

CS2040 Lab 3

Sorting

One-Day Assignment 1 – T9 Spelling

- Reminder for reading input: when reading using Scanner's `nextLine()` method after a non-`nextLine()` method (eg. `next()`, `nextInt()`), using an additional `nextLine()` call may be necessary to avoid reading in an empty line
- In your program, do not create more than 1 object to read in input (ie. use only 1 Scanner, or only 1 BufferedReader instead), as both objects “buffer” input by reading in a large chunk of input and saving it in memory
 - See “ScannerInputDemo.java”

Lab 3 – Useful API

- Some sorting methods have already been implemented in Java API
- The sorting methods provided by Java are sufficient for general use.
As such, you may not need to code out your own sorting algorithms

Lab 3 – Useful API

- `Arrays.sort(arr)` will sort a primitive array (eg. `int[]`) *arr* using double-pivot quicksort
- However, if *arr* contains an object instead of primitive data types, it will use a sorting algorithm called TimSort (not examinable)
 - Is stable
 - Not in-place
 - Runs in worst case $O(n \log n)$ time
 - Runs in $O(n)$ time if array is almost sorted
- `Collections.sort(list)` will sort a List using TimSort
 - More on lists in the next lab

Lab 3 – Useful API

- For the sorting methods provided by the Java API to function, it needs to have a way to determine how one element relates to another
 - I.e. when comparing two elements, is the first element smaller than/greater than/equal to the second element?
- Primitive data types (int, double), their associated wrapper classes, and Strings already have this built in
 - Some other Java classes have this too, but you'll likely not use them in this module (eg. Date, Month, Year classes)
- For custom classes, you'll have to add this yourself

Lab 3 – Comparing.java Example

- The program “Comparing.java” is provided as an example of how to code out comparison methods (covered in the next few slides)
- Slides cover the more theoretical parts, which may be a little difficult to understand on their own

Lab 3 – Comparable Interface

- The Comparable interface is used by Java to determine that an object type has a built-in comparison method
 - Also referred to in the Java API documentation as *natural ordering*
- An object type that has a built-in comparison method should implement the Comparable interface
 - Doing so requires the interface's compareTo(T other) method to be implemented as well
 - T is a generic type

Lab 3 – Comparable Interface

- The `compareTo(T other)` method compares two objects: the object on which this method is called (ie. `this`), and the object passed in as a parameter
- The method should return an integer:
 - A negative integer if *this* < *other*
 - Zero if *this*, and *other*, are equivalent
 - A positive integer if *this* > *other*
- See array/list A1/B1 in Comparing.java for an example

Lab 3 – Comparator Interface

- The Comparator interface is another way to compare two objects
- Note that this is in part, a workaround for the Java programming language before Java 8; it did not support function passing then (ie. you can't pass in a function directly as a parameter)
 - Rather, you pass in an object, that contains a function
- The comparator should then be passed as a parameter into the sort() method

Lab 3 – Comparator Interface

- Passing in `Comparator.reverseOrder()` as a comparator will compare elements based on the reverse of the natural ordering
 - As such, the object stored in the array/list must already have implemented `Comparable`
- See array/list A2/B2 in `Comparing.java` for an example

Lab 3 – Comparator Interface

- You can also write a custom Comparator to compare two objects
- Need to implement the `compare(T first, T second)` method
 - The return value is similar to that in `compareTo`:
 - A negative integer if `first < second`
 - Zero if first and second are equivalent
 - A positive integer if `first > second`
- See array/list A3/B3 in `Comparing.java` for an example
 - Array/list A4/B4 is a shortcut of the above (declaring the comparator in the `sort()` method directly
 - Array/list A5/B5 is a shortcut of the above (lambda methods, may be a bit abstract for first-time use; recommended for advanced users)

Lab 3 – Sorting (Arrays)

