CSE 12 Week 5 Discussion

4-30-21

Focus: PA5, Sorting, Partition, Using Code on Internet

Reminders

- PA5 is a **closed** assignment - no collaborating!

- PA2 Resubmission due TODAY (Friday, April 30th) at 11:59 PM
- PA3 Resubmission due next week (Friday, May 7th) at 11:59 PM

PA5 Overview

1. You will be implementing 3 methods in PartitionOracle.java
    - `findCounterExample`
    - `generateInput`
    - `isValidPartitionResult`
2. Writing your own tests in TestPartitionOracle.java
3. Writing 2 different Partitioners
    - FirstElePivotPartitioner.java
    - CentralPivotPartitioner.java
4. Copying an implementation of partition online (when allowed) and adapt it to a Partitioner in WebPartitioner.java

Tips: The autograder on Gradescope is very slow, so test locally. Do not use Gradescope as a debugger!

Best case vs. Worst case Runtime

Common Misconception: The best case time complexity for this sorting algorithm is O(1) because in the best case the array only has one element

Why this is not right: When we say “best case” or “worst case”, it is with respect to n, which is the size of the data. When an array only has one element, n = 1, so if this algorithm has a time complexity of O(n^2) on any array, its time complexity on an array of 1 will still be O(n^2). Hence, O(1) describes an algorithm whose runtime is independent of the size of the input data.

Best case vs. Worst case Runtime

In the context of sorting:

Best case and worst case happen when the order of values in the array happen to make this algorithm run the fastest/slowest compared to when the input array is any other array of the same size.

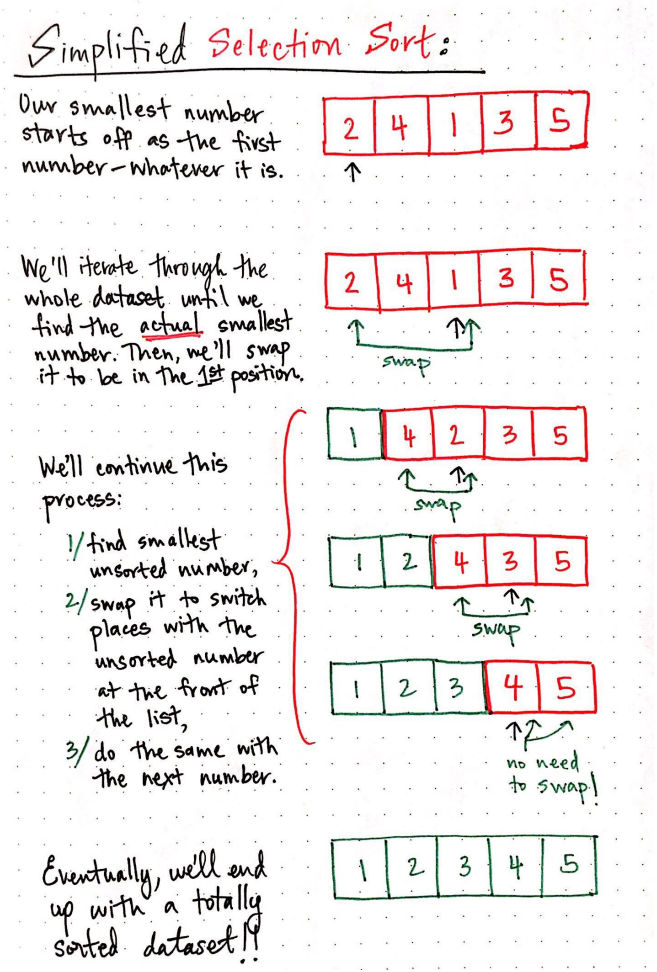
E.g. Let's say we have a group of sorted arrays and a group of unsorted arrays of the same size. If the sorting algorithm generally runs faster on the group of sorted arrays, we can say that the best case for this algorithm is when the input array is sorted

Average Case Runtime

- The performance of the algorithm on an “average” array of size n
  - Imagine we have a set of all possible input data, and we randomly select some out of this set and measure the average runtime of the algorithm on those selected data
- Sometimes you will see this term alongside “best case” and “worst case”
- Hard to prove but sometimes make sense intuitively

Sorting

Selection sort



Selection sort

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length - 1; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the worst case runtime for selection sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. o(n!)
e. None of the above

Answer - A

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length - 1; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

