CS 2100: Data Structures & Algorithms 1

Basic Sorts (Part I)

Intro to Sorting; Comparable & `compareTo()`; Bubble Sort

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Introduction to Sorting

An Introduction to Sorting

Reminder of Comparable Interface and the compareTo() method

Example of a basic sorting algorithm: Bubble Sort

Sorting

- **PROBLEM**: Given a list (usually an array but it could be a vector or linked list) of things, sort the list
- **INPUT**: An array of things (objects, primitives, whatever…)
- **OUTPUT**: A list of the same things, but in sorted order

- **The sorting problem…**
  - Given a sequence $\mathbf{a_0 \ldots a_n}$ reorder them into a permutation $\mathbf{a'_0 \ldots a'_n}$ such that $\mathbf{a'_i <= a'_{i+1}}$ for all pairs
    - Specifically, this is sorting in non-descending order…
  - Basic operation: Comparison of keys

Sorting

- In computing, we often want to **order** a set of items
  - Find the max/best or min/worst
  - Sort them in order
- Sorting a deck of cards, sorting books, or sorting a collection of numbers are all commonplace examples of sorting algorithm implementations.

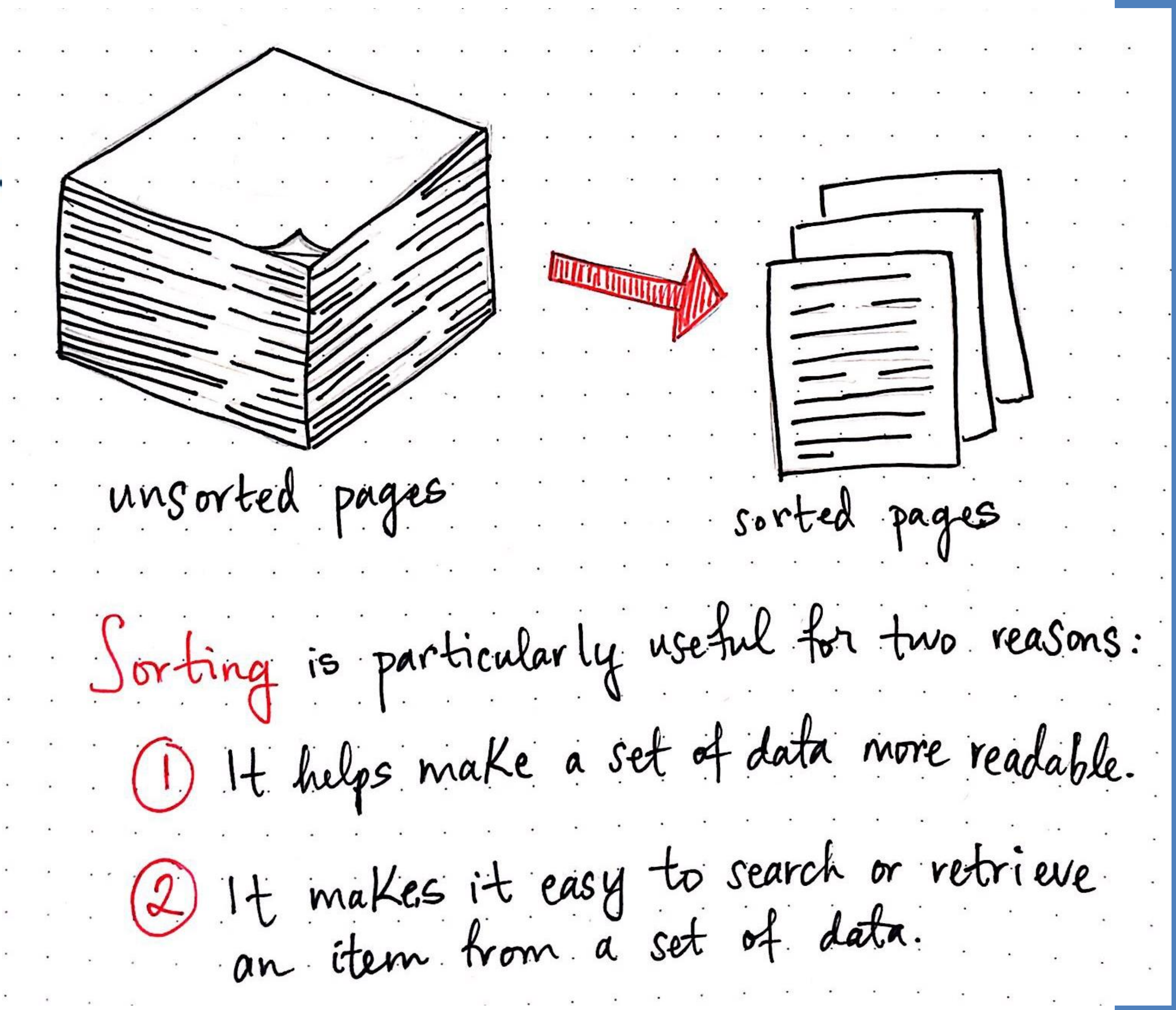


Image ref: medium.com/basecs/ (basics of sorting algorithms)

How to Sort?

- Many sorting algorithms have been found!
  - Problem is a case-study in algorithm design
  - You'll see more of these in CS 2150 and CS 4102

- Some "straightforward" sorting algorithms
  - Insertion Sort, Selection Sort, Bubble Sort
  - **Each is $O(n^2)$**
- More efficient sorting algorithms
  - Quicksort, Mergesort, Heapsort
  - **Each is O(n log n)**

Best Sorts are O(n log n)

Note: these are for sorting in RAM (not on disk)

