Data Structures and Algorithms (w/ Java)

Muchang Bahng

Spring 2023

When a program is build and run, we must worry about two computational overheads:

1. the runtime of the program, which is determined by the efficiency of the algorithm itself
2. the memory overhead of the program, which is determined by the types of data structures used.

When we want to optimize these programs, we therefore want to look at their data structures and algorithms.

Note that Java has the following **primitive types**: `int`, `long`, `float`, `double`, `boolean`, `char`. Everything else is a **reference type**.

0.1 Time Complexity and Memory Allocation

We can think of our RAM as storing our memory. Whenever we call the `new` keyword, we allocate new memory for whatever object we are storing. Say that an `int` stores 4 bytes each, and we create an array of ten integers.

```
int[] numbers = new int[10];
```

This array would take up 40 bytes of memory, and to access this part of the memory, we create a variable `numbers` that really just stores the location of the object in memory. Since variables are just references to memory locations, we can have multiple variables referencing the same object. Note that we did not use the `new` keyword here.

```
int[] stuff = numbers;
```

Therefore, both `numbers` and `stuff` points to the same array, and whatever we do with one is done to the other. If we wanted to create two distinct arrays in memory, we can do this:

```
int[] numbers = new int[10];
int[] stuff = new int[10];
```

which would take up 80 bytes of memory. Therefore, we must distinguish the *actual object in memory* and the *variable that references the object*. Say that the variable is `myList` and the object that it references to is `ArrayList<String> Object`. Then, we will denote it shorthand as

$$\texttt{myList} \mapsto \texttt{ArrayList<String> Object}$$

This is all simple enough, but it gets a bit more confusing when we talk about how functions act. When we have some function `func` and input variable, say $\texttt{myList} \mapsto \texttt{ArrayList<String> Object}$, the function create a *copy of the reference* in the backend, also named `myList` (confusingly) and acts on this copy. Note now that we are working with two references.

Therefore, the function is allowed to modify `ArrayList<String> Object` through the copied reference variable.

Example 0.1 (Modifying Referred Object)**.** Given the function

```
public static void removeFront(List<String> words) {
    words.remove(0); // this 'words' is a copy of the reference
}

List<String> words = new LinkedList<>();
words.add("CS");
removeFront(words);
System.out.println(words); // prints [] (empty)
```

if `words` references an array of 1 million elements, then `removeFront` will not copy the entire array and allocate new memory, but it will just act on the original array.

However, it cannot modify the reference of the original variable, and so you cannot "lose" a reference inside a method.

Example 0.2. Given the function

```
public static void tryBreakReference(List<String> words) {
    words = new LinkedList<>(); // this 'words' is a copy of the reference
}

List<String> words = new LinkedList<>();
words.add("CS");
tryBreakReference(words);
System.out.println(words); // Still prints ["CS"]
```

This does not affect the object that the original `words` is referencing to since the function takes only the *copied reference variable* and changes its reference to the new LinkedList.

The default value for an uninitialized (no memory allocated by call to `new`) object is `null`. If you try to call any methods to a null object, you will get a **null pointer exception error**.

Example 0.3. Let's go through an exercise. Given the code

```
public static ListNode foo(ListNode list) {
    list = list.next;
    list.next = null;
    return list;
}

public static void main(String[] args) {
    ListNode list = new ListNode(2,  new ListNode(0, new ListNode(1)));
    ListNode ret  = foo(list);
    printList(ret);     // prints 0
    printList(list);    // prints 2, 0
}
```

Let us examine the behavior of it. Note that we allocate memory for `list`, which we will denote $\texttt{list}_m$ and when we input it into `foo`, we get a copy of the reference, denoted $\texttt{list}_c$.

`foo` takes $\texttt{list}_c$ and has it reference the next value, which is 0. Therefore, $\texttt{list}_m$ is $2, 0, 1$ and $\texttt{list}_c$ is $0, 1$.

Now, we set the next value of $\texttt{list}_c$ to null, and since this modifies the LinkedList object, we have both $\texttt{list}_m$ is $2, 0$ and $\texttt{list}_c$ is 0.

