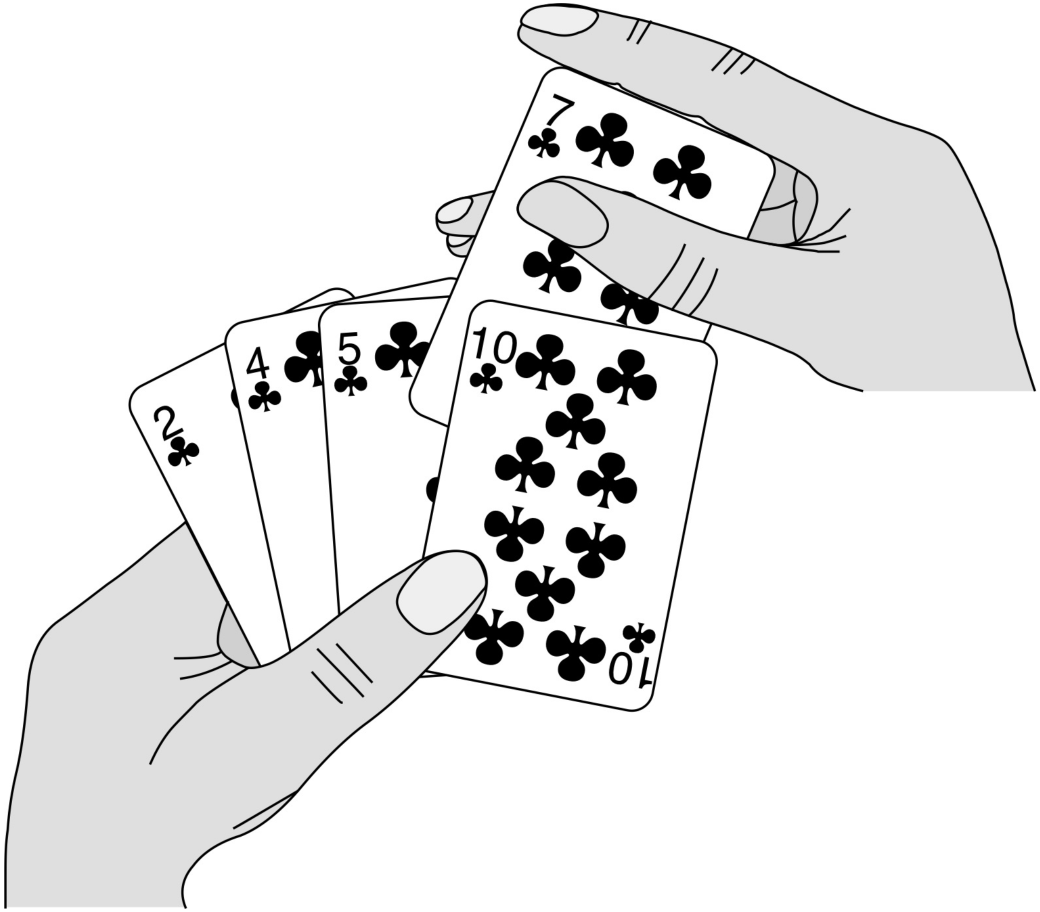
Insertion Sort & Algorithm Analysis

CLRS 2.1 & 2.2



Sorting Problem

- **Input**: A sequence of *n* numbers $a_1, a_2, \ldots, a_n$
- **Output**: A permutation (reordering) $a'_1, a'_2, \ldots, a'_n$ such that

$$a'_1 \le a'_2 \le \cdots \le a'_n$$

4, 2, 6, 1, 3, 8, 5, 7, 5, 2 → 1, 2, 2, 3, 4, 5, 5, 6, 7, 8

Motivation:

