

Asymptotic Analysis of Algorithms



EECS2030 B: Advanced
Object Oriented Programming
Fall 2019

CHEN-WEI WANG

Measuring “Goodness” of an Algorithm



1. **Correctness**:
 - Does the algorithm produce the expected output?
 - Use JUnit to ensure this.
2. Efficiency:
 - **Time Complexity**: processor time required to complete
 - **Space Complexity**: memory space required to store data

Correctness is always the priority.

How about efficiency? Is time or space more of a concern?

3 of 42

Algorithm and Data Structure



- A **data structure** is:
 - A systematic way to store and organize data in order to facilitate **access** and **modifications**
 - Never suitable for all purposes: it is important to know its **strengths** and **limitations**
- A **well-specified computational problem** precisely describes the desired **input/output relationship**.
 - **Input**: A sequence of n numbers (a_1, a_2, \dots, a_n)
 - **Output**: A permutation (reordering) $(a'_1, a'_2, \dots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$
 - An **instance** of the problem: $\{3, 1, 2, 5, 4\}$
- An **algorithm** is:
 - A solution to a well-specified **computational problem**
 - A **sequence of computational steps** that takes value(s) as **input** and produces value(s) as **output**
- Steps in an **algorithm** manipulate well-chosen **data structure(s)**.

2 of 42

Measuring Efficiency of an Algorithm



- **Time** is more of a concern than is **storage**.
- Solutions that are meant to be run on a computer should run **as fast as possible**.
- Particularly, we are interested in how **running time** depends on two **input factors**:
 1. size
e.g., sorting an array of 10 elements vs. 1m elements
 2. structure
e.g., sorting an already-sorted array vs. a hardly-sorted array
- **How do you determine the running time of an algorithm?**
 1. Measure time via **experiments**
 2. Characterize time as a **mathematical function** of the input size

4 of 42

Measure Running Time via Experiments



- Once the algorithm is implemented in Java:
 - Execute the program on *test inputs* of various *sizes* and *structures*.
 - For each test, record the *elapsed time* of the execution.

```
long startTime = System.currentTimeMillis();
/* run the algorithm */
long endTime = System.currenctTimeMillis();
long elapsed = endTime - startTime;
```

- Visualize* the result of each test.
- To make **sound statistical claims** about the algorithm's *running time*, the set of input tests must be "reasonably" *complete*.

5 of 42

Example Experiment: Detailed Statistics



n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,847,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421 (\approx 3 days)	135

- As *input size* is doubled, **rates of increase** for both algorithms are *linear*:
 - Running time* of repeat1 increases by \approx 5 times.
 - Running time* of repeat2 increases by \approx 2 times.

7 of 42

Example Experiment



- Computational Problem:**
 - Input:** A character c and an integer n
 - Output:** A string consisting of n repetitions of character c
e.g., Given input '*' and 15, output *****.
- Algorithm 1** using *String* Concatenations:

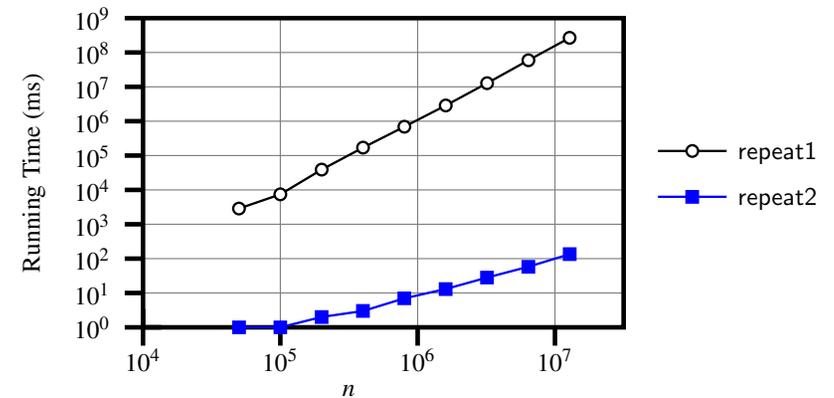
```
public static String repeat1(char c, int n) {
    String answer = "";
    for (int i = 0; i < n; i++) { answer += c; }
    return answer; }
```