0.2 Arrays

Let us take a look at the most basic data structure in Java: the array. They are mutable, homogeneous (can only store one type), and fixed in size. We can print their outputs by converting them to a string, as such with the static method:

```
System.out.println(Arrays.toString(arr));
```

0.3 Classes and Objects

Just like in other object oriented languages, Java uses classes, and everything must be used in a class. Here, we have a class with static and dynamic variables, static and dynamic methods, and constructors.

```
public class Point {
  public static String creator = "Muchang"; // static variable

  public int x; // dynamic variable
  public int y; // dynamic variable

  public Point (int x, int y) { // constructor
    this.x = x;
    this.y = y;
  }

  public Point (Point p) { // constructor
    this.x = p.x;
    this.y = p.y;
  }

  public double distanceFrom(Point other) { // dynamic method
    return Math.sqrt((x - other.x)*(x - other.x) + (y - other.y)*(y - other.y));
  }

  public static void greet() { // static method
    System.out.println("I am a point!");
  }
}
```

0.3.1 Public vs Private Variables and Attributes

Note that every single variable and method had the `public` keyword, which allows users to read/modify the variables and run the methods, *even outside the class it is defined in.* If we switched them to `private`, then we could still access them within the code, but we would not be able to read/modify them elsewhere. This is particularly useful for when we are dealing with sensitive information, and if we do want to grant users the power to read/modify them, we can use separate public getter and setter methods.

```
public class Point {
  private String name;
  public int x;
  private int y;

  public Point (String name, int x, int y) {
    this.name = name;
    this.x = x;
    this.y = y;
  }
```

```
  public getName() {    // getter method
    return this.name;
  }

  public setName(String newName) {  // setter method
    this.name = newName;
  }

  private void increment() {
    this.x ++;
    this.y ++;
  }

  public static void main(String[] args) {
    Point p = new Point(1, 2);
    System.out.println(p.x);
    System.out.println(p.y); // still accessible since within the class
    p.increment(); // still runnable since within the class
    System.out.println(p.x);
    System.out.println(p.y);
  }
}
```

0.3.2 Inheritance

0.3.3 .equals() method

Generally for objects, we should use the `.equals()` method. It must be implemented for the given class. If it is not implemented, then the `.equals()` just checks memory locations, and so calling `p = new Point(0.0, 0.0)` and `q = new Point(0.0, 0.0)` and comparing them with `p.equals(q)` would return false. When we also create an array of a certain object, we must create the actual objects in the array by calling new. For example, we don't even create the Point objects in an initialized array.

```
Point[] pointArray = new Point[5];  // creates the array and allocates memory
System.out.println(pointArray[0].x) // but didn't actually create the points, so error
```

We could also store the same references to the variables.

```
ArrayList<Point> myPoints = new ArrayList<>();
Point p = new Point(2.0, 2.0);
myPoints.add(p);
p.x = 3.4;
myPoints.add(p);

// This creates the ArrayList of form (p, p), which both reference
// the same Point object, so both of form (3.4, 2.0), (3.4, 2.0).
```

So, we should call new for every Point object.

```
ArrayList<Point> myPoints = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    myPoints.add(new Point(0.0, 0.0));
}
Point p = new Point(0.0, 0.0);
System.out.println(pointList.contains(p));
```

The final line can print either true or false:

1. It prints false if there is no .equals() method implemented in the Point class. `.contains()` uses the `.equals()` method on every element of the ArrayList by running a for loop, so it will use the default implementation of checking references.
2. If `.equals()` is implemented as before, it will check the values of x and y, so it will print true.

1 Collections Interface

Interfaces in Java specify functionality by specifying what methods exist. At almost the top, we have the **collections** interface, which represents "a group of stuff." It is divided into 3 main subinterfaces: Lists, Sets, and Maps (along with many others), which add more functionality. So, an interface represents the functionality of whatever we will create, while the implementation is the actual concrete class.

1.1 List Interface

We want to define a list that has the following methods given some list `lst`.

1. `Object y = lst.get(0)` outputs the 0th element.
2. `boolean y = lst.contains(elem)` checks if `elem`
3. `lst.add(Object elem)` adds the element to the end of the list.
4. `lst.remove(Object elem)` removes the element.
5. `lst.size()` returns size of the list.

