Picture is from the cover of the textbook CLRS.
Asymptotic Analysis
Algorithm is a step-by-step procedure carrying out a computation to solve an arbitrary instance of a problem.

E.g., sorting is a problem; a set of numbers form an instance of that and ‘solving’ involves creating a sorted output.
An **algorithm** is a step-by-step procedure carrying out a computation to solve an arbitrary instance of a problem.

- E.g., sorting is a problem; a set of numbers form an instance of that and ‘solving’ involves creating a sorted output.

A **program** is an implementation of an algorithm using a specific programming language.
An **algorithm** is a step-by-step procedure carrying out a computation to solve an arbitrary instance of a problem.

- E.g., sorting is a problem; a set of numbers form an instance of that and ‘solving’ involves creating a sorted output.

A **program** is an implementation of an algorithm using a specific programming language.

For a given problem (e.g., sorting) there can be several algorithms (e.g., Quicksort, Mergesort), and for a given algorithm (e.g., Quicksort) there can be several programs.

- Our focus in this course is on algorithms (not programs).
- How to implement a given algorithm relates to the art of **performance engineering** (writing a fast code)
Given a problem $P$, we need to

- Design an algorithm $A$ that solves $P$ (Algorithm Design)
Given a problem \( P \), we need to

- Design an algorithm \( A \) that solves \( P \) (Algorithm Design)
- Verify correctness and efficiency of the algorithm (Algorithm Analysis)
- If the algorithm is correct and efficient, implement it
  - If you implement something that is not necessarily correct or efficient in all cases, that would be a heuristic.
How should we evaluate different algorithms for solving a problem?

- In this course we are mainly concerned with amount of time it takes to solve a problem (this is called running time).
- We can think of other measures such as the amount of memory that is required by the algorithm.
- Other measures include amount of data movement, network traffic generated, etc.
How should we evaluate different algorithms for solving a problem?

- In this course we are mainly concerned with amount of **time** it takes to solve a problem (this is called **running time**)
- We can think of other measures such as the amount of **memory** that is required by the algorithm
- Other measures include amount of data movement, network traffic generated, etc.

The amount of time/memory/traffic required by an algorithm depend on the **size** of the problem

- Sorting a larger set of numbers takes more time!
Running Time of Algorithms

How to assess the running time of an algorithm?

Experimental analysis:

- Implement the algorithm in a program
- Run the program with inputs of different sizes
- Experimentally measure the actual running time (e.g., using \textit{clock()} from time.h)
Running Time of Algorithms

- How to assess the running time of an algorithm?

  **Experimental analysis:**
  - Implement the algorithm in a program
  - Run the program with inputs of different sizes
  - Experimentally measure the actual running time (e.g., using `clock()` from `time.h`)

- Shortcomings of experimental studies:
How to assess the running time of an algorithm?

**Experimental analysis:**
- Implement the algorithm in a program
- Run the program with inputs of different sizes
- Experimentally measure the actual running time (e.g., using `clock()` from `time.h`)

**Shortcomings of experimental studies:**
- We need to implement the program (what if we are lazy and those engineers are hard to employ?)
- We cannot test all input instances for the problem. What are the good samples? (remember the Morphy’s law)
- Many factors have impact on experimental timing, e.g., hardware (processor, memory), software environment (operating system, compiler, programming language), and human factors (how good was the programmer?)
Computational Models

We need to assess time/memory requirement of algorithms using models that

- Take into account all input instances
- Do not require implementation of the algorithms
- Are independent of hardware/software/programmer
Computational Models

- We need to assess time/memory requirement of algorithms using models that
  - Take into account all input instances
  - Do not require implementation of the algorithms
  - Are independent of hardware/software/programmer

- In order to achieve this, we:
  - Express algorithms using **pseudo-codes** (don’t worry about implementation)
  - Instead of measuring time in seconds, count the number of **primitive operations**
    - This requires an abstract **model of computation**
Random Access Machine (RAM) Model

- The random access machine (RAM):
  - Has a set of memory cells, each storing one ‘word’ of data.
  - Any access to a memory location takes constant time.
  - Any primitive operation takes constant time.
  - The running time of a program can be computed to be the number of memory accesses plus the number of primitive operations.

