# CSCI3160 Design and Analysis of Algorithms (2025 Fall)

## Week 1: the RAM Computation Model

Instructor: Xiao Liang[1]

Department of Computer Science and Engineering
Chinese University of Hong Kong

[1]These slides are primarily based on materials prepared by Prof. Yufei Tao (please refer to Prof. Tao's version from 2024 Fall for the original content). Some modifications have been made to better align with this year's teaching progress, incorporating student feedback, in-class interactions, and my own teaching style and research perspective.

This is not a programming course.

**Main take-away message from this course**

Computer science is a branch of mathematics with its art reflected in the beauty of algorithms.

- Programming knowledge is not necessary to study algorithms.

**Many people believe that this branch holds the future of mankind.**

In mathematics (and hence, computer science) everything—including every term and symbol—must be rigorous.

Computer science is a subject where we

1. first define a **computation model**, which is a simple yet accurate abstraction of a computing machine;
2. then slowly build up a theory in this model from scratch.

Is there a universally good model?

- No. We need different models for different purposes.

Some examples:

- Finite Automata: Used in regular expressions, compiler design, and more. They are clean, relatively easy to understand, yet powerful enough for real-world applications.
- Turing Machines: Fundamental to computational complexity theory. They (or at least the vanilla version of them) are not sensitive to polynomial differences in time or space cost.
- Boolean/Arithmetic Circuits: Commonly used in cryptography and electronic design automation (EDA). These represent a non-uniform computational model.

How to Choose a Model for the Study of Algorithms? What Are the Criteria?

- Mechanically Implementable:
  - This ensures our original goal—implementing algorithms to solve real-world problems.
  - It helps avoid logical paradoxes in the theory you established.
- Sufficiently General: The model should be general enough to capture all natural steps involved in solving problems strategically. Example: Finite automata are limited in power—they cannot recognize the language $L = \{a^n b^n \mid n \geq 0\}$ because they lack memory to "count" and match the number of $a$'s and $b$'s. This makes them unsuitable for many algorithmic problems that require more expressive computational models.
- Sensitive Enough: The model should be able to capture fine-grained differences in resource usage like time and space. Example: We often need to distinguish between algorithms with time complexities like $O(n^2)$ and $O(n \log(n))$, especially in performance-critical applications. Turing machines, while foundational, treat all polynomial-time algorithms as equivalent in complexity theory (due to polynomial-time reductions), making them too coarse for such distinctions.

# Our Choice of the Computational Model

The Random Access Machine (RAM) model

A machine has a **memory** and a **CPU**.

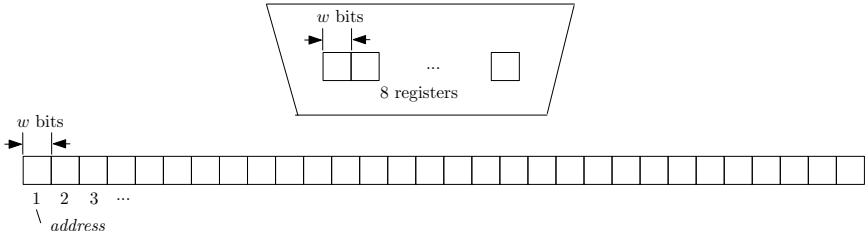
Memory

- An infinite sequence of **cells**, each of which contains the same number $w$ of bits.
- Every cell has an **address**: the first cell of memory has address 1, the second cell 2, and so on.

## The Random Access Machine (RAM) model

### CPU

- Contains a fixed number—8 in this course—of **registers**, each of which has $w$ bits (i.e., same as a memory cell).



## The Random Access Machine (RAM) model

### CPU

- Can do the following **atomic operations**:
  1. **(Register (Re-)Initialization)**
     Set a register to a fixed value (e.g., 0, $-1$, 100, etc.), or to the content of another register.

## The Random Access Machine (RAM) model

### CPU

- Can do the following **atomic operations**:
  2. **(Arithmetic)**
     Take the integers $a, b$ stored in two registers, calculate one of the following and store the result in a register:
     - $a + b$, $a - b$, $a \cdot b$, and $a/b$.

> Note: $a/b$ is "integer division", which returns an integer. For example, $6/3 = 2$ and $5/3 = 1$.

## The Random Access Machine (RAM) model

### CPU

- Can do the following **atomic operations**:
  3. **(Comparison/Branching)**
     Take the integers $a, b$ stored in two registers, compare them, and learn which of the following is true:
     - $a < b$, $a = b$, $a > b$.

## The Random Access Machine (RAM) model

### CPU

- Can do the following **atomic operations**:
  4. **(Memory Access)**
     Take a memory address $A$ currently stored in a register. Do one of the following:
     - Read the content of the memory cell with address $A$ into a designated register (overwriting the bits there).
     - Write the content of a designated register into the memory cell with address $A$ (overwriting the bits there).

