptotic behavior of functions).

- $g(n) = O(f(n))$ means that there exists constant $C$ such that $g(n) \leqslant Cf(n)$, for all sufficiently larege $n$. Put it in another way, $g(n)$ grows no faster than $f(n)$.
- Similarly, $g(n) = \Omega(f(n))$ means lower bound.
- Similarly, $g(n) = \Theta(f(n))$ means lower and upper bounded.

# Asymptotic Dominance: (assuming 1 step takes 1ns)

| $n$ $f(n)$ | $\lg n$ | $n$ | $n \lg n$ | $n^2$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|
| 10 | 0.003 $\mu$s | 0.01 $\mu$s | 0.033 $\mu$s | 0.1 $\mu$s | 1 $\mu$s | 3.63 ms |
| 20 | 0.004 $\mu$s | 0.02 $\mu$s | 0.086 $\mu$s | 0.4 $\mu$s | 1 ms | 77.1 years |
| 30 | 0.005 $\mu$s | 0.03 $\mu$s | 0.147 $\mu$s | 0.9 $\mu$s | 1 sec | $8.4 \times 10^{15}$ yrs |
| 40 | 0.005 $\mu$s | 0.04 $\mu$s | 0.213 $\mu$s | 1.6 $\mu$s | 18.3 min | |
| 50 | 0.006 $\mu$s | 0.05 $\mu$s | 0.282 $\mu$s | 2.5 $\mu$s | 13 days | |
| 100 | 0.007 $\mu$s | 0.1 $\mu$s | 0.644 $\mu$s | 10 $\mu$s | $4 \times 10^{13}$ yrs | |
| 1,000 | 0.010 $\mu$s | 1.00 $\mu$s | 9.966 $\mu$s | 1 ms | | |
| 10,000 | 0.013 $\mu$s | 10 $\mu$s | 130 $\mu$s | 100 ms | | |
| 100,000 | 0.017 $\mu$s | 0.10 ms | 1.67 ms | 10 sec | | |
| 1,000,000 | 0.020 $\mu$s | 1 ms | 19.93 ms | 16.7 min | | |
| 10,000,000 | 0.023 $\mu$s | 0.01 sec | 0.23 sec | 1.16 days | | |
| 100,000,000 | 0.027 $\mu$s | 0.10 sec | 2.66 sec | 115.7 days | | |
| 1,000,000,000 | 0.030 $\mu$s | 1 sec | 29.90 sec | 31.7 years | | |

Dominance Rankings:

- $n!$
- $c^n$
- $n^3$,
- $n \log n$
- $n$
- $\sqrt{\mathsf{n}}$
- $\log \log n$
- $\log^2 n$