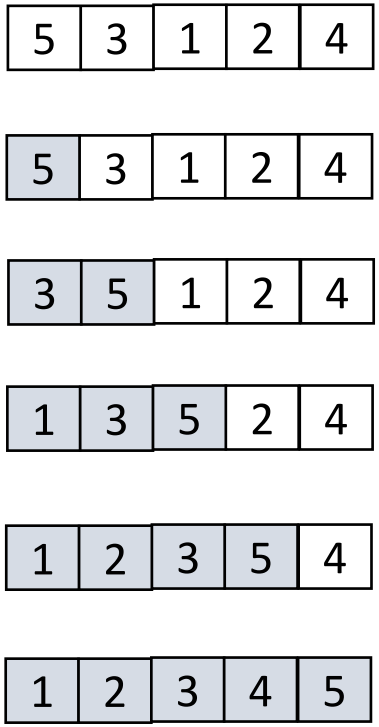
- Fundamental problem in CS
- Often used as a pre-processing step to solve other problems more efficiently
- Many approaches to solve

Q: Suppose you are given a set of 15 student papers, and you need to arrange them in alphabetical order. How do you sort them?

Insertion Sort

already sorted | yet to be processed

Idea: iteratively build up a sorted list on the left, **inserting** the next item into its appropriate position in the sorted list



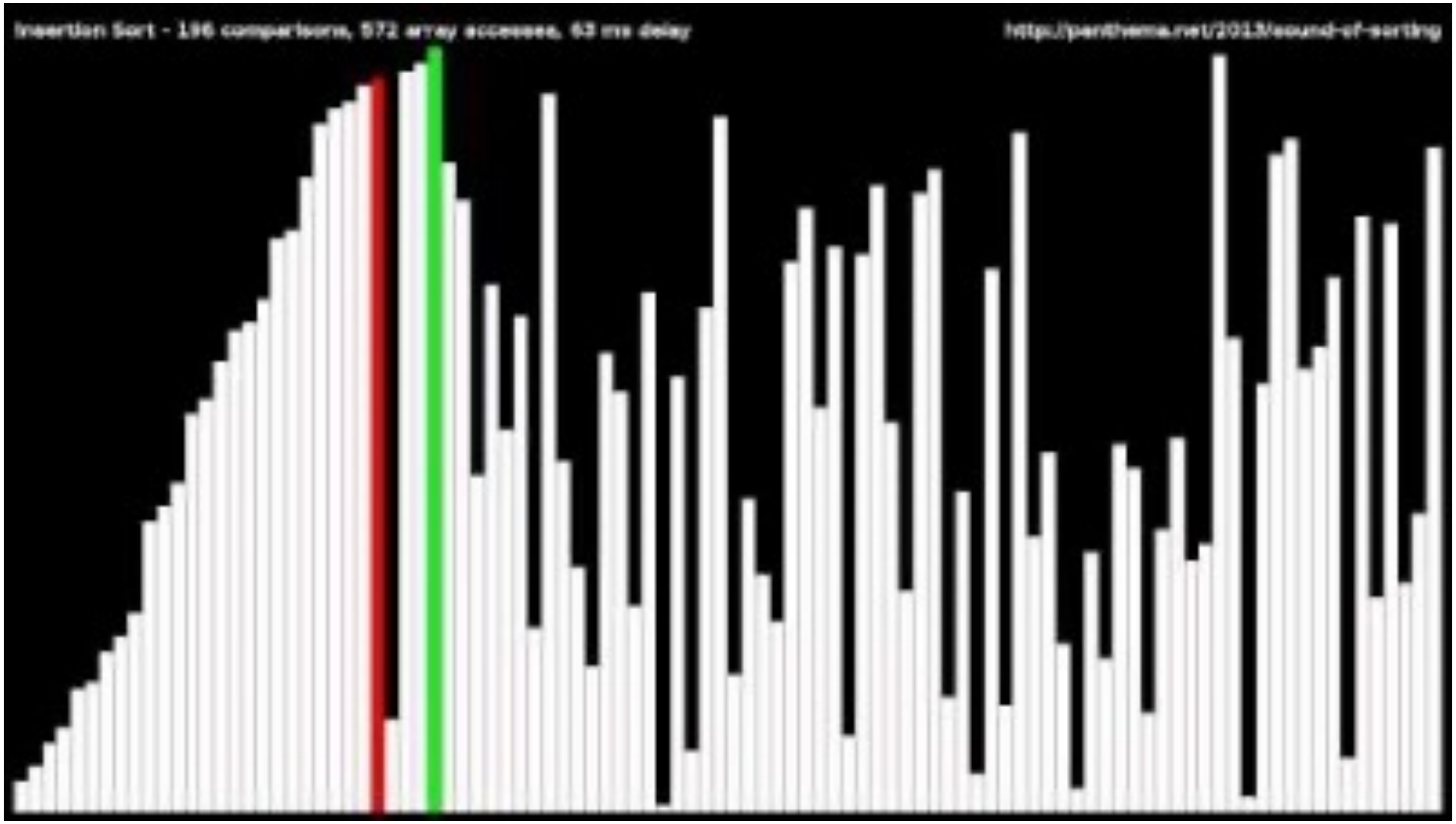
```
INSERTION-SORT(A, n)
  for $j = 2$ to $n$
      $key = A[j]$
      // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.
      $i = j - 1$
      while $i > 0$ and $A[i] > key$
          $A[i+1] = A[i]$
          $i = i - 1$
      $A[i+1] = key$
```