- Word-RAM is a RAM machine with the extra assumption that all values in our problem can ‘fit’ in a constant number of words (values are not too big).

- We often use Word-RAM model for analysis of algorithms
Random Access Machine (RAM) Model

- The **random access machine** (RAM):
  - Has a set of memory cells, each storing one ‘word’ of data.
  - Any **access to a memory location** takes constant time.
  - Any **primitive operation** takes constant time.
  - The **running time** of a program can be computed to be the number of memory accesses plus the number of primitive operations.

- Word-RAM is a RAM machine with the extra assumption that all values in our problem can ‘fit’ in a constant number of words (values are not too big).

- We often use Word-RAM model for analysis of algorithms

**Observation**

*RAM is a simplified model which only provides an approximation of a ‘real’ computer*
First, calculate the ‘cost’ (sum of memory accesses and primitive operations) for each line

- E.g., in line 5, there are 3 memory accesses and 3 primitive operations
Next, find the number of times each line is executed

- This depends on the input, we may consider best or worst case input
- Let $t_j$ be number of times the while loop is executed for inserting the $j$’th item.
  - In the best case, $t_j = 1$ and in the worst case $t_j = j$.
- Summing up all costs, in the best case we have
  \[ T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) = an - b \]
  for constant $a$ and $b$
- In the worst case, we have $T_n = \alpha n^2 + \beta n + \gamma$ for constant $\alpha, \beta, \gamma$
Primitive Operations

- RAM model implicitly assumes primitive operations have fairly similar running time

- Primitive operations:
  - basic integer arithmetic (addition, subtraction, multiplication, division, and modulo)
  - bitwise logic and bit shifts (logical AND, OR, exclusive-OR, negation, left shift, and right shift)

- Non-primitive operations:
  - exponentiation, radicals (square roots), logarithms, trigonometric functions (sine, cosine, tangent), etc.
Asymptotic Notations

Statement

So, we can express the cost (running time) of an algorithm A for a problem of size n as a function $T_A(n)$.

- How do we compare two different algorithms? say $T_A(n) = \frac{1}{1000} n^3$ and $T_B(n) = 1000n^2 + 500n + 200$.

- Summarize the time complexity using asymptotic notations!

- Idea: assume the size of input grows to infinity; identify which component of $T_A(n)$ contributes most to the grow of $T_A(n)$.

- As $n$ grows:
  - constants don’t matter (e.g., $T_A(n)$)
  - low-order terms don’t matter (e.g., $T_B(n)$)
Informally \( T_B(n) = O(T_A(n)) \) means \( T_B \) is asymptotically smaller than or equal to \( T_A \).

Is it sufficient to define \( O \) so that we have \( T_B(n) < T_A(n) \)?

- No because the inequality might not hold for small values of \( n \) which we don’t care about
- The two function might have constants we would prefer to ignore.
Informally $T_B(n) = O(T_A(n))$ means $T_B$ is asymptotically smaller than or equal to $T_A$.

Is it sufficient to define $O$ so that we have $T_B(n) < T_A(n)$?

No because the inequality might not hold for small values of $n$ which we don’t care about.

The two function might have constants we would prefer to ignore.

\[
f(n) \in O(g(n)) \iff \exists M > 0, \exists n_0 > 0 \text{ s.t. } \forall n > n_0 \text{, } f(n) \leq M \cdot g(n)
\]

\[
\text{ignore low-order terms} \quad \text{and} \quad \text{ignore constants}
\]
Let \( f(n) = 1000n^2 + 1000n \) and \( g(n) = n^3 \). Prove \( f(n) \in O(g(n)) \)