$n + (n - 1) + \ldots + 2 = (n + 2) * (n - 1) / 2 = O(n^2)$

Selection sort

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length - 1; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the best case runtime for selection sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

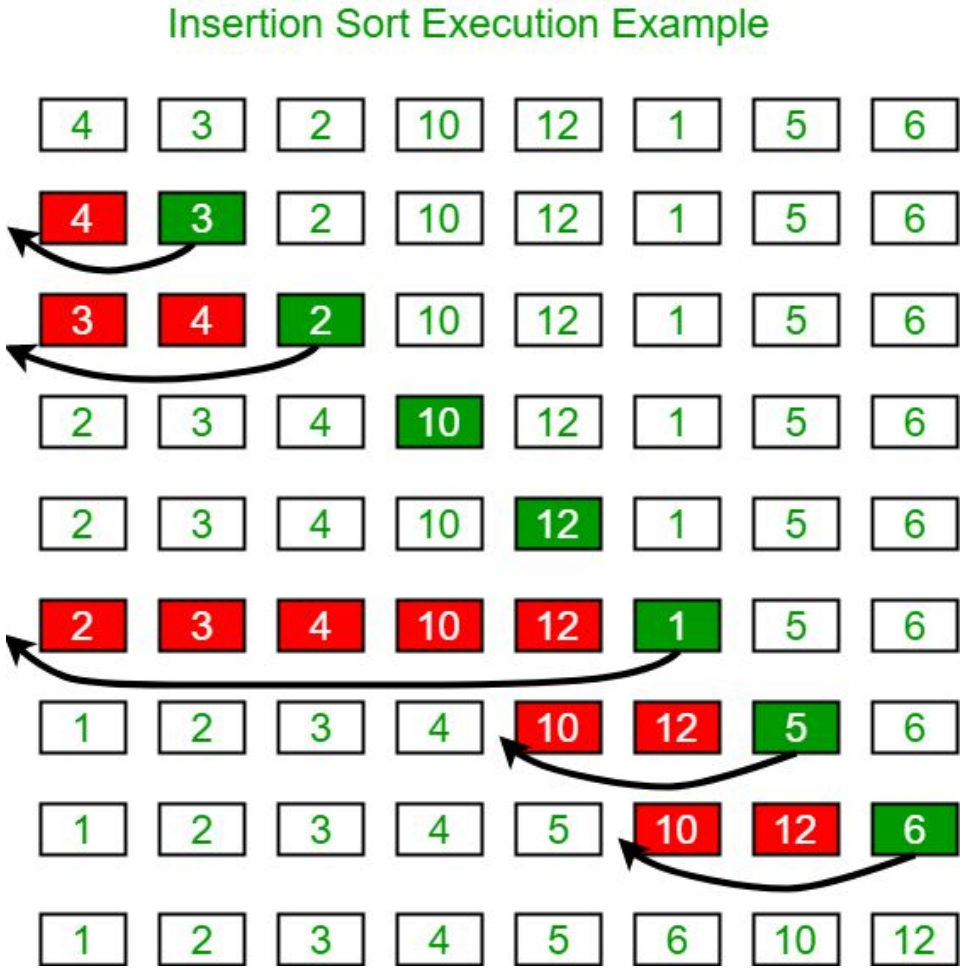
Answer - A

```
public static void sSort(int[] arr) {
  for(int i = 0; i < arr.length - 1; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
    System.out.println(Arrays.toString(arr));
  }
}
```

Even in best case, outer loop runs through all iterations while going through entire inner loop each time

So it's still O(n^2)

Insertion sort



Insertion sort

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the worst case runtime of insertion sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - A

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

Will not call break statement in worst case because entire list is out of order

$1 + 2 + \ldots + (n - 1) = (1 + (n - 1)) * (n - 1) / 2 = O(n^2)$

Insertion sort

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

What is the best case runtime of insertion sort?