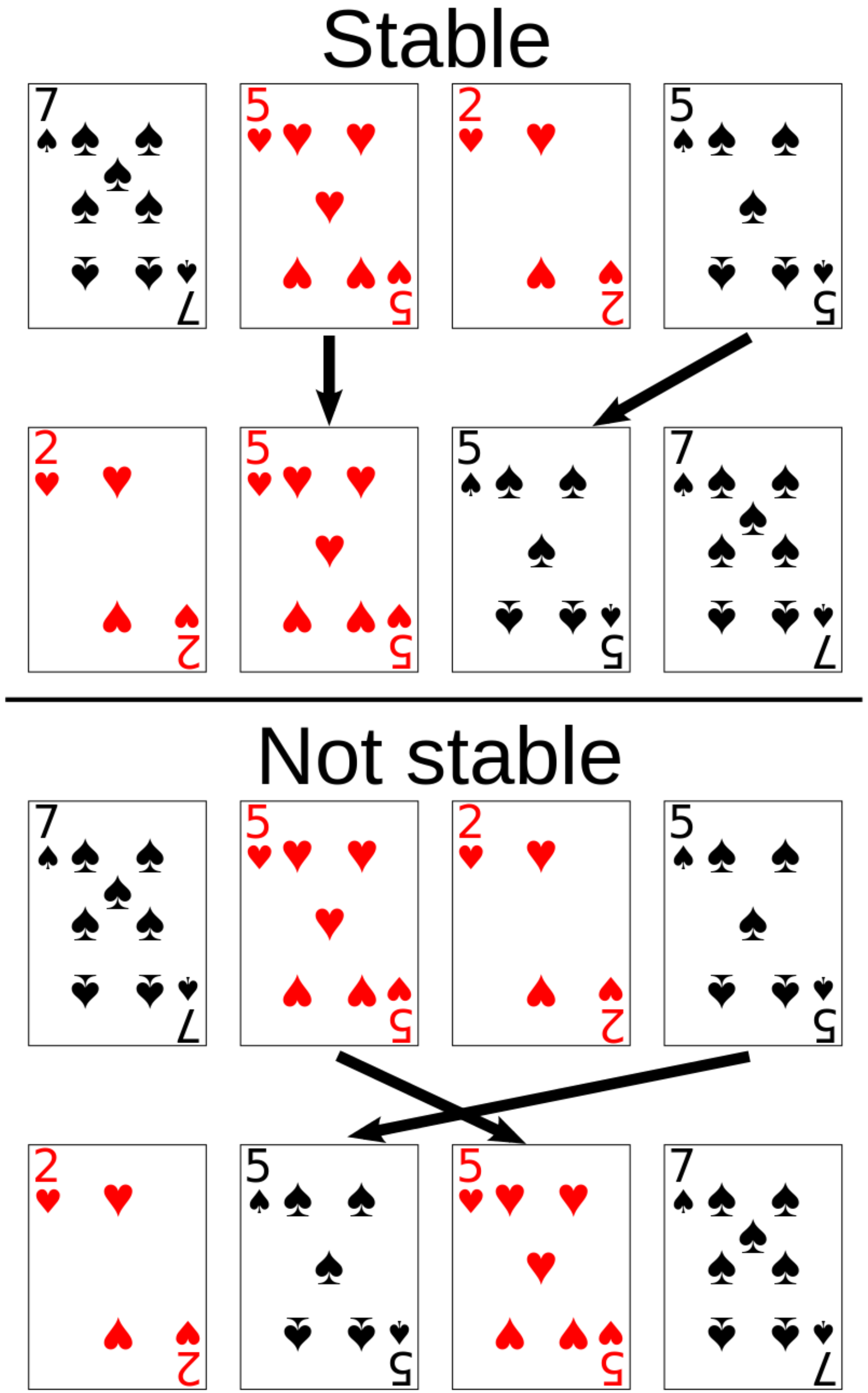
Sorting: Other Requirements

- **REQUIREMENT:** The "things" in the list must have, at a *minimum*, the less than (<) operator defined.
  - **i.e., I can't sort things if I can't tell which are less than others.**
  - In reality, we usually can utilize ***less than***, ***greater than***, and ***equals to*** operators.
  - Java does this through the **Comparable interface**.

Sorting: Other Vocabulary

- **COMPARISON Sorts**: Algorithms that **sort** by making use of direct comparisons (i.e., <= operator) and swapping elements.
- **ADJACENT Sorts**: Algorithms that **sort** by only swapping adjacent elements in the list
  - e.g., **bubble sort** and **insertion sort**
  - ...these are a **subset** of comparison sorts.
- **STABLE Sorts**: A sorting algorithm is **stable** if when two items **x** and **y** occur in the relative order **x,y** in the original list AND **x==y**, then **x** and **y** appear in the same relative order **x,y** in the final sorted list
  - *Thought exercise*: Why would we want this?
- **IN-PLACE Sorts**: A sorting algorithm is **in-place** if the algorithm uses at most Big-Theta(1) extra space (e.g., allocating another array of size **n** is NOT allowed)

Stable Sort Example:



Stable Sort Example:

Tables with students, originally sorted in alphabetical order.

BEFORE	
Name	**Grade**
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Earl	B
Fabian	B
Gill	B
Dave	C

Stable sorting because we **preserved** the initial, alphabetical **order** after we sorted by the **Grade** column.

BEFORE	
Name	**Grade**
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	**Grade**
Greg	A
Harry	A
Gill	B
Fabian	B
Earl	B
Dave	C

Unstable sorting because we **did not preserve** the initial, alphabetical **order** after we sorted by the **Grade** column.

@mgechev

Sorting in Java

- How does **Java** handle sorting?

