CS 200 fall 2022 Lecture 4 (from OO): runtime and sorting

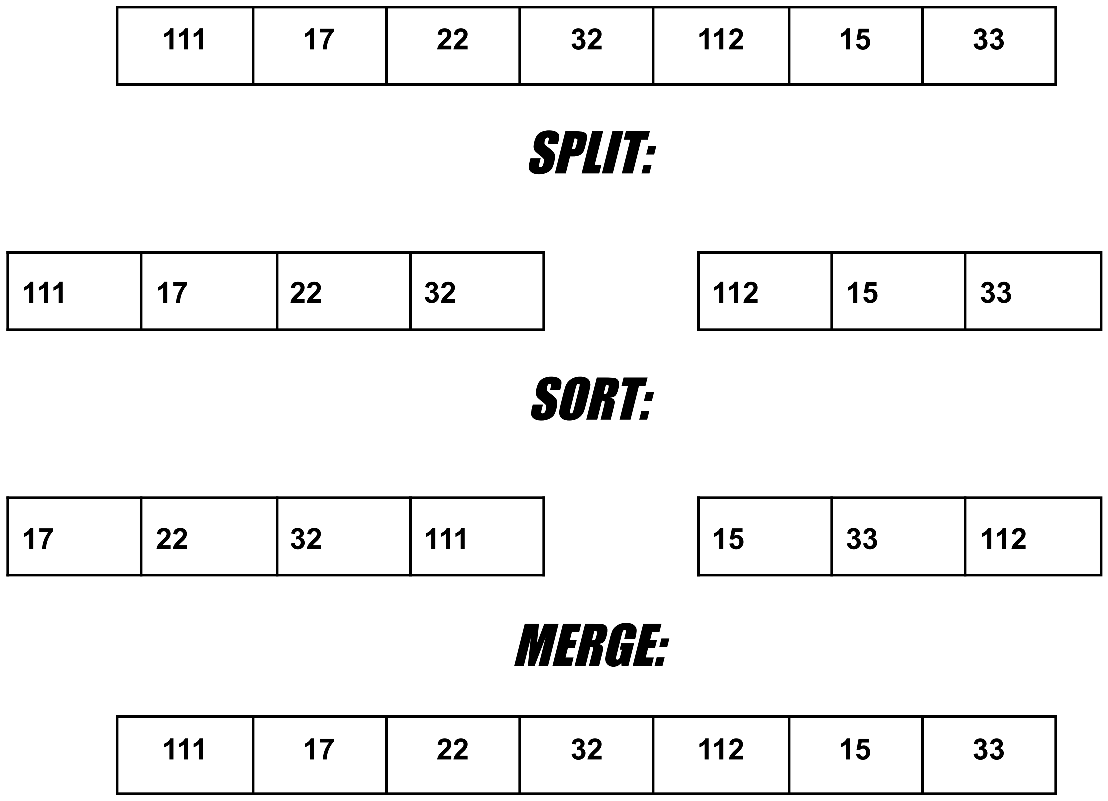
Note: we did some example runtime annotations (from the lecture handout) in the first part of the lecture. We defer to the additional readings linked from the lecture page for more information about runtime.