- Algorithm 2** using *StringBuilder* append's:

```
public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) { sb.append(c); }
    return sb.toString(); }
```

6 of 42

Example Experiment: Visualization



8 of 42

Experimental Analysis: Challenges



1. An algorithm must be *fully implemented* (i.e., translated into valid Java syntax) in order to study its runtime behaviour *experimentally*.
 - What if our purpose is to *choose among alternative* data structures or algorithms to implement?
 - Can there be a *higher-level analysis* to determine that one algorithm or data structure is *superior* than others?
2. Comparison of multiple algorithms is only *meaningful* when experiments are conducted under the same environment of:
 - *Hardware*: CPU, running processes
 - *Software*: OS, JVM version
3. Experiments can be done only on a *limited set of test inputs*.
 - What if “*important*” inputs were not included in the experiments?

9 of 42

Counting Primitive Operations



A *primitive operation* corresponds to a low-level instruction with a *constant execution time*.

- Assignment [e.g., `x = 5;`]
- Indexing into an array [e.g., `a[i]`]
- Arithmetic, relational, logical op. [e.g., `a + b`, `z > w`, `b1 && b2`]
- Accessing an attribute of an object [e.g., `acc.balance`]
- Returning from a method [e.g., `return result;`]

Q: Why is a *method call* in general *not* a primitive operation?

A: It may be a call to:

- a “*cheap*” method (e.g., printing `Hello World`), or
- an “*expensive*” method (e.g., sorting an array of integers)

11 of 42

Moving Beyond Experimental Analysis



- A better approach to analyzing the *efficiency* (e.g., *running times*) of algorithms should be one that:
 - Allows us to calculate the *relative efficiency* (rather than absolute elapsed time) of algorithms in a way that is *independent of* the hardware and software environment.
 - Can be applied using a *high-level description* of the algorithm (without fully implementing it).
 - Considers *all* possible inputs.
- We will learn a better approach that contains 3 ingredients:
 1. Counting *primitive operations*
 2. Approximating running time as *a function of input size*
 3. Focusing on the *worst-case* input (requiring the most running time)

10 of 42

Example: Counting Primitive Operations



```
1 findMax (int[] a, int n) {
2   currentMax = a[0];
3   for (int i = 1; i < n; ) {
4     if (a[i] > currentMax) {
5       currentMax = a[i]; }
6     i ++ }
7   return currentMax; }
```

of times `i < n` in **Line 3** is executed? [n]

of times the loop body (**Line 4 to Line 6**) is executed? [$n - 1$]

- **Line 2:** 2 [1 indexing + 1 assignment]
- **Line 3:** $n + 1$ [1 assignment + n comparisons]
- **Line 4:** $(n - 1) \cdot 2$ [1 indexing + 1 comparison]
- **Line 5:** $(n - 1) \cdot 2$ [1 indexing + 1 assignment]
- **Line 6:** $(n - 1) \cdot 2$ [1 addition + 1 assignment]
- **Line 7:** 1 [1 return]
- **Total # of Primitive Operations:** $7n - 2$

12 of 42

From Absolute RT to Relative RT

- Each *primitive operation* (PO) takes approximately the same, constant amount of time to execute. [say t]
- The *number of primitive operations* required by an algorithm should be **proportional** to its *actual running time* on a specific environment.
e.g., `findMax (int[] a, int n)` has $7n - 2$ POs

$$RT = (7n - 2) \cdot t$$

- Say two algorithms with RT $(7n - 2) \cdot t$ and RT $(10n + 3) \cdot t$.
 \Rightarrow It suffices to compare their **relative** running time:
 $7n - 2$ vs. $10n + 3$.
- To determine the **time efficiency** of an algorithm, we only focus on their **number of POs**.

13 of 42

Approximating Running Time as a Function of Input Size

Given the *high-level description* of an algorithm, we associate it with a function f , such that $f(n)$ returns the *number of primitive operations* that are performed on an *input of size n* .