Definition 1.1 (ArrayList)**.** An ArrayList just implements an array in the backend but with some extra systematic way to dynamically grow. If we add to an array, we either have space and can do it, or we don't and can't. If we add to an ArrayList, we can

1. simply add to the first open position if there's space left, $O(1)$
2. we grow the size of the ArrayList by creating a new larger array, copying everything, and then adding to the first open position. (linear time $O(n)$), since we have to add all the elements to the new array.

Starting with a length 1 array, if we add N elements one at a time and when full, create a new array that is

1. is twice as large (geometric growth: 1, 2, 4, 8, 16, ...). Then, we must copy at sizes 1, 2, 4, 8, ... and the total values copied looks like

$$1 + 2 + 4 + \ldots + (N/4) + (N/2) = N - 1$$

   This is what the Java.util.ArrayList implements, and you can see the performance of adding is $O(N)$.

2. has 1 more position (arithmetic growth: 1, 2, 3, ...). Then, we must copy at sizes 1, 2, 3, 4, ... and the total values copied looks like

$$1 + 2 + 3 + \ldots + (N - 1) = N(N - 1)/2$$

   If the arithmetic pattern is adding, say 1,000,000 elements, then we are wasteful of memory.

This geometric growth is a good tradeoff between performance and memory usage. It never uses more than twice the memory of an array in order to store it. Furthermore, the runtime of a geometric growth pattern is amortized constant time, which means that it is constant when averaged over a long time. This is because the vast majority of these operations are constant time, with a few add operations which require resizing to be longer. But these few ones happen less and less frequently that when averaged over a long period, we can treat it as constant.

One thing to note is that while adding to the end of an ArrayList can be efficient, adding to the front is not since it must shift the entire Array, even if there is space left.

Theorem 1.1 (ArrayList Runtime Complexity)**.** *The following are true for ArrayList* `lst`*:*

1. *Getting and Contains*
   (a) `lst.get(int index)` *is* $O(1)$.
   (b) *Getting every element is* $O(n)$
   (c) `lst.contains(Object elem)` *is* $O(n)$
2. *Adding*
   (a) `lst.add(Object elem)` *is amortized* $O(1)$.
   (b) `lst.add(0, Object elem)` *is* $O(n)$.
   (c) `lst.add(int index, Object elem)` *is on average* $O(n)$.
3. *Removing*
   (a) `lst.remove(0)` *is* $O(n)$
   (b) `lst.remove(int index)` *is* $O(n)$
   (c) `lst.remove(lst.size() - 1)` *is* $O(1)$
   (d) `lst.remove(Object elem)`

Proof. Listed.

1. Getting the element at index `index` requires us to just look at the same index in the underlying array, which is $O(1)$.
2. We loop through each element of the ArrayList and call `.equals(elem)` at each step, which results in $O(1)$.
3. Since the geometric growth of the ArrayList happens exponentially less frequently, it averages out to be $O(1)$, so amortized.
4. Adding at a specific index requires $O(n)$ since we create a new ArrayList and copy over all the elements with the added element.
5. Removing an object requires us to shift the indices of the remaining elements by 1, so this is $O(n)$.

■

Example 1.1 (String)**.** The string type is just an ArrayList of characters. It has the following attributes and methods. Let `x = "I love CS201"`

1. `int y = x.length()` outputs the length and is $O(1)$
2. `char y = x.charAt(0)` outputs a character and is $O(1)$
3. `String y = x.substring(0, 4)`
4. `boolean y = x.equals("I love CS201")`
5. `String y = x + "!!"`
6. `String[] y = x.split(" ")`
7. `String y = String.join(" ", words)`

Definition 1.2 (Linked List)**.** A linked list contains a sequence of nodes that each contain an object for its element and a reference to the next node. More specifically, it can be divided up into 3 parts:

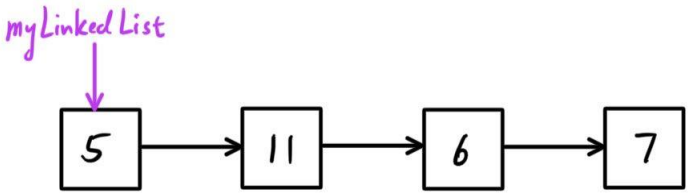
1. The variable which points to the *first* node. This can be confusing since this variable, which represents an *entire list* is just a pointer to the first node.
2. A sequence of nodes containing the element and a reference to the next node.
3. The final node containing the element and a reference to `null`.