Mergesort

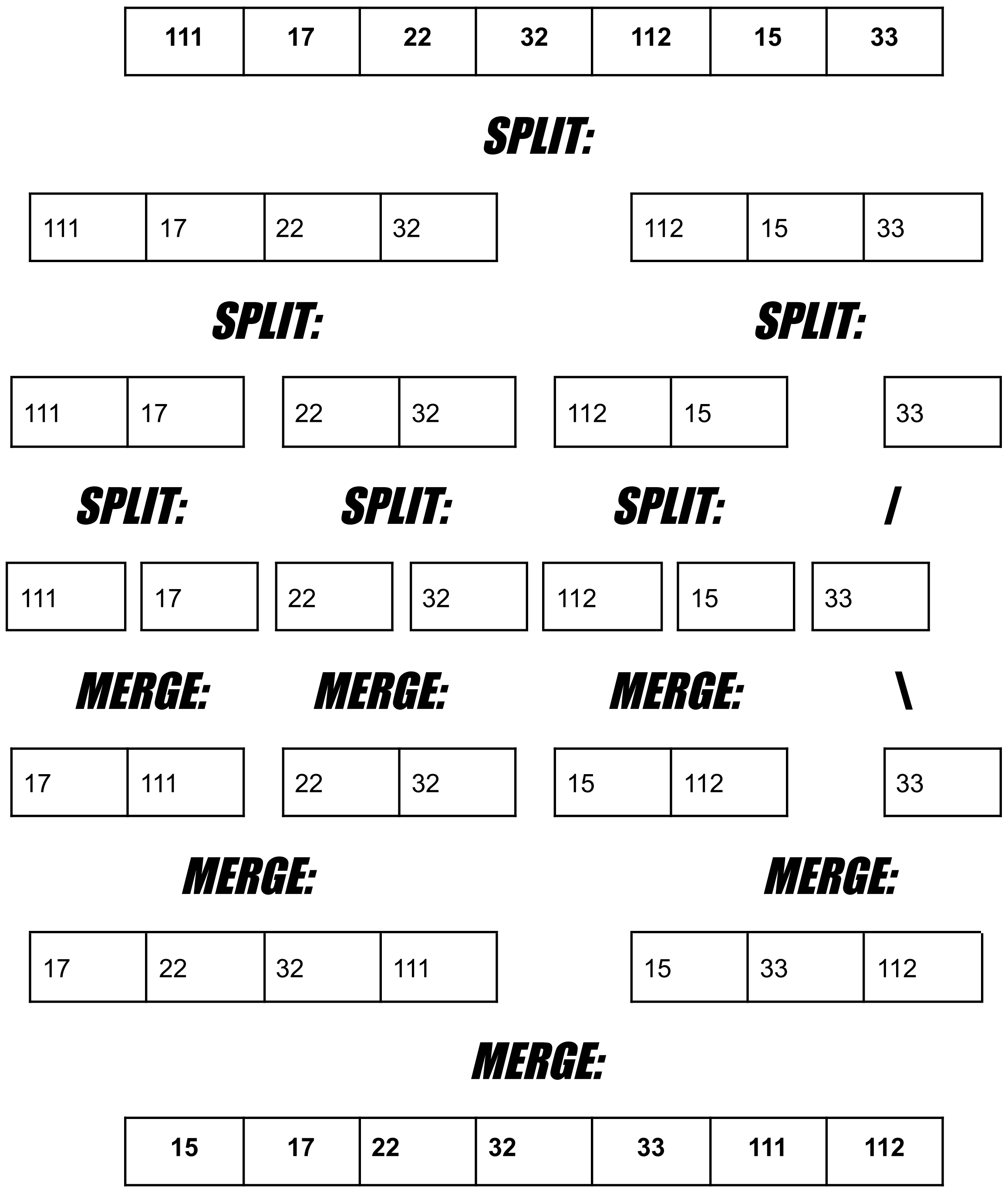
Imagine we wanted to sort the list `111, 17, 22, 32, 112, 15, 33`. You are working with one way of doing this, Quicksort, in your homework. Another way is to take advantage of the gmerging code we wrote in the last lecture and implement an algorithm called *mergesort*.

Background and example

Let's imagine splitting the list down the middle, into a list of the first four elements (`111, 17, 22, 32`) and the last three elements (`112, 15, 33`). If we had a way of sorting these smaller lists, into the lists `17, 22, 32, 111`, and `15, 33, 112`, we could simply call on the merge method we've already written to get a sorted list:



We know how to split a list by calling subList, and we know how to merge a list. How do we sort the smaller list `111, 17, 22, 32`? Well, if we split *that* list down the middle into two lists (`111, 17` and `22, 32`), sorted those two smaller lists, and merged them, we would get a sorted list. To sort the even smaller list `111, 17`, we could split it in two – and get the singleton lists `111` and `17`. Those are already sorted, so we can call merge on them, and get back `17, 111`! The idea is that we keep splitting the sub-lists until we get down to a singleton list or an empty list, which are already sorted, and then merge the smaller lists to get progressively bigger sorted lists:



Mergesort algorithm

Notice that, from the outside in, we're repeating the same action: let's give a name to this idea of splitting a list in half, sorting the halves, and merging the sorted halves. If we call this mergesort, we can notice that the action of sorting the smaller halves is akin to performing mergesort on those two halves. As an example, to mergesort `111, 17, 22, 32, 112, 15, 33`, we split it into its two halves (`111, 17, 22, 32` and `112, 15, 33`), mergesort each half, and merge the sorted halves. Thus, the idea of mergesort can be summarized as:

1. If the input list is of size 1 or 0, return the list (because it's already sorted!)
2. Otherwise, split the list down the middle into two halves
3. Mergesort each half
4. Merge each sorted half and return the result

In code, this looks like:

```
public static List<Integer> mergesort(List<Integer> lst) {
    if (lst.size() <= 1) {
        return new LinkedList<Integer>(lst);
    } else {
        int midpoint = lst.size() / 2;
        List<Integer> firstHalf = lst.subList(0, midpoint);
        List<Integer> secondHalf = ls.subList(midpoint, lst.size());
        List<Integer> sortedFirstHalf = mergesort(firstHalf);
        List<Integer> sortedSecondHalf = mergesort(secondHalf);
        return merge(firstHalf, secondHalf);
    }
}
```