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

Answer - B

```
public static void iSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    System.out.print(Arrays.toString(arr) + " -> ");
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else {
        break;
      }
    }
    System.out.println(Arrays.toString(arr));
  }
}
```

Best case: sorted list

As a result, will call break statement each time inner loop is entered

Inner loop runs O(1)

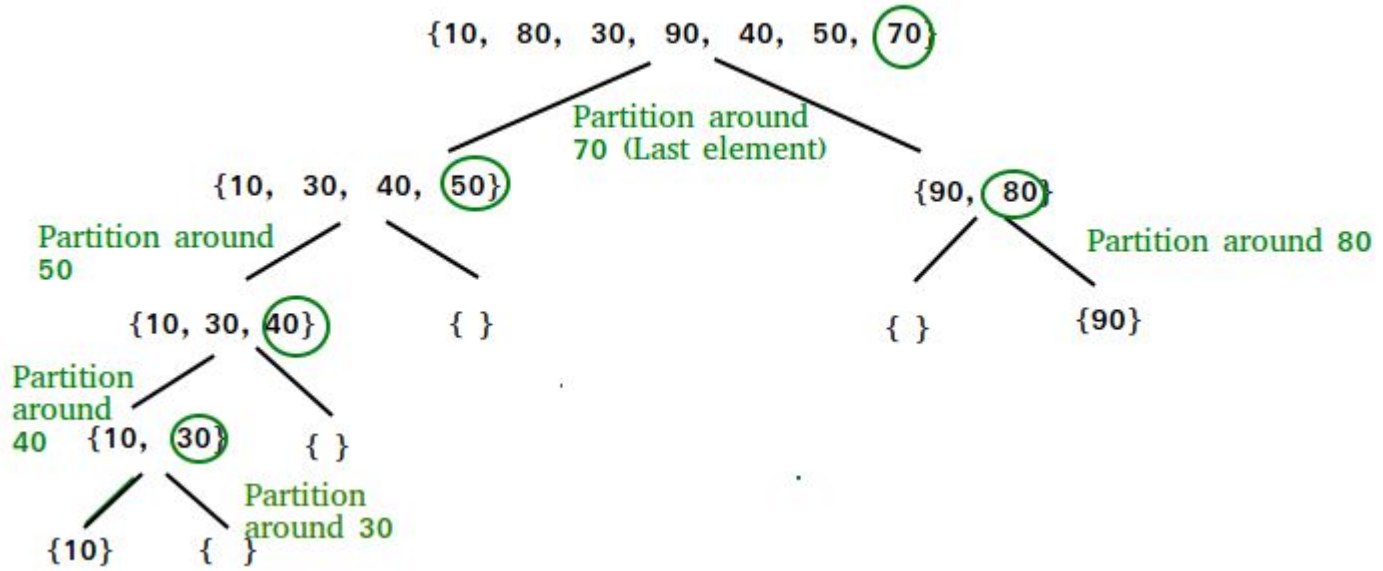
Outer loop still runs O(n)

O(n) * O(1) = O(n)

Quicksort Summary

1. Pick a pivot
    a. This can be the first element, last element, middle element etc. (Needs to be consistent though)
2. Partition around the pivot
3. Repeat with the subarrays determined by the pivot

Graphic visualization of quicksort



Resultant single-element arrays to combine: `{{10}, {30}, {40}, {50}, {70}, {80}, {90}}`

Partition

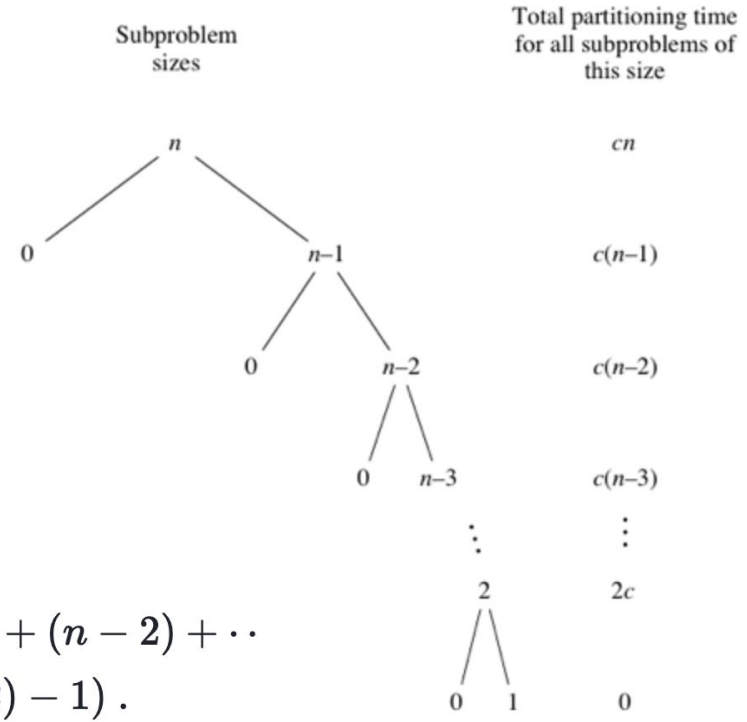
```
public int partition(String[] array, int low, int high) {
     if(low == high) { return low; }
     int pivotIndex = high - 1;
     String pivot = array[pivotIndex];
     int smallerBeforeIndex = low;
     int largerAfterIndex = high - 2;
     while(largerAfterIndex >= smallerBeforeIndex) {
          if(Integer.parseInt(array[smallerBeforeIndex]) >
                    Integer.parseInt(pivot)) {
               swap(array, smallerBeforeIndex, largerAfterIndex);
               largerAfterIndex -= 1;
          } else {
               smallerBeforeIndex += 1;
          }
     }

     if(Integer.parseInt(array[smallerBeforeIndex]) < Integer.parseInt(pivot)){
          swap(array, smallerBeforeIndex + 1, pivotIndex);
          return smallerBeforeIndex + 1;
     } else{
          swap(array, smallerBeforeIndex, pivotIndex);
          return smallerBeforeIndex;
     }
}
```

