Authorship Compilation Solutions

Allyson Park

Discussion questions written for UC Berkeley's CS61B: Data Structures + Algorithms

Contents

1 Senior Class **2**

2 Re-cursed with Asymptotics! **3**

3 ADT Matchmaking **5**

4 Disjoint Sets, a.k.a. Union Find **6**

5 A Side of Hash Browns **7**

6 A Tree Takes on Graphs **9**

7 Minimalist Moles **11**

8 The Shortest Path To Your Heart **12**

9 Sorta Interesting, Right? **15**

10 Zero One Two-Step **16**

1 Senior Class

For each line in the `main` method of our `testPeople` class, if something is printed, write it next to the line. If the line results in an error, write next it whether it is a compile time error or runtime error, and then proceed as if that line were not there.

```
1  public class Person {
2      public String name;
3      public int age;
4
5      public Person(String name, int age) {
6          this.name = name;
7          this.age = age;
8      }
9
10     public void greet(Person other) {System.out.println("Hello, " + other.name);}
11 }
12
13
14 public class Grandma extends Person {
15
16     public Grandma(String name, int age) {
17         super(name, age);
18     }
19
20     @Override
21     public void greet(Person other) {System.out.println("Hello, young whippersnapper");}
22
23     public void greet(Grandma other) {System.out.println("How was bingo, " + other.name + "?");}
24 }
25
26 public class testPeople {
27     public static void main(String[] args) {
28         Person n = new Person("Neil", 12);
29         Person a = new Grandma("Ada", 60);
30         Grandma v = new Grandma("Vidya", 80);
31         Grandma al = new Person("Alex", 70); // Compile time error
32         n.greet(a); // "Hello Ada"
33         n.greet(v); // "Hello Vidya"
34         v.greet(a); // "Hello, young whippersnapper"
35         v.greet((Grandma) a); // "How was bingo, Ada?"
36         a.greet(n); // "Hello, young whippersnapper"
37         a.greet(v); // "Hello, young whippersnapper"
38         ((Grandma) a).greet(v); // "How was bingo, Vidya?"
39         ((Grandma) n).greet(v); // Runtime error
40     }
41 }
```

2 Re-cursed with Asymptotics!

(a) What is the runtime of the code below in terms of n?

```
1  public static int[] curse(int n) {
2      if (n <= 0) {
3          return 0;
4      } else {
5          return n + curse(n - 1);
6      }
7  }
```

This takes $\Theta(n)$ time. On each recursive call, we do a constant amount of work. We make n recursive calls, because we go from n to 1. Then n recursive layers with 1 work at each layer is overall $\Theta(n)$ much work.

(b) Assume our BST (Binary Search Tree) below is perfectly bushy. What is the runtime of a single `find` operation in terms of N, the number of nodes in the tree? In this setup, assume a `Tree` has a `Key` (the value of the tree) and then pointers to two other trees, `Left` and `Right`.

```
1  public static BST find(BST T, Key sk) {
2     if (T == null)
3        return null;
4     if (sk.compareTo(T.key) == 0))
5        return T;
6     else if (sk.compareTo(T.key) < 0)
7        return find(T.left, sk);
8     else
9        return find(T.right, sk);
10 }
```

Find operations on a BST take $O(\log(N))$ time, as the height of a perfectly bushy BST is log(N). In the worst case scenario, the key we're looking at is all the way at a leaf, so we have to traverse a path from root to leaf of length log(N).