This algorithm sorts **in-place**, meaning it works directly on the provided array and only a constant amount of additional memory is used

Insertion Sort

already sorted | yet to be processed

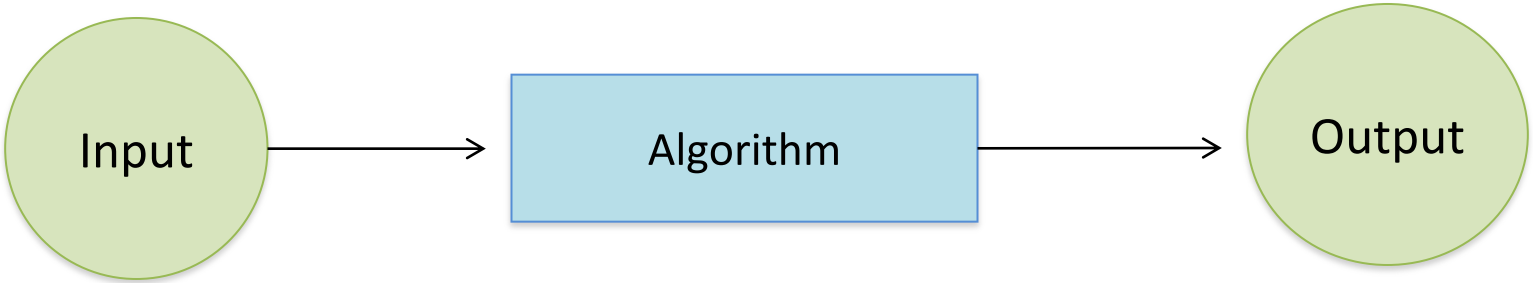
Idea: iteratively build up a sorted list on the left, **inserting** the next item into its appropriate position in the sorted list



https://www.youtube.com/watch?v=8oJS1BMKE64

Algorithm Analysis

- An algorithm is a step-by-step procedure for performing some task (ex: sorting a set of integers) in a finite amount of time.



- We are concerned with the following properties:
    - Correctness
    - Efficiency (how fast it is, how many resources it needs)

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

Loop invariant: *At the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A[1..j-1], but in sorted order*

We must show three things about a loop invariant:

- **Initialization**: It is true prior to the first iteration of the loop
- **Maintenance**: If it’s true before an iteration of the loop, it remains true before the next iteration
- **Termination**: When the loop terminates, the invariant gives us a useful property that helps show the algorithm is correct