- $f(n) = 5$ [constant]
- $f(n) = \log_2 n$ [logarithmic]
- $f(n) = 4 \cdot n$ [linear]
- $f(n) = n^2$ [quadratic]
- $f(n) = n^3$ [cubic]
- $f(n) = 2^n$ [exponential]

15 of 42

Example: Approx. # of Primitive Operations

- Given # of primitive operations counted precisely as $7n - 2$, we view it as

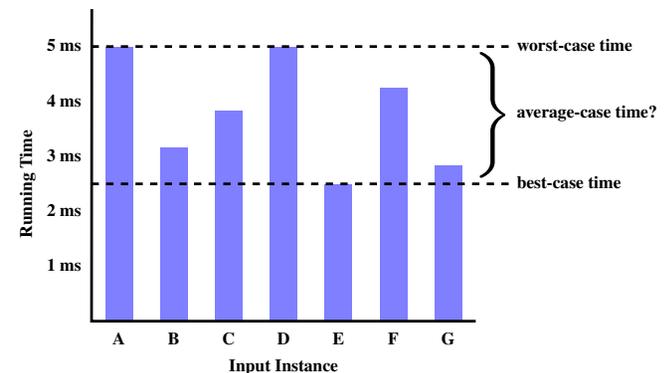
$$7 \cdot n^1 - 2 \cdot n^0$$
- We say
 - n is the **highest power**
 - 7 and 2 are the **multiplicative constants**
 - 2 is the **lower term**
- When approximating a function (considering that input size may be very large):
 - Only** the **highest power** matters.
 - multiplicative constants** and **lower terms** can be dropped. $\Rightarrow 7n - 2$ is approximately n

Exercise: Consider $7n + 2n \cdot \log n + 3n^2$:

- highest power?** [n^2]
- multiplicative constants?** [7, 2, 3]
- lower terms?** [$7n + 2n \cdot \log n$]

14 of 42

Focusing on the Worst-Case Input



- Average-case** analysis calculates the *expected running times* based on the probability distribution of input values.
- worst-case** analysis or **best-case** analysis?

16 of 42

What is Asymptotic Analysis?



Asymptotic analysis

- Is a method of describing *behaviour in the limit*:
 - How the *running time* of the algorithm under analysis changes as the *input size* changes without bound
 - e.g., contrast $RT_1(n) = n$ with $RT_2(n) = n^2$
- Allows us to compare the *relative* performance of alternative algorithms:
 - For large enough inputs, the *multiplicative constants* and *lower-order* terms of an exact running time can be disregarded.
 - e.g., $RT_1(n) = 3n^2 + 7n + 18$ and $RT_2(n) = 100n^2 + 3n - 100$ are considered **equally efficient**, *asymptotically*.
 - e.g., $RT_1(n) = n^3 + 7n + 18$ is considered **less efficient** than $RT_2(n) = 100n^2 + 100n + 2000$, *asymptotically*.

Three Notions of Asymptotic Bounds



We may consider three kinds of *asymptotic bounds* for the *running time* of an algorithm:

- Asymptotic **upper** bound $[O]$
- Asymptotic lower bound $[\Omega]$
- Asymptotic tight bound $[\Theta]$

Asymptotic Upper Bound: Definition

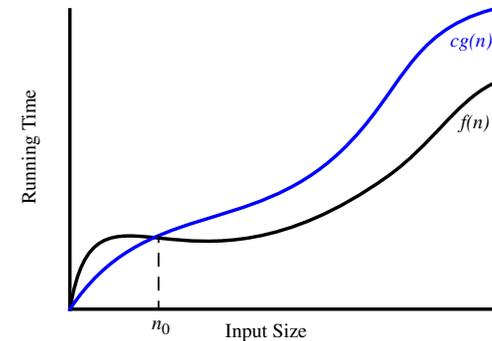


- Let $f(n)$ and $g(n)$ be functions mapping positive integers (input size) to positive real numbers (running time).
 - $f(n)$ characterizes the running time of some algorithm.
 - $O(g(n))$ denotes *a collection of* functions.
- $O(g(n))$ consists of *all* functions that can be upper bounded by $g(n)$, starting at some point, using some constant factor.
- $f(n) \in O(g(n))$ if there are:
 - A real *constant* $c > 0$
 - An integer *constant* $n_0 \geq 1$
 such that:

