CS2040 Lab 3
Sorting

One-Day Assignment 1 – T9 Spelling

- Reminder regarding output: the format of your output should follow the provided sample output
  - Eg. this question requires that a "Case #X: " be printed first, where X is an integer beginning from the value 1
  - Not doing so will lead to Kattis flagging your answer as incorrect, even if the main part of the question (printing the keypresses) is otherwise correct

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 will usually not need to code out your own sorting algorithms

Lab 3 – Useful API

- Arrays.sort(arr) will sort a primitive array (eg. int[]) *arr* in ascending order 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
  - Ie. 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
Arrays.sort(int[] arr)	Sorts *arr* using double-pivot quicksort, if *arr* contains a primitive data type	O(n log n)
Arrays.sort(int[] arr, int start, int end)	Sorts *arr* using double-pivot quicksort from *start* (inclusive) to *end* (exclusive), if *arr* contains a primitive data type	O(n log n), where n = size of range

"int" can also be "double", "long", "char" etc.
No way to use a comparator for primitive data types

Lab 3 – Sorting (Arrays)

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

Lab 3 – Sorting (Collections)

Method name	Description	Time
Collections.sort(List<YourClass> list)	Sorts *list* using Timsort, provided the list contains elements which implement the *Comparable* interface	O(n log n)
Collections.sort(List<YourClass> list, Comparator<YourClass> comp)	Sorts *list* 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 – 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 1a – 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 1a – 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

Take-Home Assignment 1b – 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?

One-Day Assignment 2 – Sort of Sorting

- Given a list of names (strings)
- Sort them according to the first two letters of their names
  - Guaranteed that name has at least two letters
  - Note: the default comparison method for strings uses the entire string
- Some form of stable sort required

One-Day Assignment 2 – Sort of Sorting

- Again, consider cases which are covered by the sample input, and corner cases which may not be covered:
- Covered:
  - Names with the same first 2 letters:
    - Poincare
    - Pochhammmer
- Not covered:
  - Comparing an uppercase letter with a lowercase letter
    - Zoe
    - amy