We can implement these functionalities in the `ListNode` class, which are used to build a linked list of integers.

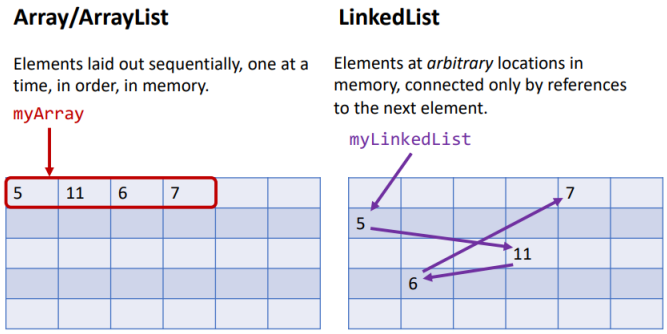
```
public class ListNode {
    int info;           // value i.e. element
    ListNode next;      // reference to next ListNode

    ListNode(int x) {
        info = x;
    }
    ListNode(int x, ListNode node) {
        info = x;
        next = node;
    }
}
```

The following diagram represents a linked list.



But in reality, the elements are all located random in memory and can only be found by references.



To print everything in a linked list, we just loop over the nodes as long as the nodes are not null.

```
public static void printList(ListNode list) {
    while (list != null) {          // common conditional for traversing
        System.out.println(list.info);
        list = list.next;
    }
}
```

Theorem 1.2 (Linked List Runtime Complexity)**.** *The following are true for a basic LinkedList* `lst`*:*

1. *Getting and contains*
   (a) `lst.get(int index)` *is* $O(n)$ *on average (unless you get the first index, which is fast).*
   (b) *Getting every element in the list is* $O(n^2)$*.*
   (c) `lst.contains(Object elem)` *is* $O(n)$
2. *Adding*
   (a) *Start:* `lst.add(0, Object elem)` *is* $O(1)$
   (b) *Middle:* `lst.add(int index, Object elem)` *is on average* $O(n)$*.*
   (c) *End:* `lst.add(Object elem)` *is* $O(n)$
3. *Removing*
   (a) *Start:* `lst.remove(0)` *or* `lst.remove()` *is* $O(1)$
   (b) *Middle:* `lst.remove(int index)` *is on average* $O(n)$
   (c) *End:* `lst.remove(lst.size() - 1)` *is* $O(n)$

Proof. Listed.

1. We must traverse from the beginning of the linked list, and so it is $O(n)$. If we just pay attention to the first (or last, for doubly-linked list) element, then this is just $O(1)$.
2. Getting every element is just looping an $O(n)$ operation n times, so $O(n^2)$.
3. You need to iterate through each element and call `.equals(elem)`, so it is $O(n)$.
4. We can simply take the reference
5. Adding

■

Even though our basic LinkedList solves the problem of adding in the beginning, in order to add in the middle or end, we must get to that position (which is $O(n)$ time) before we are able to utilize our $O(1)$ add. This is quite inefficient, especially when we do repeated adding, so we should keep track of certain "markers" that indicate where our current node is. **Iterators** do this naturally, so we would like to implement some current notion of position.Below we implement a new linked list (of integers).

Definition 1.3 (Iterator)**.** An **iterator** is a Java interface that has the two methods:

1. `.hasnext()` checks if there is an element after the current one.
2. `.next()` prints out the next element.

We want to implement iterators to any collections or whatever custom class if we want to be able to use enhanced for loops over them.

Definition 1.4 (DIYLinkedList)**.** Note the following:

1. Adding to the end (`add`) and to the front (`addtoFront`) are both $O(1)$, since we always have access to the dynamic attributes `first` and `last`.

```
public class DIYLinkedList implements Iterable<Integer> {
    private class ListNode {
        int value;
        ListNode next;
        public ListNode(int value) {
            this.value = value;
        }
        public ListNode(int value, ListNode next) {
            this.value = value;
            this.next = next;
        }
    }

    private ListNode first;
    private ListNode last;
    private int size;

    public DIYLinkedList() {
        size = 0;
    }

    public int size() {
        return size;
    }

    public int get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        current = first;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current.value;
    }

    public void add(int elem) {
        // add to end
        if (last == null) {
            last = new ListNode(elem);
            first = last;
        }
        else {
            last.next = new ListNode(elem);
            last = last.next;
        }
        size++;
    }

    public void addToFront(int element) {
        // add to front
        first = new ListNode(element, first);
        size++;
    }
```

```
    private class DIYListIterator implements Iterator<Integer> {
        ListNode current = first;

        @Override
        public boolean hasNext() {
            return current != null;
        }

        @Override
        public Integer next() {
            int value = current.value;
            current = current.next;
            return value;
        }
    }

    @Override
    public Iterator<Integer> iterator() {
        return new DIYListIterator();
    }
}
```

