Sorting and Searching

- In this lecture we will cover:
  - Linear Search
  - Binary Search
  - Bubble Sort
  - Selection Sort

Note: other sorts are covered in your reading. We don't test on them, but very worth reviewing for CS 165.

Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

CS 165 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:
  - Computer science has a number of concentrations.
    - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
    - Software Engineering
    - Computing Systems
    - Human Centered Computing
    - Networks and Security
    - Artificial Intelligence
    - Computer Science Education.
  - Minors:
    - Minor in Computer Science - choose your own adventure minor
    - Minor in Machine Learning - popular with stats/math, and engineering
    - Minor in Bioinformatics - Biology + Computer Science

Linear Search

You have already been doing this!

Recall back to finding a character in a String, and the 'find' method you wrote.

Let's modify that to numbers in an array.

In [2]:
```
int[] vals = {10, 11, 30, 25, 12, 13};

public static int linearSearch(int[] arr, int key) {
    for(int i = 0; i < arr.length; i++) {
        if(arr[i] == key) return i;
    }
    return -1; // not found
}

System.out.println(linearSearch(vals, 30));
System.out.println(linearSearch(vals, 13));
System.out.println(linearSearch(vals, 100));
```

```
2
5
-1
```

Let's also do this with objects!

The following code **is wrong** even if our tests are coming out correct. Table discussion, how do I correct it?

In [2]:
```
String[] values = {"aang", "zuko", "toph", "katara", "iroh", "sokka"};
Integer[] values2 = {10, 11, 30, 25, 12, 13};

public static int linearSearch(Object[] values, Object key) {
    for(int i = 0; i < values.length; i++) {
        if(key == values[i]) return i;
    }
    return -1;
}

System.out.println(linearSearch(values, "aang"));
System.out.println(linearSearch(values2, 13));
```

```
0
5
```

In [6]:
```
Scanner in = new Scanner(System.in);
System.out.print("Enter a name to search for: ");
String name = in.nextLine(); // typing in aang
System.out.println(linearSearch(values, name));
```

```
Enter a name to search for: -1
```

Thinking Further / Advanced Concept

- Not directly related to linear search, but wanted to mention / remind you.
- `==` compares memory addresses, but it gets complicated with Strings and Boxing and Unboxing
  - With Strings - java will sometimes use the same String
    - immutable, saves memory if the contents are the same

    - With numbers, boxing and unboxing will get you
- However, that doesn't work when reading in from files or System.in
- This is why it is **critical** to use `.equals` ! The above code should be corrected as follows:

In [7]:
```
public static int linearSearch(Object[] values, Object key) {
    for(int i = 0; i < values.length; i++) {
        if(key.equals(values[i])) return i;
    }
    return -1;
}

System.out.println(linearSearch(values, "aang"));
System.out.println(linearSearch(values2, 13));


Scanner in = new Scanner(System.in);
System.out.print("Enter a name to search for: ");
String name = in.nextLine(); // typing in anng
System.out.println(linearSearch(values, name));
```

```
0
5
Enter a name to search for: 0
```

Linear Search Cost - Big-O

- Big O notation is something you cover in CS165 and CS 220
- Helps determine the 'worst/average/best' case scenario for an algorithm.
- Thinking about a linear search
    - What is the 'best' case on finding an item (least number of operations)?
    - What is the 'worst' case for finding an item?

Worst Case

- The worst case is what makes the Big-O
- Given any size of your array, you have N elements
- As such the Big O is $O(n)$ for a linear search.
- Can we do better?

Binary Search

- If the elements are sorted
    - [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    - We can quickly search by 'dividing' the array into two parts each time
    - We ask ourselves, is the key 'larger' or smaller each time, until we find it!