- Remember the **Java Collections Framework??**

Collections Framework

- The Java Collections Framework is *really*:
  - A common set of operations for "abstract" data structures:
    - List Interface: operations for any kind of list
    - Set Interface: operations for any kind of set
    - Map Interface: operations for any kind of map
  - A set of useful **concrete classes** that we can use:
    - E.g. `ArrayList, HashMap, TreeSet`, …
  - A common set of operations for <u>all</u> Collections:
    - `Collection` **Interface**: operations we can perform on any Collection object
    - `Collections` **Class**: contains *static methods* that can process Collection and List objects

Check out the Collections class

- There are many methods
  - Many have nothing to do with **order**
- We will concentrate on ones relating to order
- In particular `Collections.sort()`!
- ★ Check out the Collections API
- In the JCF there is a Class called Collections
- In this class there is a method called `sort()`
- Collections.sort() requires all objects (classes) it is about to sort to implement the **Comparable interface**, by overriding the stub and implementing the compareTo() method – write it so that Collections.sort() knows how to sort **YOUR** items

Sorting in Java – Collections.sort()

- We want to be able to do something like this:

```
public static< T > void sort(List< T > list);

/* Call the method like this: */
ArrayList< Integer > a = new ArrayList< Integer >();
/* FILL ARRAY WITH LOTS OF STUFF */
Collections.sort(a); //sort it
```

Comparable Interface

- **Collections Framework** provides a `Comparable` interface
  - Defines the *natural ordering* of objects of a class

> *"This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's* ***natural ordering****, and the class's* ***compareTo method*** *is referred to as its natural comparison method."* – Comparable API

Implementing Comparable

Using ***Generics***!