(c) Can you find a runtime bound for the code below? We can assume the System.arraycopy method takes $\Theta(N)$ time, where N is the number of elements copied. The official signature is `System.arrayCopy(Object sourceArr, int srcPos, Object dest, int destPos, int length)`. Here, `srcPos` and `destPos` are the starting points in the source and destination arrays to start copying and pasting in, respectively, and `length` is the number of elements copied.

```
1  public static void silly(int[] arr) {
2      if (arr.length <= 1) {
3          System.out.println("You won!");
4          return;
5      }
6      int newLen = arr.length / 2
7      int[] firstHalf = new int[newLen];
8      int[] secondHalf = new int[newLen];
9      System.arraycopy(arr, 0, firstHalf, 0, newLen);
10     System.arraycopy(arr, newLen, secondHalf, 0, newLen);
```

```
11      silly(firstHalf);
12      silly(secondHalf);
13  }
```

At each level, we do N work, because the call to System.arraycopy. You can see that at the top level, this is N work. At the next level, we make two calls that each operate on arrays of length N/2, but that total work sums up to N. On the level after that, in four separate recursive function frames we'll call System.arraycopy on arrays of length N/4, which again sums up to N for that whole layer of recursive calls.

Now we look for the height of our recursive tree. Each time, we halve the length of N, which means that the length of the array N on recursive level k is roughly $N * \frac{1}{2}^k$. Then we will finally reach our base case $N <= 1$ when we have $N * \frac{1}{2}^k = 1$. Doing some math, we see this can be transformed into $N = 2^k$, which means $k = log_2(N)$. In other words, the number of layers in our recursive tree is $log_2(N)$. If we have $log_2(N)$ layers with $\Theta(N)$ work on each layer, we must have $\Theta(Nlog(N))$ runtime.

3 ADT Matchmaking

Match each task to the correct Abstract Data Type for the job by drawing a line connecting matching pairs.

1. You want to keep track of all the unique users who have logged on to your system.	a) List
2. You are creating a version control system and want to associate each file name with a Blob.	b) Map
3. We are running a server and want to service clients in the order they arrive.	c) Set
4. We have a lot of books at our library and we want our website to display them in some sorted order. We have multiple copies of some books and we want each listing to be separate.	d) Queue

1)-c) You should use a set because we only want to keep track of unique users (i.e. if a user logs on twice, they shouldn't show up in our data structure twice). Additionally, our task doesn't seem to require that the structure is ordered.

2)-b) You should use a map. Maps naturally let you pair a key and value, and here we could have the file name be the key, and the blob be the value.

3)-d) We should use a queue. We can push clients to the front of the queue as they arrive, and pop them off the queue as we service them.

4)-a) We should use a list because a list is an ordered collection of items. Additionally, we need to allow for duplicate items because we have multiple copies of some books.

4 Disjoint Sets, a.k.a. Union Find

In lecture, we discussed the Disjoint Sets ADT. Some authors call this the Union Find ADT. Today, we will use union find terminology so that you have seen both.

(a) Omitted, authored by other

(b) Omitted, authored by other

(c) Omitted, authored by other

(d) What is the runtime for "connect" and "isConnected" operations using our Quick Find, Quick Union, and Weighted Quick Union ADTs? Can you explain why the Weighted Quick union has better runtimes for these operations than the regular Quick Union?

Runtime comparisons			
OPERATION	Quick Find	Quick Union	WQU
Connect	O(N)	O(N)	O(logN)
IsConnected	O(1)	O(N)	O(logN)

The Weighted Quick Union has better run times because by picking the smaller tree to be the child, we can achieve shorter overall heights in our underlying tree. This means that for any child, traversing up the tree to find it's root, or it's set representative, is limited to this shortened tree height. For both our standard Quick Union and Weighted Quick Union, the time it takes to connect two items depends on this height, as it requires checking the roots of the current items and then changing one to be the other (if they're not already connected). Then the time it takes to find the root of the current element is proportional to the time it takes to connect two items. Similarly, the time it takes to check if two items are connected relies on finding the roots of the current elements.

Not included in this chart is the WQU with path compression. While the proof for it's runtime is out of scope for this class, it achieves amortized constant runtime for both of Connect and isConnected.

5 A Side of Hash Browns

We want to map food items to their yumminess. We want to be able to find this information in constant time, so we've decided to use java's built-in `HashMap` class! Here, the key is an `String` representing the food item and the value is an `int` yumminess rating.