Theorem 1.3 (Appending)**.** *Appending two ListNodes (of size n and m) is $O(n)$ time.*

```
public static ListNode append(ListNode listA, ListNode listB) {
    ListNode first = listA;
    while (listA.next != null) {
        listA = listA.next;
    }
    listA.next = listB;

    return first;
}
```

Theorem 1.4 (Reversing)**.** *When we reverse a linked list, we want to work with it one step at a time by establishing a **loop invariant**, which is just some condition that we want to be true every iteration. In this case, our invariant is "after k iterations, `rev` points to the reverse of the first k nodes."*

```
public ListNode reverse(ListNode front) {
    ListNode rev = null;
    ListNode list = front;
    while (list != null) {
        ListNode temp = list.next;
        list.next = rev;
        rev = list;
        list = temp;
    }
    return rev;
}
```

Example 1.2. Here are three reversing examples, in increasing difficulty:

1. If `front` is a ListNode with `front.next == null`, then `reverse(front)` will return

$$\texttt{reverse(front)} \mapsto \texttt{front} \mapsto \texttt{null}$$

2. If we have a linked list `list` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null`, then `reverse(list.next)` will return

$$\texttt{reverse(list.next)} \mapsto 3 \mapsto 2 \mapsto \texttt{null}$$

3. If we have a linked list `list` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null`, then after running `reverse(list.next)`, the original list variable will be

$$\texttt{list} \mapsto 1 \mapsto 2 \mapsto \texttt{null}$$

   This is because after the method call, we have $3 \mapsto 2 \mapsto$ `null`, but the 1 still points to 2! Therefore, the original list, which points to 1, will point to 2, which points to `null`.

1.2 Set Interface

Sets are collections that are unordered and store unique elements. We want to define a set that has the following methods given some set `st`.

1. `boolean y = st.contains(elem)` checks if `elem` is in set.
2. `st.add(Object elem)` adds element and returns false if already there.
3. `st.remove(Object elem)` removes element.
4. `st.size()` returns size of the list.

We can loop over a set not with a regular for loop, but with an enhanced for loop (since sets implement iterables but are not ordered by index).

Theorem 1.5. *We can also convert between lists and sets by taking an empty ArrayList and using the .addAll() method.*

```
List<String> myList = new ArrayList<>();
myList.addAll(mySet);
```

Definition 1.5 (HashSet)**.** The HashSet implementation offers constant time performacne for basic operations (add, remove, contains, size), under some assumptions. To count unique words in an Array of Strings, using a HashSet is much much faster than using ArrayLists, since the ArrayList code uses the contains function, which itself is linear.

1.3 Map Interface

Maps pair keys with values, like a dictionary. We want to define a map that has the following methods given some map `mp`.

1. `boolean y = mp.containsKey(k)` checks if `k` is a key in the map.
2. `mp.get(Object k)` returns the value associated with key `k`.
3. `mp.put(Object k, Object v` adds the key value pair `k`, `v`.

We can also loop over a map with an enhanced for loop.

Definition 1.6 (Hash Tables)**.** A hash table is an array of key value pairs. But rather than adding to positions in order from 0, 1, 2, ..., we will calculate the hash of the key, which would return an int that specifies where we store this key-value pair. So to store, `<"ok", 8>`, we will calculate `hash("ok") == 5` and store it in the 5th index.

```
0
1
2 <"hi", 5>
3
4
5 <"ok", 8>
6
7
```

We can immediately see how this makes search easier, since if we want to find the value associated with the key "ok", then we can calculate the hash of it to find the index and look it up on the array. Java implements this with the `.hashCode()` method on the key. More specifically, to get this index, we get the hash, we calculate `Math.abs(key.hashCode()) % list.size()` (remember to take the modulus to get the index between 0 and the list size).