Method name	Description	Time
<code>Arrays.sort(int[] arr)</code>	Sorts <i>arr</i> using double-pivot quicksort, if <i>arr</i> contains a primitive data type	$O(n \log n)$
<code>Arrays.sort(int[] arr, int start, int end)</code>	Sorts <i>arr</i> using double-pivot quicksort from <i>start</i> (inclusive) to <i>end</i> (exclusive), if <i>arr</i> contains a primitive data type	$O(n \log n)$, where n = size of range

No way to use a comparator for primitive data types

Lab 3 – Sorting (Arrays)

Method name	Description	Time
<code>Arrays.sort(YourClass[] arr)</code>	Sorts <i>arr</i> using Timsort, provided the array contains elements which implement the <i>Comparable</i> interface	$O(n \log n)$
<code>Arrays.sort(YourClass[] arr, int start, int end)</code>	Sorts <i>arr</i> using Timsort from <i>start</i> (inclusive) to <i>end</i> (exclusive), provided the array contains elements which implement the <i>Comparable</i> interface	$O(n \log n)$, where n = size of range
<code>Arrays.sort(YourClass[] arr, Comparator<YourClass> comp)</code>	Sorts <i>arr</i> using Timsort, using the provided comparator	$O(n \log n)$
<code>Arrays.sort(YourClass[] arr, int start, int end, Comparator<YourClass> comp)</code>	Sorts <i>arr</i> using Timsort from <i>start</i> (inclusive) to <i>end</i> (exclusive) using the provided comparator	$O(n \log n)$, where n = size of range

Lab 3 – Sorting (Collections)

Method name	Description	Time
<code>Collections.sort(List<YourClass> list)</code>	Sorts <i>list</i> using Timsort, provided the list contains elements which implement the <i>Comparable</i> interface	$O(n \log n)$
<code>Collections.sort(List<YourClass> list, Comparator<YourClass> comp)</code>	Sorts <i>list</i> using Timsort using the provided comparator	$O(n \log n)$

No way to sort only within a given range for lists using API

Take-Home Assignment 1a – Best Relay Team

- Given a list of runners, and their times as the first runner/subsequent runners, find the team arrangement that would result in the shortest time taken
- Trying all possible permutations of 4 runners would take too long
($500C4$ (choose 4 different runners) * $4C1$ (choose 1 runner to take the first 100m)) = 10 billion+, when $n = 500$
- Can we try to find all permutations of a smaller subset of runners instead?
 - If so, is there an easy way to determine which runners we should consider?

Take-Home Assignment 1b – Card Trading

- Given T card types, their buy/sell prices, and the N initial cards Anthony has in his deck, determine the maximum amount of money that can be earned while keeping at least 2 or more cards for K different card types
- Only one of the following can be done for each card type:
 - Buy up to 2 cards of that type
 - Sell all (owned) cards of that type

Take-Home Assignment 1b – Card Trading

- The "int" data type may be insufficient for this question (its range is up to 2.1 billion); consider using "long" instead
 - Since buy/sell prices can be up to 1 billion, with 100,000 different card types, the maximum answer could be around 200 trillion
- Question asks for a deck with exactly K types of cards which Anthony owns more than 2 of
 - Does Anthony need to have more than 2 cards of any given type?
 - No, having more than 2 cards is unnecessary
 - Should Anthony end up with any card types which he has only one card for?
 - No, as it does not contribute to a combo, this should not be part of the final deck

Take-Home Assignment 1b – Card Trading

- Anthony can start off owning pairs of multiple card types already
 - Should Anthony keep *all* of his starting pairs of cards to form a complete deck?
 - No, it may be possible to sell off some cards which have a high selling price, in order to buy even more cards with a cheaper buying price
 - Otherwise, should Anthony sell off *all* of his starting cards?
 - No, some card types have a low selling price, so it may be better to keep them as a combo instead
 - It might help to consider how much it "costs" to keep a card type as a combo, instead of selling it

One-Day Assignment 2 – Signs

- Given a list of signs (strings)
- Sort them according to the middle letter(s)
 - Guaranteed that each sign has between 1 to 20 letters inclusive
 - Note: the default comparison method for strings uses the entire string
- If two signs have the same middle letter(s), relative to each other, they should appear in the same order as given in the input.