- The `Comparable` **interface** requires only **one** method:
  - `.compareTo(T o)` – compare **this** object to "**o**"
- We must implement the interface and define `T`:

```
public class PhoneBookEntry implements Comparable<PhoneBookEntry> {
    ...
    @Override
    public int compareTo(PhoneBookEntry o) {...}
}
```

Fill in ***actual type***!

- Comparable interface is generic, where you must include the type of the class
- The type inside the <> defines **T**

Implementing Comparable ~ fulfilling the contract

- Implement .compareTo(T o) to fulfill the contract

```
public int compareTo(T o) { … }
```

  - Format: `string1.compareTo(string2) //returns an int`

  - Programming convention: **Return value as follows**:
    - **zero** if the same ~ sameness should be same as .equals()
    - **negative value** if first item strictly **less** than second
    - **positive value** if first item strictly **greater** than second

  - We don't care about the actual value

In Order for Your Items To Be Comparable...

- If you ever want to put your own objects in **Collections**, and use **sort()** you must:
  1. Make your class implement the Comparable interface
  2. Implement (write) the compareTo() method in your class
- How to write **compareTo()**?
  - Think about state-variables that determine natural order
  - Compare them and return proper-value
  - *What makes one of your objects less-than or greater-than the other?*

Example: Student Class

- Student class "*implements*" the Comparable interface: `Comparable<Student>`
- Must fulfil contract: override the compareTo() method stub
- St1.compareTo(St2);
- Body: define the *natural ordering* of the class
- Now that we can say one student is > or < another, we can create a BST of type Student *(otherwise we can't!)*

```java
public class Student implements Comparable<Student> {
    protected String name;
    protected int score;

    public Student (String name, int score) {
        this.name = name;
        this.score = score;
    }

    public String toString() {
        return name + " - " + score;
    }

    @Override
    public int compareTo(Student o) {
        // TODO Auto-generated method stub
        return 0;
    }
```

Requirements For Sorting

- Two requirements for Collections.sort()
- **R1:** The list (the parameter) must implement Java's List< T > interface. The List will definitely be a **collection** of things.
- **R2:** The **items** in the List must implement Java's Comparable interface. This ensures they can be compared to each other.
  - Comparable means that we can always use the `compareTo(Object other)` method to do the actual sorting.

Example: Writing compareTo()

- Imagine something like an entry in a **phonebook**
  - Order by last name, first name, then number

-
```
int compareTo(PhoneBookEntry item2 ) {
     int retVal= this.last.compareTo(item2.last);
     if ( retVal != 0 ) return retVal;
     retVal = this.first.compareTo(item2.first);
     if ( retVal != 0 ) return retVal;
     retVal = this.phNum - item2.phNum;
     return retVal;
}
```

The type is the <u>type of the class</u>! (Not "Object" like the equals() method!)

compareTo() for Strings!

Use of subtraction when dealing with numbers (a primitive) – will still be pos/neg/zero

PhoneBookEntry
last: String first: String phNum: int
compareTo(PhoneBookEntry, item) : int

compareTo() and various types

- Strings:
  - compareTo() with Strings uses alphabetical order to give you an “order” of Strings
  - Format: stringA.compareTo(stringB); // returns an int

- Numbers (ints) – e.g. sort students by score
  - Use subtraction method (*not* compareTo())
    - If “this.score” is 80 and “o.score” is 90
    - this.score – o.score is: 80-90 = -10 (negative)
  - This will sort student scores in ascending order   (Question: how to sort in descending order??)

```java
@Override
public int compareTo(Student o) {
    return this.score - o.score;
}
```

- Object /reference types: use compareTo() !

compareTo() and various types

- booleans: (assume sort "true" before "false" for an "isAutomatic" attribute)
  - Check values for both *[this is only one example of how it can be done]*

Typical way to handle booleans:

```
if(this.isAutomatic == true && other.isAutomatic == false) {
    return -1; // this before other
}
else if (this.isAutomatic == false && other.isAutomatic == true){
    return +1; // this after other
}
else
    return 0; // equal; order doesn't matter
```

- **Another option:**
- `if(this.isAutomatic && !other.isAutomatic) { return -1; }`

```
...
```