1. To `put(key, value)`, we compute `hash(key)` and add it in that index. If there is already a key-value pair there, then update the value.
2. To `get(key)`, we compute `hash(key)` and retrieve it from the index.
3. To `containsKey(key)`, we compute `hash(key)` and check if the key exists at the index hash position of the list.

So running `get(key)` on a HashMap looks up position `hash(key)` in the hash table and returns the value there. Immediately, we see that if `hash` is not injective (which it isn't), then we can run into collisions. This is solved using chaining or bucketing. Bucketing basically takes each index in the array and stores not just one key-value pair, but a list of key-value pairs. So basically, when we want to search for the value of a key, we compute the index of it with `hash(key)`, which would return a list of key-value pairs. Java would iterate through these keys and call the `.equals()` method to compare them. This can be a problem if you are dealing with keys that are custom built classes.

1. This means that if the key types are something that you've custom built, then you must define the `.equals()` methods in them! You must override and implement `.equals()`.
2. We must also custom implement the `.hashCode()` method.

This is optional, but we can override the `toString()` method to print something we want. Obviously, if we create a custom `hashCode()` method that trivially maps to 0, then we would just have one giant list in the bucket at index 0, which is no more efficient than a list search. So, we should ideally assume that given N pairs with M buckets, our hashing function is built so that the probability of two random (unequal keys) hash to the same bucket is 1/M. Note that this hash function is completely deterministic. We should talk about runtime/memory tradeoff. Given N pairs and M buckets (with SUHA):

1. $N >> M$ means too many pairs in too few buckets, so runtime inefficient
2. $M >> N$ means too many buckets for too few pairs, so memory wasteful
3. M slightly larger than N is the sweet spot.

To maintain an ideal ratio, we basically create a new larger table (with geometric resizing) and rehash/copy everything until we reach it.

2 Algorithms

2.1 Iteration vs Recursion

There are two general paradigms in which we can code algorithms. So far, we have taken the iterative approach, which works with functions that simply loop over a collection.

Example 2.1 (Iteration)**.** The following is an iterative method to count all the elements in a LinkedList.

```
public int countIter(ListNode list) {
    int total = 0;
    while (list != null) {
        total++;
        list = list.next;
    }
    return total;
}
```

A recursive approach of constructing the same function requires us to do two steps:

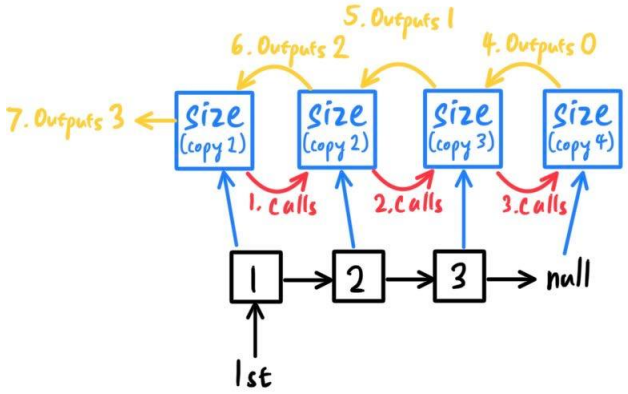
1. Consider the base case, like in an induction proof.
2. Assuming that the function can be solved for some subset of the input (what Fain refers to as "the oracle"), determine what we should do with the result of the recursive call.

An important thing to note is that the method does not call itself. It calls an identical clone, with its own state, which makes up what we call the **call stack**. Each local call gets its own call frame, with its own local variables, etc., and invoking the method does not resume until invoked method returns, a condition we call **eager evaluation**.

Example 2.2 (Recursion)**.** The following is a recursive method to count all the elements in a linked list.

```
public int size(ListNode list) {
    if (list == null) return 0;
    return 1 + size(list.next);
}
```

Doing this on, say a linked list `lst` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null` gives us the following diagram:



Note that we must ensure that every recursive call gets closer to the base case, or this may never end. Speaking of runtime, let us state a method to compute the runtime complexity of recursive algorithms.