In [8]:
```
// quick helper method so we can see what is going on
public static void rangePrint(int[] arr, int start, int end) {
    StringBuilder result = new StringBuilder();
    result.append("[");
    for(int i = start; i <= end; i++) {
```

```
        result.append(arr[i]).append(", ");
    }
    result.replace(result.length() - 2, result.length(), "]"); // replaces the last ,
    System.out.println(result);
}
```

In [12]:

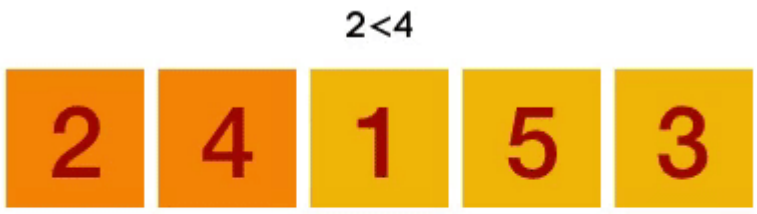
```
public static int binarySearch(int[] arr, int key) {
   return binarySearch(arr, key, 0, arr.length-1);
}
public static int binarySearch(int[] arr, int key, int first, int last){
   int mid = (first + last) / 2;
   if(first > last) return -1; // not found
   rangePrint(arr, first, last);

   if (arr[mid] == key ) return mid; // index found!
   if (arr[mid] < key ){
      first = mid + 1;
      return binarySearch(arr, key, first, last);
   }
   last = mid -1;
   return binarySearch(arr, key, first, last);
 }


int arr[] = {10,20,30,40,50};
int key = 40;
int loc = binarySearch(arr,key);
if(loc > 0) System.out.printf("Value found at %d, and the value is %d%n", loc, arr[loc
```

```
[10, 20, 30, 40, 50]
[40, 50]
Value found at 3, and the value is 40
```

- Much quicker!
- The Big O? is $O(\log n)$
- But what if your data isn't sorted?
  - There is an additional cost to that!
  - Let's explore two sorts

Bubble Sort

- Also called sorting by exchange
  - Kenneth Inverson
- "Bubbles" up the largest numbers
  - Takes the first number
    - if the next is less, swap it so the larger moves up
    - if the next is more, shift to that number, and continue
- Pass 1
  - the largest number ends up at the end
- Pass 2
  - the second largest number ends up at the end
- and so on



reference

In [19]:
```
public static void bubbleSort(int a[]) {
    for(int i=0; i<a.length; i++) {
        System.out.printf("Step %d: %s%n", i, Arrays.toString(a));
        for(int j=0; j<a.length-i-1; j++) {
        if(a[j] > a[j+1]) {
          int temp = a[j];
          a[j] = a[j+1];
          a[j+1] = temp;
        }
      }
    }
    System.out.println("The final array is " + Arrays.toString(a));
  }

int[] arr = {12, 15, 6, 8, 19, 5};
bubbleSort(arr);
```

```
Step 0: [12, 15, 6, 8, 19, 5]
Step 1: [12, 6, 8, 15, 5, 19]
Step 2: [6, 8, 12, 5, 15, 19]
Step 3: [6, 8, 5, 12, 15, 19]
Step 4: [6, 5, 8, 12, 15, 19]
Step 5: [5, 6, 8, 12, 15, 19]
The final array is [5, 6, 8, 12, 15, 19]
```

In [20]:
```
// worst case scenario?
int[] worst = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
bubbleSort(worst);
```

```
Step 0: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Step 1: [9, 8, 7, 6, 5, 4, 3, 2, 1, 10]
Step 2: [8, 7, 6, 5, 4, 3, 2, 1, 9, 10]
Step 3: [7, 6, 5, 4, 3, 2, 1, 8, 9, 10]
Step 4: [6, 5, 4, 3, 2, 1, 7, 8, 9, 10]
Step 5: [5, 4, 3, 2, 1, 6, 7, 8, 9, 10]
Step 6: [4, 3, 2, 1, 5, 6, 7, 8, 9, 10]
Step 7: [3, 2, 1, 4, 5, 6, 7, 8, 9, 10]
Step 8: [2, 1, 3, 4, 5, 6, 7, 8, 9, 10]
Step 9: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The final array is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Worst Case - a reversed list
- Every element has to move
  - every element looks at every element

- Which creates $n * n$ repetitions
- $O(n^2)$
- Furthermore, there just as many *swap* operations! (notice the if + swap is inside the inner loop)
- Bubble Sort is one of the easiest sorts to write
    - It is also one of ones we should never use

Student Practice:

Given the following array:

[3, 2, 5, 1, 8]

Write what the array looks like for each step!

In [21]:
```
bubbleSort(new int[]{3, 2, 5, 1, 8});
```