Another Example: Sorting People By Height

- If you wish to **sort a List of Person objects** (by height, in this case):

```
public class Person implements Comparable< Person >{
  private int age;
  private double height;

  // Forced to have this method (by interface)
  // Determines the ordering of Persons
  public int compareTo(Person other){
    return this.height - other.height;
  }
}
```

```
ArrayList< Person > p = new ArrayList < Person >();
/* Add a bunch of people objects */

//height is used to sort the objects
Collections.sort(p);
```

Bubble Sort

Bubble Sort

- First sorting algorithm we will look at
- NOT a good choice (efficiency-wise)
- Only showing as an introduction / most basic approach

Bubble Sort

Overall Idea:

- For each pair of adjacent elements, **swap** the bigger one up one position if necessary so that the largest item "bubbles" to the highest index in the list. Repeat **n** times.

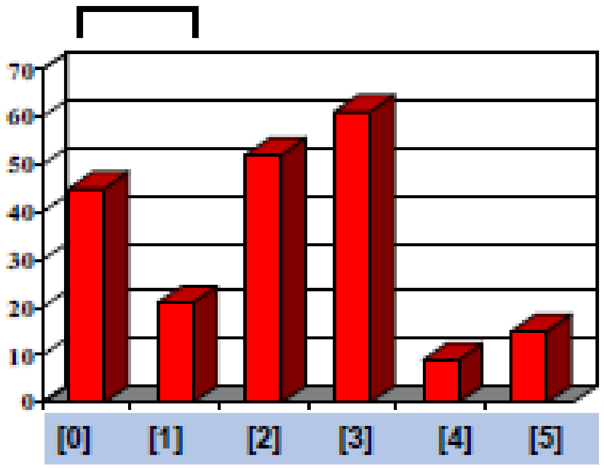
6 5 3 1 8 7 2 4

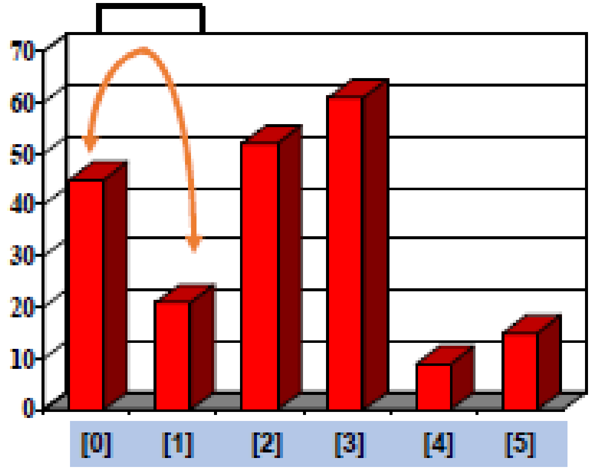
- Bubble Sort Pseudocode:

```
bubbleSort(List list):
  for each i from 0 to n-2
    for each j from 0 to n-i-1
      if list[j] > list[j+1]
        swap list[j] and list[j+1]
```

Bubble Sort

- To sort an array of n elements in ascending order, we use a nested loop:
- The outer loop executes n – 1 times.
- For each iteration of the outer loop, the inner loop steps through all the unsorted elements of the array and does the following:
  - Compares the current element with the next element in the array.
  - If the next element is smaller, it swaps the two elements.





Bubble Sort – Simple Number Example

```
original:   3 9 6 1 2   (underlined=out of order in next pass)
pass 1:
  swap 9 and 6    3 6 9 1 2
  swap 9 and 1    3 6 1 9 2
  swap 9 and 2    3 6 1 2 9
pass 2:
  swap 6 and 1    3 1 6 2 9
  swap 6 and 2    3 1 2 6 9
pass 3:
  swap 3 and 1    1 3 2 6 9
  swap 3 and 2    1 2 3 6 9
pass 4:
  no swaps 1 2 3 6 9    Sorted!
```

Bubble Sort: Analysis

- **Bubble sort is $\Theta(n^2)$.** Why?
- Even worse: Bubble sort will **ALWAYS** do the most amount of work possible.
  - Why? Because the outer and inner loops ALWAYS run completely through. Are never cut short for any reason.
  - This is primarily why bubble sort is **a very bad choice for sorting**.