Lecture 1: Introduction

The goal of this class is to teach you to **solve** computation problems, and to communicate that your solutions are **correct** and **efficient**.

Problem

- Binary relation from **problem inputs** to **correct outputs**
- Usually don't specify every correct output for all inputs (too many!)
- Provide a verifiable **predicate** (a property) that correct outputs must satisfy
- **Bounded** inputs
  - In this room, is there a pair of students with same birthday?
  - Not general, small input
- **Unbounded** inputs (generalization, arbitrarily large)
  - Given any set of n students, is there a pair of students with same birthday?
  - (Size of input is often called 'n', but not always!)

Algorithm

- Procedure mapping each input to a **single** output (deterministic)
- Algorithm **solves** a problem if returns a correct output for every problem input
- Birthday Matching
  - Maintain **record** of names and birthdays (initially empty)
  - Interview each student in some order
    - If birthday exists in record, return found pair!
    - Else add name and birthday to record
  - Return None if last student interviewed without success

Correctness

- Programs/algorithms have constant size
- For **bounded** inputs, can use case analysis
- For **unbounded** inputs, algorithm must be **recursive** or loop in some way
- Must use **induction** (why recursion is such a key concept in computer science)
- Birthday Matching
  - Induct on first k: number of students in record
  - **Base case**: $k = 0$, record has no match, and algorithm does not find one
  - If first k contains a match, already returns correctly by induction, thus so does $k + 1$
  - Else first k do not have match, so if first $k + 1$ has match, it contains $k + 1$
  - Algorithm then checks whether birthday of student $k + 1$ already exists in first k $\square$

Efficiency

- Produces a correct output in **polynomial time** with respect to input size n
- (Sometimes no efficient algorithm exists for a problem, L20)
- Asymptotic Notation (from prereq)
  - Upper bounds (O), lower bounds (Ω), tight bounds (Θ)
  - $\in, =$, is, order
  - Particles in universe estimated $< 10^{100}$

input	constant	logarithmic	linear	log-linear	quadratic	polynomial	exponential
n	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^c)$	$2^{\Theta(n^c)}$
1000	1	≈ 10	1000	$\approx$ 10,000	1,000,000		$2^{1000} \approx 10^{301}$

Model of Computation (Word-RAM)

- **Memory**: Addressable sequence of machine words
- **Machine word**: block of w bits (word size, a w-bit Word-RAM)
- **Processor** supports many **constant time** operations on words:
  - **integer** arithmetic: `(+, -, *, //, %)`
  - **logical** operators: `(&&, ||, !, ==, <, >, <=, =>)`
  - (**bitwise** arithmetic: `(&, |, <<, >>, ...)`)
  - Given word a, can **read** word at address a, **write** word to address a
- Memory address must be able to access every place in memory
- Assumption: $w \geq$ # bits to represent largest memory address $(\log_2 n)$
- 32-bit words $\rightarrow$ max ~ 4 GB memory
- 64-bit words $\rightarrow$ max $\sim 10^{10}$ GB of memory
- **Python** is a more complicated model of computation, implemented on a Word-RAM

Data Structure

- A **data structure** is a way to store non-constant data, that supports a set of operations
- Collection of operations is called an **interface**
  - Sequence: Extrinsic order to items (first, last, nth)
  - Set: Intrinsic order to items (queries based on item keys)
- Data structures may implement the same interface with different performance
- **Static Array** - fixed width slots, fixed length, static sequence interface
  - `StaticArray(n)`: allocate a new static array of size n in $\Theta(n)$ time
  - `StaticArray.at(i)`: return word stored at array index i in $\Theta(1)$ time
  - `StaticArray.set(i, x)`: write word x to array index i in $\Theta(1)$ time
- Stored word can hold the address of a larger object
- Like Python `tuple` plus `set(i, x)`
- Python `list` is a **dynamic array** (see L02)

```python
def birthday_match(students):
    '''
    Find a pair of students with the same birthday
    Input:  tuple of student (name, bday) tuples
    Output: tuple of student names or None
    '''
    n = len(students)                           # O(1)
    record = StaticArray(n)                     # O(n)
    for k in range(n):                          # n
        (name1, bday1) = students[k]            # O(1)
        for i in range(k):                      # k      Check if in record
            (name2, bday2) = record.at(i)       # O(1)
            if bday1 == bday2:                  # O(1)
                return (name1, name2)           # O(1)
        record.set(k, (name1, bday1))           # O(1)
    return None                                 # O(1)
```

Analysis

- Two loops: outer $k \in \{0, \ldots, n-1\}$, inner is $i \in \{0, \ldots, k\}$
- Running time is $O(n) + \sum_{k=0}^{n-1}(O(1) + k \cdot O(1)) = O(n^2)$
- Quadratic in n is **polynomial**. Efficient?
- Can do better using different data structure!

How to Solve an Algorithms Problem

1. Reduce to a problem you already know (use data structure or algorithm)

Search Problem (Data Structures)	Sort Algorithms	Shortest Path Algorithms
Static Array (L01)	Insertion Sort (L03)	Breadth First Search (L09)
Linked List (L02)	Selection Sort (L03)	DAG Relaxation (L11)
Dynamic Array (L02)	Merge Sort (L03)	Depth First Search (L10)
Sorted Array (L03)	Counting Sort (L05)	Topological Sort (L10)
Direct-Access Array (L04)	Radix Sort (L05)	Bellman-Ford (L12)
Hash Table (L04)	AVL Sort (L07)	Dijkstra (L13)
Balanced Binary Tree (L06-L07)	Heap Sort (L08)	Johnson (L14)
Binary Heap (L08)		Floyd-Warshall (L18)

2. Design your own (recursive) algorithm

   - Brute Force
   - Decrease and Conquer
   - Divide and Conquer
   - **Dynamic Programming** (L15-L19)
   - Greedy / Incremental