How can we test **any** partition?

Direct unit testing is not always possible. For example, if we step through our *own* partition algorithm, we know what exact array it should output, and we can use `assertArrayEquals` to compare the returned array and what we expect. But, different partitioners could return different arrays that are still valid partitions! In this case, we want to test for a valid partition by testing whether or not the array produced matches with our general constraints: all elements from before are still present, the array length hasn't changed, all elements to the left of the pivot index are less than or equal to the pivot value, and all elements to the right of the pivot index are greater than or equal to the pivot value

Quicksort Worst Case

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

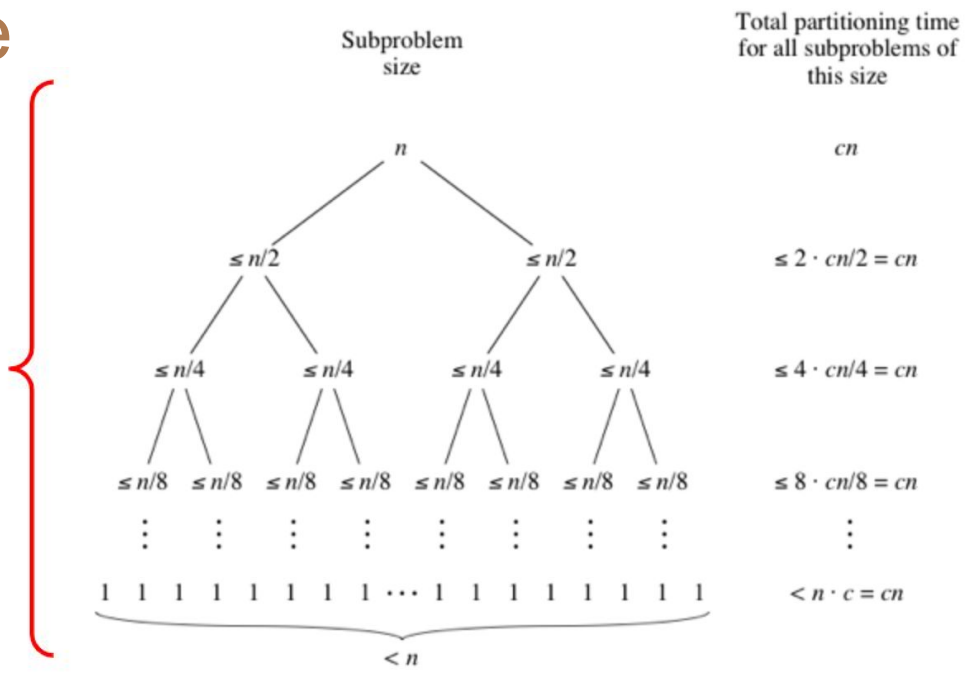


$$cn + c(n-1) + c(n-2) + \cdots + 2c = c(n + (n-1) + (n-2) + \cdots$$
$$= c((n+1)(n/2) - 1)\ .$$