$$f(n) \leq c \cdot g(n) \text{ for } n \geq n_0$$

- For each member function $f(n)$ in $O(g(n))$, we say that:
 - $f(n) \in O(g(n))$ [f(n) is a member of "big-Oh of g(n)"]
 - $f(n)$ is $O(g(n))$ [f(n) is "big-Oh of g(n)"]
 - $f(n)$ is order of $g(n)$

Asymptotic Upper Bound: Visualization



From n_0 , $f(n)$ is upper bounded by $c \cdot g(n)$, so $f(n)$ is $O(g(n))$.

Asymptotic Upper Bound: Example (1)



Prove: The function $8n + 5$ is $O(n)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$8n + 5 \leq c \cdot n$$

Can we choose $c = 9$? What should the corresponding n_0 be?

n	8n + 5	9n
1	13	9
2	21	18
3	29	27
4	37	36
5	45	45
6	53	54

Therefore, we prove it by choosing $c = 9$ and $n_0 = 5$.

We may also prove it by choosing $c = 13$ and $n_0 = 1$. Why?

21 of 42

Asymptotic Upper Bound: Proposition (1)



If $f(n)$ is a polynomial of degree d , i.e.,

$$f(n) = a_0 \cdot n^0 + a_1 \cdot n^1 + \dots + a_d \cdot n^d$$

and a_0, a_1, \dots, a_d are integers, then $f(n)$ is $O(n^d)$.

- We prove by choosing

$$\begin{aligned} c &= |a_0| + |a_1| + \dots + |a_d| \\ n_0 &= 1 \end{aligned}$$

- We know that for $n \geq 1$: $n^0 \leq n^1 \leq n^2 \leq \dots \leq n^d$
- Upper-bound effect: $n_0 = 1$? $[f(1) \leq (|a_0| + |a_1| + \dots + |a_d|) \cdot 1^d]$

$$a_0 \cdot 1^0 + a_1 \cdot 1^1 + \dots + a_d \cdot 1^d \leq |a_0| \cdot 1^d + |a_1| \cdot 1^d + \dots + |a_d| \cdot 1^d$$

- Upper-bound effect holds? $[f(n) \leq (|a_0| + |a_1| + \dots + |a_d|) \cdot n^d]$

$$a_0 \cdot n^0 + a_1 \cdot n^1 + \dots + a_d \cdot n^d \leq |a_0| \cdot n^d + |a_1| \cdot n^d + \dots + |a_d| \cdot n^d$$

23 of 42

Asymptotic Upper Bound: Example (2)



Prove: The function $f(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq c \cdot n^4$$

$$f(1) = 5 + 3 + 2 + 4 + 1 = 15$$

Choose $c = 15$ and $n_0 = 1$!

22 of 42

Asymptotic Upper Bound: Proposition (2)



$$O(n^0) \subset O(n^1) \subset O(n^2) \subset \dots$$

If a function $f(n)$ is *upper bounded* by another function $g(n)$ of degree d , $d \geq 0$, then $f(n)$ is also upper bounded by all other functions of a *strictly higher degree* (i.e., $d + 1$, $d + 2$, etc.).

e.g., Family of $O(n)$ contains:

$$\begin{aligned} &n^0, 2n^0, 3n^0, \dots \\ &n, 2n, 3n, \dots \end{aligned}$$

[functions with degree 0]
[functions with degree 1]

e.g., Family of $O(n^2)$ contains:

$$\begin{aligned} &n^0, 2n^0, 3n^0, \dots \\ &n, 2n, 3n, \dots \\ &n^2, 2n^2, 3n^2, \dots \end{aligned}$$

[functions with degree 0]
[functions with degree 1]
[functions with degree 2]

