# 01. Algorithm Fundamentals

## CPSC 535 ~ Spring 2019

Kevin A. Wortman

California State University, Fullerton

January 27, 2019

# Problems

*Computational problem:* definition of output and desired output

Each is a mathematical object that could be stored in a computer data structure.

**Sorting problem**
**input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
**output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

# Algorithms

*Instance (of a problem):* Concrete input datum

**Example**

$\langle 71, 14, 31, 2, 82 \rangle$

*Algorithm:* well-defined computational procedure that unerringly transforms input to output

# Motivation

Why do we care about algorithms, or algorithmic efficiency?

- Algorithms are automating major parts of the economy: operations research, high frequency trading, machine learning, etc.
- Efficiency can mean the difference between computations being viable, sustainable, for human use versus impractical.
- The principle of not wasting product.
- Intriguing mathematical questions are worth studying in their own right.

# Data Structures

*Data Structure:* method for storing, organizing data

- Data members e.g. head pointer, tail pointer
- *Invariant(s)* defining how the structure must be organized to remain valid, e.g. head points to first node, tail points to last node
- Defined *operations*, each operation is an algorithm that operates on the structure.

# Pseudocode

*Pseudocode:* code-like notation for conveying algorithms

- Goal is clear communication to a human audience
- Not compiled, so no need to be syntactically perfect
- Not software engineering; no need for error checking, modularity, encapsulation, etc.

*Algorithm implementer* is a specific role and skill set, bridging the gap between scholarly pseudocode and industrial coding practices.

# Insertion Sort

```
function INSERTION-SORT(A)
    for j = 2 to A.length do
        key = A[j]
        // Insert A[j] into the sorted sequence A[1 ... j-1].
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i-1
        end while
        A[i+1] = key
    end for
end function
```

# Pseudocode Observations

- Algorithm is a function/procedure, input is argument(s)
- No global variables
- Code-like but not compile-able code
- Arrays start at 1
- No error checking or modularity
- Translatable into practically any programming langauge

# Analysis

*Analysis:* establish how efficient an algorithm is

- Usually a mathematical proof (alternatively empirical evidence)
- Usually analyze for time spent (or disk I/O, space, energy, randomness, etc.)
- Usually summarize resource use by *order of growth* in asymptotic notation; $O(n), \Theta(n^2)$, etc.

# RAM model

*Computational model:* defines how a computer executes an algorithm, specifically enough to measure time (or other resources)
*Random Access Machine (RAM):*

- "default" computational model, approximates a generic real-world CPU and memory
- CPU has instructions for integer arithmetic, floating point arithmetic, control (jump, call, return, if), logic (or, and, not), data copying.
- one step $\equiv O(1)$ instructions; each pseudocode statement counts as 1 step (except function calls)
- CPU has some $O(1)$ word size, e.g. 32 or 64 bit; one instruction is limited to writing that many bits
- cannot "cheat" by packing $\Theta(n)$ information in one word

# Worst-Case Analysis

- In a time analysis, we need to prove how much time insertion sort takes when run
- depends on the type of input, e.g. pre-sorted, completely jumbled, in between
- convention: analyze the *worst case* for the algorithm at hand
- generous to skeptics, conservative for software engineers
- as an exception, sometimes analyze *average case* of deliberately randomized algorithms

*Claim:* The worst-case time complexity of insertion sort is $\Theta(n^2)$.

# Divide-and-conquer

1. **divide** input into several smaller instances of the same problem (often, divide input in half)
2. **"conquer"** by recursively solving all the sub-problems; may involve a simple base case
3. **combine** the many solutions into one coherent solution for the original problem

# Merge sort

*Merge sort:* classical sorting algorithm using divide-and-conquer

**divide:** chop list of $n$ unsorted elements into two lists of $n/2$ elements each
**conquer:** merge-sort each unsorted list; if $n \leq 1$, nothing to do
**combine:** *merge* two sorted lists of $n/2$ elements, into one sorted list of $n$ elements

# Merge pseudocode

**Ensure:** $A[p \dots r]$ is sorted

**function** MERGE-SORT(A, p, r)
    **if** $p < r$ **then**
        $q = \lfloor (p + r)/2 \rfloor$
        MERGE-SORT$(A, p, q)$
        MERGE-SORT$(A, q + 1, r)$
        MERGE$(A, p, q, r)$
    **end if**

**Require:** $p \leq q < r$, $A[p \dots q]$ is sorted, $A[q + 1 \dots r]$ is sorted
**Ensure:** $A[p \dots r]$ is sorted

**function** MERGE(A, p, q, r)
    $n_1 = (q - p + 1)$, $n_2 = (r - q)$
    let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be new arrays
    $L[1 \dots n_1] = A[p \dots q]$
    $R[1 \dots n_2] = A[p + 1 \dots q]$
    $L[n_1 + 1] = R[n_2 + 2] = \infty$
    $i = j = 1$
    **for** $k = p$ to $r$ **do**
        **if** $L[i] \leq R[j]$ **then**
            $A[k] = L[i]$
            $i = i + 1$
        **else**
            $A[k] = R[j]$
            $j = j + 1$
        **end if**
    **end for**

# Merge sort analysis

The worst-case time complexity of merge sort is given by the *recurrence relation*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

*Claim:* $T \in \Theta(n \log n)$

*Claim:* Merge sort uses $\Theta(n)$ extra space for $L$ and $R$. (Observe that at most one merge is happening at any moment, and the largest merge uses $n + 2$ extra array elements.)

# Insertion sort versus merge sort

Insertion sort: $\Theta(n^2)$ time, $\Theta(1)$ space

Merge sort: $\Theta(n \log n)$ time, $\Theta(n)$ space

Merge sort is subjectively more convoluted

Space vs. time tradeoff (typical)

Efficiency vs. convolution tradeoff (typical)

Refactoring convolution into algorithm design is usually a win