We can simplify the code by getting rid of the intermediate variables:

```
public static List<Integer> mergesort(List<Integer> lst) {
    if (lst.size() <= 1) {
        return new LinkedList<Integer>(lst);
    } else {
        return merge(mergesort(lst.subList(0, lst.size() / 2)),
            lst.subList(lst.size() / 2, lst.size());
    }
}
```

Understanding mergesort recursion

In what order do the mergesort calls run? Let's imagine running mergesort on the list [`5, 4, 3, 2, 1]`:

1. Call `mergesort([5, 4, 3, 2, 1])`
2. Since `[5, 4, 3, 2, 1]` is of length greater than 1, the result depends on the result of running mergesort on `[5, 4]` and `[3, 2, 1]`. The running code cannot return from calling `mergesort([5, 4, 3, 2, 1])` yet, until those recursive calls finish.
3. Call `mergesort([5, 4])`
4. Since `[5, 4]` is of length greater than 1, the result depends on the result of running mergesort on `[5]` and `[4]`. The running code cannot return from calling `mergesort([5, 4])` yet, until those recursive calls finish.
5. Call `mergesort([5])`
6. Since `[5]` is of length 1, the method call simply returns the list `[5]`.
7. The call to `mergesort([5, 4])` is not done yet, because it also depends on `mergesort([4])`
8. Call `mergesort([4])`
9. Since `[4]` is of length 1, the method call simply returns the list `[4]`.
10. The call to `mergesort([5, 4])` can finish, by calling merge on the lists `[5]` and `[4]` (the results of steps 6 and 9) and returns `[4, 5]`
11. The call to `mergesort([5, 4, 3, 2, 1])` is not done yet, because it also depends on `mergesort([3, 2, 1])`
12. Call `mergesort([3, 2, 1])`
13. Since `[3, 2, 1]` is of length greater than 1, the result depends on the result of running mergesort on `[3, 2]` and `[1]`. The running code cannot return from calling `mergesort([3, 2, 1])` yet, until those recursive calls finish.
14. Call `mergesort([3, 2])`
15. Since `[3, 2]` is of length greater than 1, the result depends on the result of running mergesort on `[3]` and `[2]`. The running code cannot return from calling `mergesort([3, 2])` yet, until those recursive calls finish.
16. Call `mergesort([3])`
17. Since `[3]`is of length 1, the method call simply returns the list `[3]`.
18. The call to `mergesort([3, 2])` is not done yet, because it also depends on `mergesort([2])`
19. Call `mergesort([2])`
20. Since `[2]` is of length 1, the method call simply returns the list `[2]`.
21. The call to `mergesort([3, 2])` can finish, by calling merge on the lists `[3]` and `[2]` (the results of steps 17 and 20) and returns `[2, 3]`
22. The call to `mergesort([3, 2, 1])` is not done yet, because it also depends on `mergesort([1])`

23. Call `mergesort([1])`
24. Since `[1]` is of length 1, the method call simply returns the list `[1]`
25. The call to `mergesort([3, 2, 1])` can finish, by calling merge on the lists `[2, 3]` and `[1]` (the results of steps 21 and 24) and returns `[1, 2, 3]`
26. The call to `mergesort([5, 4, 3, 2, 1])` can finish, by calling merge on the lists `[4, 5]` and `[1, 2, 3]` (the results of steps 10 and 25) and returns `[1, 2, 3, 4, 5]`

Runtime of mergesort

The runtime of mergesort can be expressed as recurrence, because it is a recursive method. It will also depend on the runtime of merge.

Stop and think: If the input list, to mergesort is of length n, calling merge on the two halves of that input list will run in O(N). Why is that?

The worst case runtime of mergesort is through the else-statement (**stop and think: why is the base case not the worst case?**). subList will run in O(N) time, and the method also does some constant-time (O(1)) operations to compute the midpoint of the list and call on subList and other methods. It also calles mergesort twice, both times on an input of size N/2 (**stop and think: verify this for yourself.**), and then calls on merge (an O(N) operation). Two O(N) operations will still have an O(N) total runtime (**stop and think: why?**) Hence, if we use $T_{ms}(N)$ to represent the time to run mergesort on a list of length N, we have

$$T_{ms}(N) = O(1) + O(N) + 2 * T_{ms}(N/2)$$

The closed-form of this recurrence relation turns out to be *O(N * log N)*. You can see a more detailed explanation in this book chapter. Intuitively, however, this makes sense – for each step of mergesort, we are doing some operation in linear time (the *N* of *N * log N*) and some operation on a problem half the size (the *log N* of *N * log N*).