24 of 42

Asymptotic Upper Bound: More Examples



- $5n^2 + 3n \cdot \log n + 2n + 5$ is $O(n^2)$ $[c = 15, n_0 = 1]$
- $20n^3 + 10n \cdot \log n + 5$ is $O(n^3)$ $[c = 35, n_0 = 1]$
- $3 \cdot \log n + 2$ is $O(\log n)$ $[c = 5, n_0 = 2]$
 - Why can't n_0 be 1?
 - Choosing $n_0 = 1$ means $\Rightarrow f(\boxed{1})$ **is** upper-bounded by $c \cdot \log \boxed{1}$:
 - We have $f(\boxed{1}) = 3 \cdot \log 1 + 2$, which is 2.
 - We have $c \cdot \log \boxed{1}$, which is 0.
 - $\Rightarrow f(\boxed{1})$ **is not** upper-bounded by $c \cdot \log \boxed{1}$ $[\text{Contradiction!}]$
- 2^{n+2} is $O(2^n)$ $[c = 4, n_0 = 1]$
- $2n + 100 \cdot \log n$ is $O(n)$ $[c = 102, n_0 = 1]$

25 of 42

Classes of Functions



upper bound	class	cost
$O(1)$	constant	<i>cheapest</i>
$O(\log(n))$	logarithmic	
$O(n)$	linear	
$O(n \cdot \log(n))$	"n-log-n"	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(n^k), k \geq 1$	polynomial	
$O(a^n), a > 1$	exponential	<i>most expensive</i>

27 of 42

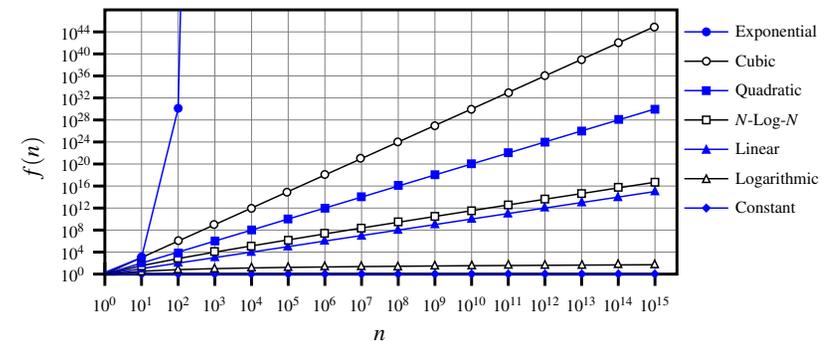
Using Asymptotic Upper Bound Accurately



- Use the big-Oh notation to characterize a function (of an algorithm's running time) **as closely as possible**.
For example, say $f(n) = 4n^3 + 3n^2 + 5$:
 - Recall: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \dots$
 - It is the **most accurate** to say that $f(n)$ is $O(n^3)$.
 - It is **true**, but not very useful, to say that $f(n)$ is $O(n^4)$ and that $f(n)$ is $O(n^5)$.
 - It is **false** to say that $f(n)$ is $O(n^2)$, $O(n)$, or $O(1)$.
- Do not include **constant factors** and **lower-order terms** in the big-Oh notation.
For example, say $f(n) = 2n^2$ is $O(n^2)$, do not say $f(n)$ is $O(4n^2 + 6n + 9)$.

26 of 42

Rates of Growth: Comparison



28 of 42

Upper Bound of Algorithm: Example (1)



```
1  maxOf (int x, int y) {
2      int max = x;
3      if (y > x) {
4          max = y;
5      }
6      return max;
7  }
```

- # of primitive operations: 4
2 assignments + 1 comparison + 1 return = 4
- Therefore, the running time is $O(1)$.
- That is, this is a *constant-time* algorithm.