Quicksort Best Case

a. O(n^2)
b. O(n)
c. O(nlogn)
d. O(n!)
e. None of the above

logn levels

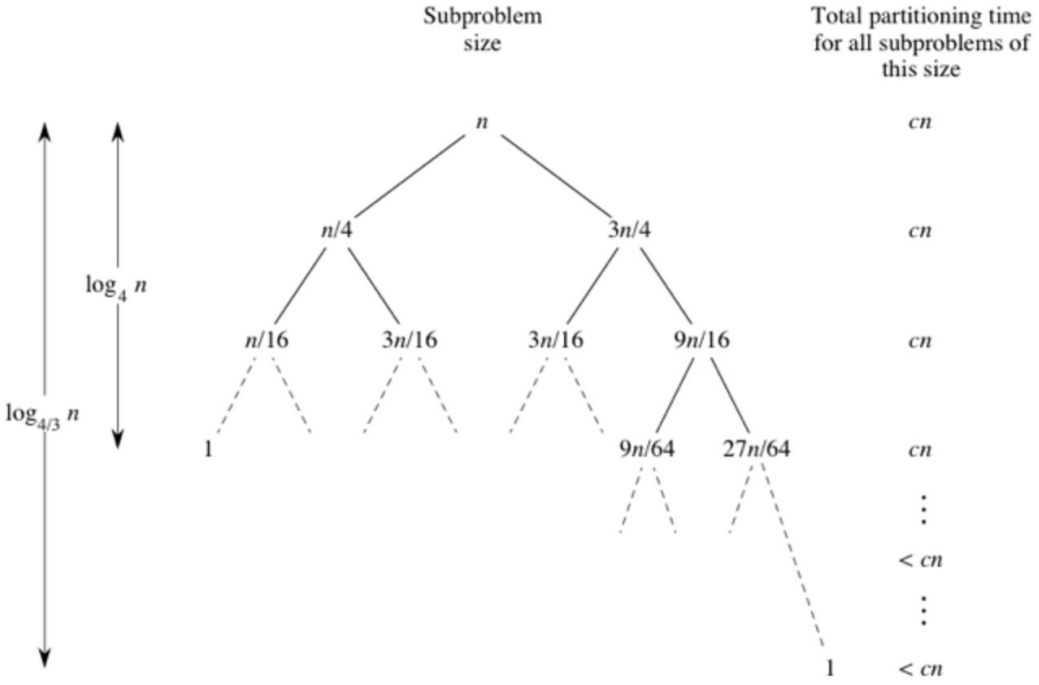


Quicksort Average Case

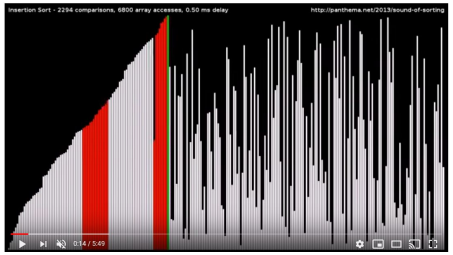
Hard to prove but we can still make sense of it from an example.

Runtime: O(nlogn)

Note: in a coding interview, if you used a built-in sorting method for some programming language, we assume that its tight big-O bound is O(nlogn)



Sorting Algorithm Visualizations



WARNING: flashing lights

https://www.youtube.com/watch?v=kPRA0W1kECg

If you do not have a hearing sensitivity, we recommend having the volume on during these visualizations. Lower pitches correspond to operations on smaller bars/values, and higher pitches correspond to operations on larger bars/values. Check out the top left corner for the current sort method and number of certain operations.

The first 4 sort methods are the 4 you've learned so far: insertion, selection, quick, and merge sort.

Note on bogo sort: this is a bit of a joke in CS. Bogo sort works by randomly moving all of the values until it finds the sorted solution by chance; it is the last sort method in the video. It has run time upper bound: O((n+1)!)

Using Code from Internet

Licenses

main [redacted] / LICENSE

Go to file ···

[redacted] is licensed under the **MIT License**

A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

Permissions	Limitations	Conditions
✓ Commercial use	✕ Liability	ⓘ License and copyright notice
✓ Modification	✕ Warranty	
✓ Distribution		
✓ Private use		

This is not legal advice. Learn more about repository licenses.

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

This is a human-readable summary of (and not a substitute for) the license. Disclaimer.

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Where to look

- GitHub: Some repositories have a file named LICENSE or LICENSE.txt in them that specifies the allowed usage of their code. **If a repo doesn't contain a LICENSE file, don't use it!**
- Stack Overflow: See https://stackoverflow.com/help/licensing for license of publicly available user answers
- Any other website that has a license or a terms of service that specifies the legal usage of their code

Link to Google Slide

https://docs.google.com/presentation/d/1h5ou2JO-jVzELldhID0rV1Es62lJLerflu-allpNYCo/edit?usp=sharing