```
Step 0: [3, 2, 5, 1, 8]
Step 1: [2, 3, 1, 5, 8]
Step 2: [2, 1, 3, 5, 8]
Step 3: [1, 2, 3, 5, 8]
Step 4: [1, 2, 3, 5, 8]
The final array is [1, 2, 3, 5, 8]
```

Selection Sort

- Searches the array for the **lowest** value
- Moves that to the start index
    - repeats, incrementing index
- Also was called Jump Down sort
    - Elliott Organick

In [25]:
```
public static void selectionSort(int a[]) {
    for (int i = 0; i < a.length-1; i++){
        System.out.printf("Step %d: %s%n", i, Arrays.toString(a));
        int min = i; // assume current is the lowest
        for (int j = i+1; j < a.length; j++) {
            if (a[j] < a[min]) min = j;
        }
        // swap
        int temp = a[min];
        a[min] = a[i];
        a[i] = temp;
    }
    System.out.println("The final array is " + Arrays.toString(a));
}
```

In [29]:
```
int[] arr2 = {12, 15, 6, 8, 19, 5};
selectionSort(arr2);
System.out.println();
System.out.println();
```

```
int[] worst2 = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
selectionSort(worst2);
```

```
Step 0: [12, 15, 6, 8, 19, 5]
Step 1: [5, 15, 6, 8, 19, 12]
Step 2: [5, 6, 15, 8, 19, 12]
Step 3: [5, 6, 8, 15, 19, 12]
Step 4: [5, 6, 8, 12, 19, 15]
The final array is [5, 6, 8, 12, 15, 19]


Step 0: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Step 1: [1, 9, 8, 7, 6, 5, 4, 3, 2, 10]
Step 2: [1, 2, 8, 7, 6, 5, 4, 3, 9, 10]
Step 3: [1, 2, 3, 7, 6, 5, 4, 8, 9, 10]
Step 4: [1, 2, 3, 4, 6, 5, 7, 8, 9, 10]
Step 5: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Step 6: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Step 7: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Step 8: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The final array is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Worse case still has n*n combinations
  - note: the fact it is one less step factors out
- $O(n^2)$
- However, it has far less *swap* operations
  - Notice the swap is outside the inner loop
- As such, selection is a better alternative than bubble sort
  - Still not the best!

Sorting Objects

- This is all great, but what about sorting objects?
- We can't use < or > for objects
  - Maybe primitives inside the object, but that gets complicated.
- In java:
  - Comparable\<T> interface
  - has a single method:
    - public int compareTo(\<T> obj)
    - where T is the type you specify (similar to ArrayList\<Type>)
      - returns:
        - 1 if the object is greater than obj
        - 0 if they are equal
        - -1 if the object is less than obj
  - Also recommended to overwrite .equals(Obj) if you implement compareTo
    - Has to do with how certain classes interact

In [30]:
```
String a = "ada";
String b = "beatrice";
```

```
System.out.println(a.compareTo(b));
```

```
-1
```

In [33]:
```
public class FullName implements Comparable<FullName> {
    private String first;
    private String last;

    public FullName(String first, String last) {
        this.first = first;
        this.last = last;
    }

    public String toString() {
        return String.format("%s, %s", last, first);
    }

    public boolean equals(Object obj) { // discuss why does this have to be Object!
        if(obj == this) return true; // same object
        if(! (obj instanceof FullName)) return false;
        FullName other = (FullName)obj;
        return first.equals(other.first) && last.equals(other.last);
    }

    public int compareTo(FullName other) {
        if(last.compareTo(other.last) != 0) return last.compareTo(other.last); // sort
        if(first.compareTo(other.first) != 0) return first.compareTo(other.first); //
        return 0; // they are exactly equal
    }


}
```