## The Random Access Machine (RAM) model

### CPU

- Can do the following **atomic operations**:
  5. **(Randomness)**
     - RANDOM($x$, $y$): Given integers $x$ and $y$ (satisfying $x \leq y$), this operation returns an integer chosen uniformly at random in $[x, y]$, and places the random integer in a register.

## The Random Access Machine (RAM) model

An **execution** is a sequence of atomic operations.

Its **cost** (also called its **running time**, or simply, **time**) is the length of the sequence, namely, the number of atomic operations.

## The Random Access Machine (RAM) model

A **word** is a sequence of $w$ bits, where $w$ is called the **word length**.

- In other words, each memory cell and CPU register store a word.

Unless otherwise stated, you do not need to pay attention to the value of $w$ in this course.

## Algorithm

- An **input** refers to the initial state of the registers and the memory before an execution starts.
- An **algorithm** is a piece of description that, given an input, can be utilized to **unambiguously** produce a sequence of atomic operations, namely, the **execution** of the algorithm.
  - In other words, it should be always clear what the next atomic operation should be, given the outcome of all the previous atomic operations.
- The **cost** of an algorithm on an input is the length of its execution on that input (i.e., the number of atomic operations required).
- The **space** of an algorithm on an input is the **largest** memory address accessed by the algorithm's execution on that input.

## Deterministic Algorithms vs. Random Algorithms

An algorithm is **deterministic** if it never invokes the atomic operation RANDOM. Otherwise, the algorithm is **randomized**.

On the same input, the cost of a deterministic algorithm is a fixed integer—it remains the same every time you execute the algorithm.

The cost of a randomized algorithm, however, is a random variable. Even on the same input, the cost may change each time the algorithm is executed.

Example

1. **do**
2. $r = \text{RANDOM}(0, 1)$
3. **until** $r = 1$

How many times would Line 2 be executed? The answer is—"we don't know" (in fact, the line may be executed an infinite number of times)! Every time the above "algorithm" is executed, it may produce a new sequence of atomic operations.

## Expected Cost of a Randomized Algorithm

Let $X$ be a random variable that equals the cost of an algorithm on an input. The **expected cost** of the algorithm on the input is the expectation of $X$.

Wait a Moment: It is a well-known fact that **different distributions can share the same expected value, yet behave quite differently.** So,

- **Question:** Why is expected running time a good metric for analyzing randomized algorithms?

# Why Expected Running Time?

Why is expected running time a good metric for analyzing randomized algorithms?

1. Simplicity and Analytic Tractability:
   - The expected running time is usually easier to compute and analyze than other probabilistic metrics (like the median, variance, or tail bounds).
   - It provides a single, concrete number that summarizes the algorithm's typical performance.
2. Linearity of Expectation:
   - One of the most powerful tools in probability theory is the linearity of expectation. It allows us to analyze complex algorithms by breaking them into smaller parts and summing their expected costs—even if those parts are dependent.
   - This makes expected running time a very modular and flexible tool in algorithm design.
3. Concentration Bounds: If the running time of a randomized algorithm has very high variance, the expectation may be misleading. In most cases, this can be reconciled if we utilize concentration bounds like
   - the Markov inequality, Chebyshev inequality, or Chernoff bounds

Next, we will use Markov inequality as an example. (We also talked about Chernoff bounds during the first lecture. But that will not appear in quizzes/exams. )

# Markov's Inequality

Markov's Inequality provides an upper bound on the probability that a non-negative random variable is much larger than its expectation.

**Statement**

Let $X$ be a non-negative random variable and $a > 0$. Then:

$$\boldsymbol{Pr}[X \geq a] \leq \frac{\mathbb{E}[X]}{a}.$$

# Intuition Behind Markov's Inequality

- If the average value of $X$ is small, the chance that $X$ is very large must be small.

**Example**

Suppose $\mathbb{E}[X] = 5$. Then:

$$\boldsymbol{Pr}[X \geq 25] \leq \frac{5}{25} = 0.2$$

So there's at most a 20% chance that $X$ exceeds 25.

# Applying Markov to Running Time

Let $T$ be the random variable for the running time of a randomized algorithm.
Suppose:

- $T \geq 0$
- $\mathbb{E}[T] = \mu$

Then for any $c > 1$:

$$\boldsymbol{Pr}[T \geq c \cdot \mu] \leq \frac{1}{c}$$

Example

If $\mu = 10$, then:

$$\boldsymbol{Pr}[T \geq 100] \leq \frac{1}{10}$$

# Example: Las Vegas Algorithm

- A Las Vegas algorithm always gives the correct result, with a random running time.
- Suppose $\mathbb{E}[T] = \mu$
- Run the algorithm with a timeout of $c \cdot \mu$

$$\boldsymbol{Pr}[\text{timeout}] \leq \frac{1}{c}$$

- Run for $3\mu$ steps $\Rightarrow$ fail with probability $\leq \frac{1}{3}$
- Repeat 3 times $\Rightarrow$ succeed with good-enough probability $1 - \left(\frac{1}{3}\right)^3 \approx 0.963$.