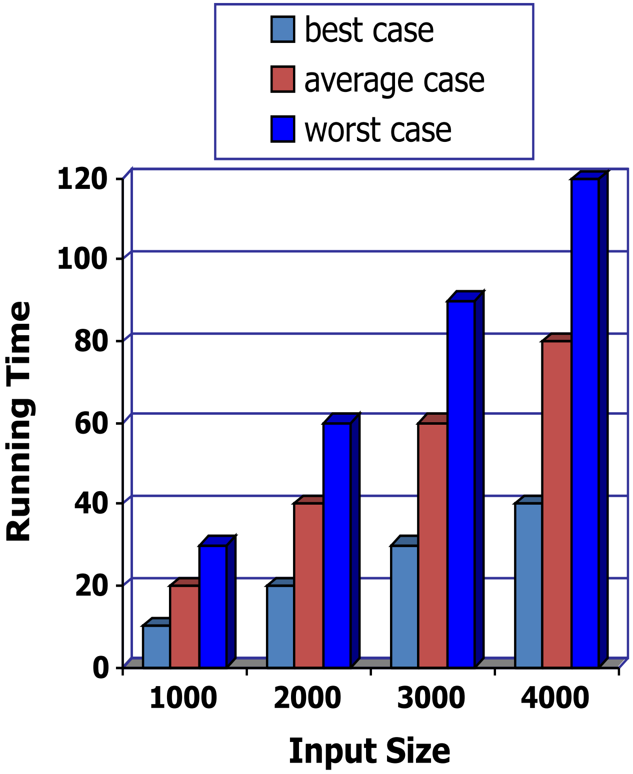
Running time

- The running time of an algorithm typically grows with the input size.

- Average case time is often difficult to determine.

- We often focus on the worst case running time.
  - Easier to analyze
  - Good standard of success

Q: How to determine run time?

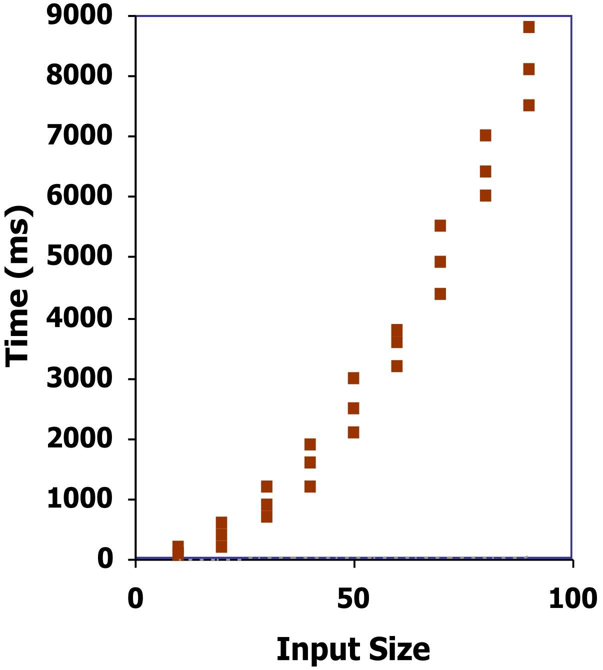


(1) Experimental studies

- Write a program implementing the algorithm

- Run the program with inputs of varying size and composition
  - Use a method like `std::clock()` to get an accurate measure of the actual running time

- Plot the results



<u>Limitations of experimental studies</u>

- Need to implement the algorithm
  - may be difficult
- Experiments done on a limited set of test inputs
  - may not be indicative of running times on other inputs not included in the experiment
- Difficult to compare
  - same hardware and software environments must be used

(2) Theoretical Analysis

- Takes into account all possible inputs

- Characterizes running time by *f(n)*, a function of the input size *n*
    - allows us to evaluate the speed of an algorithm independent of hardware/software environment

- Uses pseudocode, the preferred notation for describing algorithms
    - mix of natural language and high-level programming constructs that describe the main ideas behind an algorithm implementation
    - no implementation necessary
    - preferred notation for describing algorithms
    - language-agnostic, hiding implementation details

```
Algorithm arrayMax($A$, $n$)
    Input array $A$ of $n$ integers
    Output maximum element of $A$

    $currentMax = A[1]$
    for $i = 1$ to $n$ do
        if $A[i] > currentMax$ then
            $currentMax = A[i]$
    return $currentMax$
```