Theorem 2.1 (Runtime Complexity of Recursive Algorithms)**.** *To compute the runtime complexity, we must consider two things:*

1. *At what rate are we approaching the base case?*
2. *How long each recursion takes.*

Once we know these two, we can simply multiply them.

Example 2.3 (Reverse)**.** We can reverse a LinkedList with the following recursive algorithm.

```
public static ListNode reverse(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode reversedLast = list.next;
    ListNode reversedFirst = reverse(list.next);
    reversedLast.next = list;
    list.next = null;
    return reversedList;
}
```

Example 2.4. The following algorithm

```
public static ListNode rec(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode after = rec(list.next);
    if (list.info <= after.info) {
        list.next = after;
        return list;
    }
    return after;
}
```

2.2 Sorting

As a start, we can sort only ordered things, so we will be talking about arrays and lists (both ArrayLists and LinkedLists). The `Java.util` implementations sort from least to greatest, sorts in place (i.e. mutates the array rather than creating a new one), and is stable (i.e. does not reorder elements if not needed).

1. `Arrays.sort(int[] x)` is used to sort arrays and is $O(n \log n)$
2. `Collections.sort(List<> x)` is used to sort lists and is $O(n \log n)$.

By definition, sorting requires some sort of order $\leq$ defined on a set, and this order can be implemented through a comparable interface, which has a `a.compareTo(b)` method, which returns a negative integer for $\leq$, 0 for $=$, and a positive integer for $\geq$.

Example 2.5. Since strings implement this interface, we can compare them lexicographically, which is the **natural order** for String objects.

1. `"a".compareTo("b")` returns -1
2. `"b".compareTo("b")` returns 0
3. `"b".compareTo("a")` returns 1

Example 2.6. We can create a custom class of Blobs and compare them by their names.

```
public class Blob implements Comparable<Blob> {
    String name;
    String color;
    int size;

    @Override
    public int compareTo(Blob other) {
        return this.name.compareTo(other.name);
    }
}
```

Therefore, after putting them all into an array, we can call the `Arrays.sort` method to sort them with our custom `compareTo` operator.

```
List<Blob> myBlobs = new ArrayList<>();
myBlobs.add(new Blob("bo", "blue" 4);
myBlobs.add(new Blob("al", "red", 2);
myBlobs.add(new Blob("cj", "green", 1);
myBlobs.add(new Blob("di", "red", 4);

System.out.println(myBlobs);
// [("bo", "blue" 4), ("al", "red", 2), ("cj", "green", 1), ("di", "red", 4)]

Collections.sort(myBlobs);
System.out.println(myBlobs);
// [("al", "red", 2), ("bo", "blue" 4), ("cj", "green", 1), ("di", "red", 4)]
```

If we want to make multiple custom sorting systems that is not the natural order, we have to define a separate helper class implementing a `comparator` interface.

```
public class Blob implements Comparable<Blob> {
    String name;
    String color;
    int size;

    @Override
    public int compareTo(Blob other) {
        return this.name.compareTo(other.name);
    }

    public class BlobComparator implements Comparator<Blob> {
        @Override
        public int compare(Blob a, Blob b) {
            int sizeDiff = a.size - b.size;
            if (sizeDiff != 0) {
                return (-1) * sizeDiff;
            }
            return a.compareTo(b);
        }
    }

    public static void main(String[] args) {
        \\ assume myBlobs already defined

        System.out.println(myBlobs);
        // [("bo", "blue" 4), ("al", "red", 2), ("cj", "green", 1), ("di", "red", 4)]

        Collections.sort(myBlobs, new BlobComparator()); // custom sorting
        System.out.println(myBlobs);
        // [("bo", "blue" 4), ("di", "red", 4), ("al", "red", 2), ("cj", "green", 1)]
    }
}
```

In summary, comparables allow you to define an natural ordering, while comparators allow you to define other custom orderings. Furthermore, when comparing two `a` and `b`, comparables are methods on the specific object `a.compareTo(b)`, while comparators are methods on the `Comparator` object `c.compare(a, b)`.