One-Day Assignment 2 – Signs

- How to determine middle letter(s)?
- Case 1: Odd number of characters
 - How to determine the 1 middle character? What can you do with Strings?
- Case 2: Even number of characters
 - How to determine the 2 middle characters? Can I use the same approach as Case 1?

Maybe we can find some useful function in the documentation?

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

One-Day Assignment 2 – Signs

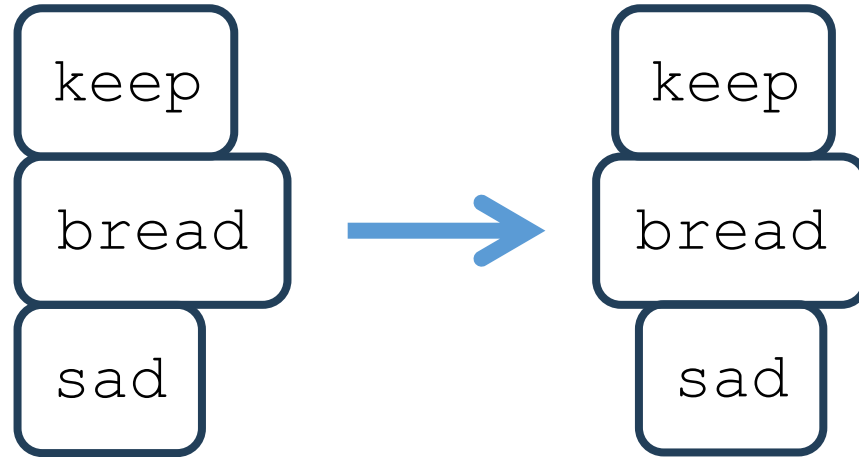
keep

bread

sad

One-Day Assignment 2 – Signs

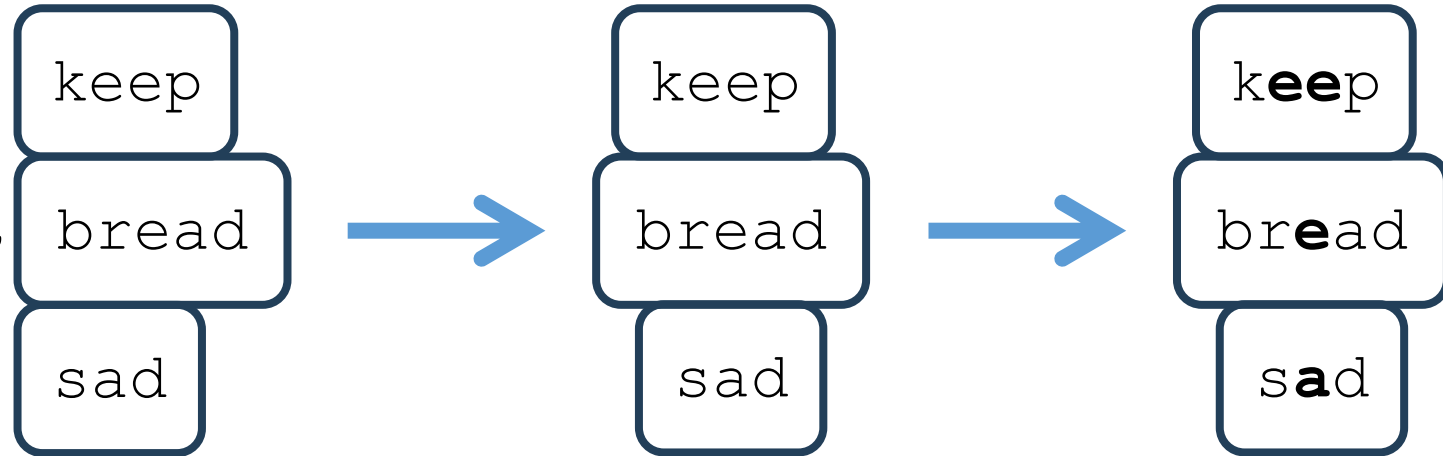
- Center-aligned



One-Day Assignment 2 – Signs

- Center-aligned

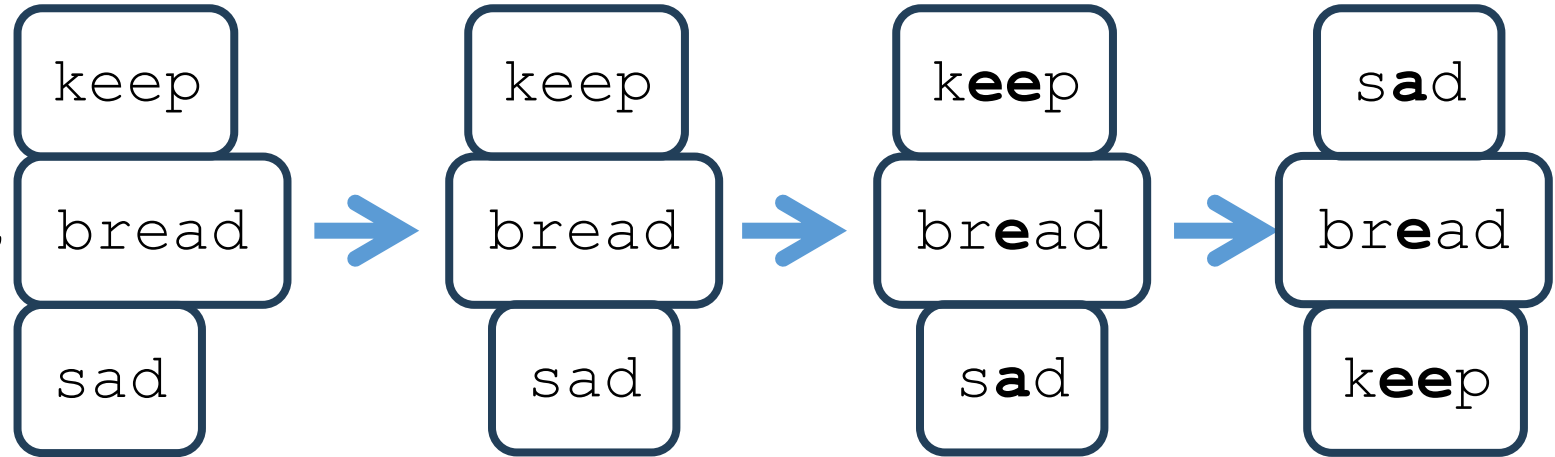
- “Middle letter(s)”



One-Day Assignment 2 – Signs

- Center-aligned

- “Middle letter(s)”



- Sorted alphabetically

One-Day Assignment 2 – Signs

Think about the questions in the following slides. Are they relevant to the problem? Will it affect how your program is written?

One-Day Assignment 2 – Signs

Correctness

- How are strings of different length compared?
- Can the strings be repeated? Will it affect how your program is written?
- Can a sign be empty? Can it contain (white)spaces? – affects I/O
- What happens if signs have same middle letters?

One-Day Assignment 2 – Signs

Efficiency

- Are there many signs?
- Can each sign be very long (i.e. consists of many characters)?
- Choice of “sorting” algorithm? Do we need a stable and/or in-place sorting algorithm? Remember what sort Java uses?

One-Day Assignment 2 – Signs

Tips and planning

- How would you solve the problem on paper, without a computer, for small **N**
- Visualize the input, any intermediate step and/or output if it helps
- Understand problem first before developing algorithm. Start coding only after you have a proper algorithm.