In [52]:
```
// shout out, these are the ENIAC programmers from World War II!
FullName[] eniac = { new FullName("Kay", "McNulty"),
                     new FullName("Betty", "Holberton"),
                     new FullName("Jean", "Jennings"),
                     new FullName("Maryln", "Wescoff"),
                     new FullName("Ruth", "Lichterman"),
                     new FullName("Frances", "Bilas")};

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

```
[McNulty, Kay, Holberton, Betty, Jennings, Jean, Wescoff, Maryln, Lichterman, Ruth, B
ilas, Frances]
```

In [54]:
```
public static void selectionSort(Comparable a[]) { // notice type!
    for (int i = 0; i < a.length-1; i++){
        System.out.printf("Step %d: %s%n", i, Arrays.toString(a));
        int min = i; // assume current is the lowest
        for (int j = i+1; j < a.length; j++) {
            if (a[j].compareTo(a[min]) < 0) min = j;
        }
        // swap
        Comparable temp = a[min];
        a[min] = a[i];
        a[i] = temp;
    }
```

```
    System.out.println("The final array is " + Arrays.toString(a));
}
```

Notice, we changed the < to compareTo < 0

Also, notice the type - take a moment to discuss the code changes, and why. Why couldn't I just use Object?

In [56]:
```
selectionSort(Arrays.copyOf(eniac, eniac.length));
```

```
Step 0: [McNulty, Kay, Holberton, Betty, Jennings, Jean, Wescoff, Maryln, Lichterman,
Ruth, Bilas, Frances]
Step 1: [Bilas, Frances, Holberton, Betty, Jennings, Jean, Wescoff, Maryln, Lichterma
n, Ruth, McNulty, Kay]
Step 2: [Bilas, Frances, Holberton, Betty, Jennings, Jean, Wescoff, Maryln, Lichterma
n, Ruth, McNulty, Kay]
Step 3: [Bilas, Frances, Holberton, Betty, Jennings, Jean, Wescoff, Maryln, Lichterma
n, Ruth, McNulty, Kay]
Step 4: [Bilas, Frances, Holberton, Betty, Jennings, Jean, Lichterman, Ruth, Wescoff,
Maryln, McNulty, Kay]
The final array is [Bilas, Frances, Holberton, Betty, Jennings, Jean, Lichterman, Rut
h, McNulty, Kay, Wescoff, Maryln]
```

If this was in an ArrayList, I would have a built in library to sort it!

`Collections.sort()`

In [59]:
```
List<FullName> eniac_list = Arrays.asList(eniac);

Collections.sort(eniac_list); // a built in library for any 'collection' in java!
System.out.println(Arrays.toString(eniac));
```

```
[Bilas, Frances, Holberton, Betty, Jennings, Jean, Lichterman, Ruth, McNulty, Kay, We
scoff, Maryln]
```

Sorts

You are just learning sorting algorithms, there are a bunch of them (merge and quick sort being popular - both having a recursive element)

Sort	Average Case
Bubble	$O(n^2)$
Selection	$O(n^2)$
Insertion	$O(n^2)$
Quick	$O(n \log n)$
Merge	$O(n \log n)$
Linear Search	$O(n)$
Binary Search	$O(\log n)$

CS 165 and CS 220 you will cover these more!

Overview

Knowing the right algorithm matters when dealing with larger datasets, but there are also a lot of tools built into our languages that help you.

For this class, knowing the basic couple sorts will get your brain thinking about algorithmic efficiency, you will explore this more in the future.

Also, knowing `.equals` and `.compareTo` are essential for java programming as the built in Collections make major use of them!