29 of 42

Upper Bound of Algorithm: Example (3)



```
1  containsDuplicate (int[] a, int n) {
2      for (int i = 0; i < n; ) {
3          for (int j = 0; j < n; ) {
4              if (i != j && a[i] == a[j]) {
5                  return true; }
6              j ++; }
7          i ++; }
8      return false; }
```

- Worst case is when we reach Line 8.
- # of primitive operations $\approx c_1 + n \cdot n \cdot c_2$, where c_1 and c_2 are some constants.
- Therefore, the running time is $O(n^2)$.
- That is, this is a *quadratic* algorithm.

31 of 42

Upper Bound of Algorithm: Example (2)



```
1  findMax (int[] a, int n) {
2      currentMax = a[0];
3      for (int i = 1; i < n; ) {
4          if (a[i] > currentMax) {
5              currentMax = a[i]; }
6          i ++ }
7      return currentMax; }
```

- From last lecture, we calculated that the # of primitive operations is $7n - 2$.
- Therefore, the running time is $O(n)$.
- That is, this is a *linear-time* algorithm.

30 of 42

Upper Bound of Algorithm: Example (4)



```
1  sumMaxAndCrossProducts (int[] a, int n) {
2      int max = a[0];
3      for (int i = 1; i < n; i++) {
4          if (a[i] > max) { max = a[i]; }
5      }
6      int sum = max;
7      for (int j = 0; j < n; j++) {
8          for (int k = 0; k < n; k++) {
9              sum += a[j] * a[k]; } }
10     return sum; }
```

- # of primitive operations $\approx (c_1 \cdot n + c_2) + (c_3 \cdot n \cdot n + c_4)$, where $c_1, c_2, c_3,$ and c_4 are some constants.
- Therefore, the running time is $O(n + n^2) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

32 of 42

Upper Bound of Algorithm: Example (5)



```

1 triangularSum (int[] a, int n) {
2   int sum = 0;
3   for (int i = 0; i < n; i++) {
4     for (int j = i; j < n; j++) {
5       sum += a[j]; } }
6   return sum; }

```

- # of primitive operations $\approx n + (n - 1) + \dots + 2 + 1 = \frac{n(n+1)}{2}$
- Therefore, the running time is $O(\frac{n^2+n}{2}) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

33 of 42

Basic Data Structure: Arrays



- An array is a sequence of indexed elements.
- *Size* of an array is **fixed** at the time of its construction.
- Supported *operations* on an array:
 - *Accessing*: e.g., `int max = a[0];`
Time Complexity: $O(1)$ [constant operation]
 - *Updating*: e.g., `a[i] = a[i + 1];`
Time Complexity: $O(1)$ [constant operation]
 - *Inserting/Removing*:

```

String[] insertAt(String[] a, int n, String e, int i)
String[] result = new String[n + 1];
for(int j = 0; j <= i - 1; j++){ result[j] = a[j]; }
result[i] = e;
for(int j = i + 1; j <= n - 1; j++){ result[j] = a[j-1]; }
return result;

```

Time Complexity: $O(n)$ [linear operation]

34 of 42

Array Case Study: Comparing Two Sorting Strategies



- Problem:

Input: An array a of n numbers $\langle a_1, a_2, \dots, a_n \rangle$

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- We propose two *alternative implementation strategies* for solving this problem.
- At the end, we want to know which one to choose, based on *time complexity*.

35 of 42

Sorting: Strategy 1 – Selection Sort



- Maintain a (initially empty) *sorted portion* of array a .
- From left to right in array a , select and insert *the minimum element* to the end of this sorted portion, so it remains sorted.

```

1 selectionSort(int[] a, int n)
2   for (int i = 0; i <= (n - 2); i++)
3     int minIndex = i;
4     for (int j = i; j <= (n - 1); j++)
5       if (a[j] < a[minIndex]) { minIndex = j; }
6     int temp = a[i];
7     a[i] = a[minIndex];
8     a[minIndex] = temp;

```

- How many times does the body of *for loop* (Line 4) run?
- Running time? [$O(n^2)$]

$\underbrace{n}_{\text{find } \{a[0], \dots, a[n-1]\}} + \underbrace{(n-1)}_{\text{find } \{a[1], \dots, a[n-1]\}} + \dots + \underbrace{2}_{\text{find } \{a[n-2], a[n-1]\}}$

- So selection sort is a *quadratic-time algorithm*.