Theorem 2.2 (Comparator Shorthands)**.** *Here are some comparator shorthands:*

1. *To create a comparator that compares according to the natural order, we just do*

```
Comparator<String> c = Comparator.naturalOrder();
c.compare("a", "b") \\ -1
c.reversed().compare("a", "b"); \\ 1
```

2. *To create a comparator that compares according to the length, we just do*

```
Comparator<String> c = Comparator.comparing(String::length);
c.compare("this", "is") \\ 1
c.compare("is", "it") \\ 0
```

3. *If we want to compare according to the length and then the natural order, then we just do*

```
Arrays.sort(arr, Comparator.
    comparing(String::length).
        thenComparing(Comparator.naturalOrder()));
```

Definition 2.1 (Selection Sort)**.** The **selection sort** algorithm is an iterative algorithm with the loop invariant that "on iteration i, the first i elements are the smallest i elements in sorted order." On iteration i, we must find the smallest element from index i onward and swap that with the element at index i.

```
public static void selectSort(int[] ar) {
    for (int i = 0; i < ar.length; i++) {
        int minDex = i;
        for (int j=i+1; j < ar.length; j++) {
            if (ar[j] < ar[minDex]) {
                minDex = j;
            }
        }
        int temp = ar[i];
        ar[i] = ar[minDex];
        ar[minDex] = temp;
    }
}
```

This is $O(n^2)$.

Definition 2.2 (MergeSort Algorithm)**.** An improved version of this algorithm uses a recursive method, which does the steps

1. Take an array split it into two halves.
2. Sort the first half and then sort the second half.
3. Merge the two halves so that the combined total is sorted.

and has the base case that sorting an array of length 1 is just itself. To do this, we must describe the helper function `merge`, which will merge two sorted arrays into a bigger sorted array. We basically want to loop through each indices of each array and add the smaller element to the new bigger array until we've exhausted all elements in one of the arrays. Then, we just copy the rest of the elements in the other array over.

Furthermore, we can create a wrapper function `mergeSort`, which helps us initialize the parameters to the recursive call, allowing for more convenience.

```
public static void mergeSort(int[] ar) {
    mergeHelper(ar, 0, ar.length);
}

public static void merge(int[] ar, int l, int mid, int r) {
    int[] sorted = new int[r - l];
    int sDex = 0; int lDex = l; int rDex = mid;
    while (lDex < mid && rDex < r) {
        if (ar[lDex] <= ar[rDex]) {
            sorted[sDex] = ar[lDex];
            lDex++;
        }
        else {
            sorted[sDex] = ar[rDex];
            rDex++;
        }
        sDex++;
    }
    if(lDex == mid) {System.arraycopy(ar, rDex, sorted, sDex, r - rDDex); }
    else {System.arraycopy(ar, lDex, sorted, sDex, mid - lDex); }
    System.arraycopy(sorted, 0, ar, l, r - l);
}

public static void mergeHelper(int[] ar, int l, int r) {
    int diff = r - l;
    if (diff < 2) {return;}     // base case if 0 or 1 elements
    int mid = l + diff/2
    mergeHelper(ar, l, mid);    // recursively sort 1st half
    mergeHelper(ar, mid, r);    // recursively sort 2nd half
    merge(ar, l, mid, r);       // merge the 2 sorted parts
}
```

There will be $O(\log n)$ levels of recursion, and for each recursion we will have to run the `merge` function, which is linear in the number of elements we are sorting ($O(n)$), so the total time complexity of this algorithm is $O(n \log n)$. We can also determine the **recurrence relation** of this algorithm as

$$T(N) = T(N/2) + T(N/2) + O(N) = O(N \log N)$$

2.3 Searching

Definition 2.3 (Binary Search)**.** Given that we have a sorted list (this is important!), we can search for the index of an element in $O(\log n)$ time. We want the loop invariant "if the target is in the array/list, it is in the range [low, high]." Let us have a list of N elements, and at every step, we either

1. get our desired element and its index, or
2. cut down our search space by half

The code can be a bit more general by implementing a generic type `T`.

```
public static <T> int binarySearch(List<T> list, T target, Comparator<T> comp) {
    int low = 0;
    int high = list.size() - 1;
    while (low <= high) {
        int mid = (low + high)/2;   // rounds down since integer division
        T midval = list.get(mid);
```

```
        int cmp = comp.compare(midval, target);
        if (cmp < 0)
            low = mid + 1;
        else if (cmp > 0)
            high = mid - 1;
        else
            return mid;         // target found
    }
    return -1;      // target not found
}
```