Pseudo-code details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

- Method call

  var.method (***arg*** [, ***arg***…])

- Return value

  return ***expression***

- Expressions

  = Assignment

  == Equality testing

  n^2 Superscripts and other mathematical formatting allowed

Random Access Machine (RAM) Model

- Views a computer as a generic one-processor
  - Simplistic
  - Instructions executed one after the other; no concurrent operations
  - No concern with memory hierarchy

- Instructions are those commonly found in real computers:
  - Arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling)
  - Data movement (load, store, copy)
  - Control (conditional and unconditional branch, subroutine call and return)

- **Each instruction takes a constant amount of time**

RAM-model analyses are usually excellent predictors of performance on actual machines

Analysis of insertion sort

```
INSERTION-SORT($A, n$)                                          cost    times
  for $j = 2$ to $n$                                            $c_1$   $n$
      $key = A[j]$                                              $c_2$   $n-1$
      // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.    $0$     $n-1$
      $i = j - 1$                                               $c_4$   $n-1$
      while $i > 0$ and $A[i] > key$                            $c_5$   $\sum_{j=2}^{n} t_j$
          $A[i+1] = A[i]$                                       $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
          $i = i - 1$                                           $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
      $A[i+1] = key$                                            $c_8$   $n-1$
```

Note: t_j is the number of times the while loop test is executed for that value of *j*

Best-case running time?

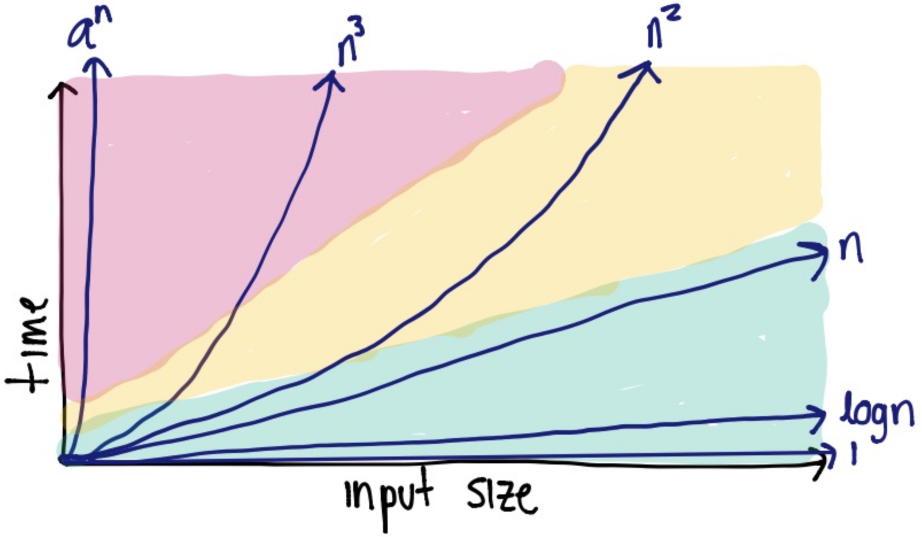
Worst-case running time?

- We can express the best-case running time as $an + b$ for some constants a, b. Thus, this is a **linear function** of *n*.
- We can express the worst-case running time as $an^2 + bn + c$ for some constants a, b, c. Thus, this is a **quadratic function** of *n*.

Order of growth

It's the **rate of growth**, or the order of growth, of the running time which is most interesting. As *n* grows large, how does the algorithm perform?

Constant	≈ 1
Logarithmic	$\approx \log n$
Linear	$\approx n$
Quadratic	$\approx n^2$
Cubic	$\approx n^3$
Polynomial	$\approx n^k$ (for $k \geq 1$)
Exponential	$\approx a^n$ ($a > 1$)



Growth rate is not affected by

- constant coefficients, nor
- lower-order terms

We can say that insertion-sort has a worst-case running time of $\theta(n^2)$ *"theta of n-squared"*

Ex: $10^2n + 10^5$ is a **linear** function

Ex: $10^5n^2 + 10^8n$ is a **quadratic** function