36 of 42

Sorting: Strategy 2 – Insertion Sort



- Maintain a (initially empty) *sorted portion* of array *a*.
- From left to right in array *a*, insert *one element at a time* into the “right” spot in this sorted portion, so it remains sorted.

```
1 insertionSort(int[] a, int n)
2   for (int i = 1; i < n; i++)
3     int current = a[i];
4     int j = i;
5     while (j > 0 && a[j - 1] > current)
6       a[j] = a[j - 1];
7       j--;
8     a[j] = current;
```

- *while loop* (L5) exits when? $j \leq 0$ or $a[j - 1] \leq \text{current}$
- Running time? $[O(n^2)]$
 $O(\underbrace{1}_{\text{insert into } \{a[0]\}} + \underbrace{2}_{\text{insert into } \{a[0], a[1]\}} + \dots + \underbrace{(n-1)}_{\text{insert into } \{a[0], \dots, a[n-2]\}})$
- So insertion sort is a *quadratic-time algorithm*.

37 of 42

Comparing Insertion & Selection Sorts



- *Asymptotically*, running times of selection sort and insertion sort are both $O(n^2)$.
- We will later see that there exist better algorithms that can perform better than quadratic: $O(n \cdot \log n)$.

39 of 42

Sorting: Alternative Implementations?



- In the Java implementations for *selection* sort and *insertion* sort, we maintain the “sorted portion” from the *left* end.
 - For *selection* sort, we select the *minimum* element from the “unsorted portion” and insert it to the *end* in the “sorted portion”.
- For *insertion* sort, we choose the *left-most* element from the “unsorted portion” and insert it at the “*right spot*” in the “sorted portion”.
- **Question:** Can we modify the Java implementations, so that the “sorted portion” is maintained and grown from the *right* end instead?

38 of 42

Index (1)



- [Algorithm and Data Structure](#)
- [Measuring “Goodness” of an Algorithm](#)
- [Measuring Efficiency of an Algorithm](#)
- [Measure Running Time via Experiments](#)
- [Example Experiment](#)
- [Example Experiment: Detailed Statistics](#)
- [Example Experiment: Visualization](#)
- [Experimental Analysis: Challenges](#)
- [Moving Beyond Experimental Analysis](#)
- [Counting Primitive Operations](#)
- [Example: Counting Primitive Operations](#)
- [From Absolute RT to Relative RT](#)
- [Example: Approx. # of Primitive Operations](#)

40 of 42



Index (2)

[Approximating Running Time as a Function of Input Size](#)
[Focusing on the Worst-Case Input](#)
[What is Asymptotic Analysis?](#)
[Three Notions of Asymptotic Bounds](#)
[Asymptotic Upper Bound: Definition](#)
[Asymptotic Upper Bound: Visualization](#)
[Asymptotic Upper Bound: Example \(1\)](#)
[Asymptotic Upper Bound: Example \(2\)](#)
[Asymptotic Upper Bound: Proposition \(1\)](#)
[Asymptotic Upper Bound: Proposition \(2\)](#)
[Asymptotic Upper Bound: More Examples](#)
[Using Asymptotic Upper Bound Accurately](#)
[Classes of Functions](#)

41 of 42



Index (3)

[Rates of Growth: Comparison](#)
[Upper Bound of Algorithm: Example \(1\)](#)
[Upper Bound of Algorithm: Example \(2\)](#)
[Upper Bound of Algorithm: Example \(3\)](#)
[Upper Bound of Algorithm: Example \(4\)](#)
[Upper Bound of Algorithm: Example \(5\)](#)
[Basic Data Structure: Arrays](#)
[Array Case Study:](#)
[Comparing Two Sorting Strategies](#)
[Sorting: Strategy 1 – Selection Sort](#)
[Sorting: Strategy 2 – Insertion Sort](#)
[Sorting: Alternative Implementations?](#)
[Comparing Insertion & Selection Sorts](#)

42 of 42