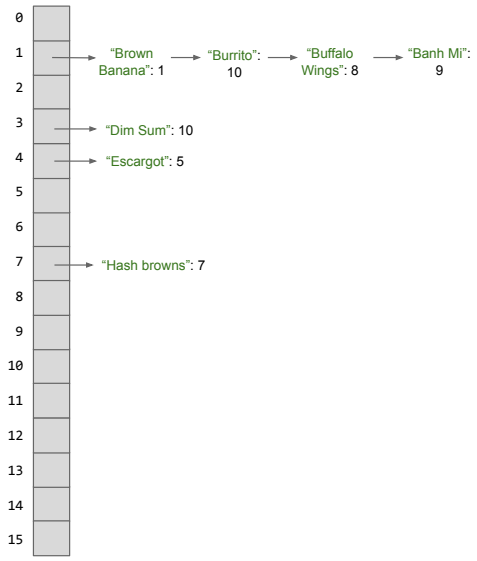
For simplicity, let's say that here a `String`'s hashcode is the first letter's position in the alphabet (A = 0, B = 1... Z = 25). For example, the `String` "hash brown" starts with "h", and "h" is 7th letter in the alphabet (0 indexed), so the hashcode would be 7. Note that in reality, a `String` has a much more complicated `hashCode()` implementation.

Our hashMap will compute the index as the key's hashcode value modulo the number of buckets in our HashMap. Assume the initial size is 4 buckets, and we double the size our HashMap as soon as the load factor reaches 3/4.

(a) Draw what our `HashMap` would look like after the following operations.

```
1 HashMap<Integer, String> hm = new HashMap<>();
2 hm.put("Hashbrowns", 7);
3 hm.put("Dim sum", 10);
4 hm.put("Escargot", 5);
5 hm.put("Brown bananas", 1);
6 hm.put("Burritos", 10);
7 hm.put("Buffalo wings", 8);
8 hm.put("Banh mi", 9);
```

Note that we resize from 4 to 8 when adding escargot (because we have 3 items and 4 buckets) and then from 8 to 16 when adding buffalo wings (because we have 6 items and 8 buckets).



(b) Do you see a potential problem here with the behavior of our `hashMap`? How could we solve this?

Here, adding a bunch of food items that start with the letter "B" result in one bucket with a lot of items. No matter how many times we resize, our current `hashCode()` will result in this problem! Imagine if we added in 100 more items that started with the letter b. Though we would resize and keep our load factor low, it wouldn't change the fact that our operations will now be slow (hint:

linear time) because now we essentially have to iterate over a linked list with pretty much all the items in the hashMap to do things like `get("Burrito")`, for example.

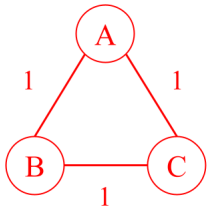
A solution to this would be to have a better `hashCode()` implementation. A better implementation would distribute the Strings more randomly and evenly. While knowing how to write such a `hashCode()` is difficult and out of scope for this class, you can look at the real `hashCode()` implementation for Java `String`s if you're curious!

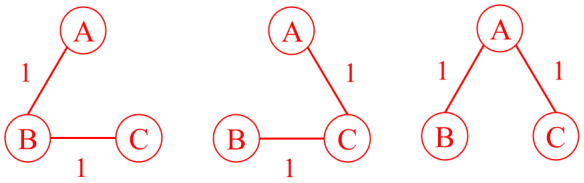
6 A Tree Takes on Graphs

Your friend at Stanford has made some statements about graphs, but you believe they are all false. Provide counterexamples to each of the statements below:

(a) "Every graph has one unique MST."

This false statement can be disproved with the below example. The graph given below has three valid MSTs, which are shown underneath it. In general, graph will only be guaranteed to have a unique MST if all of it's edge weights are unique. If a graph has duplicate edge weights, there may or may not be a unique MST.

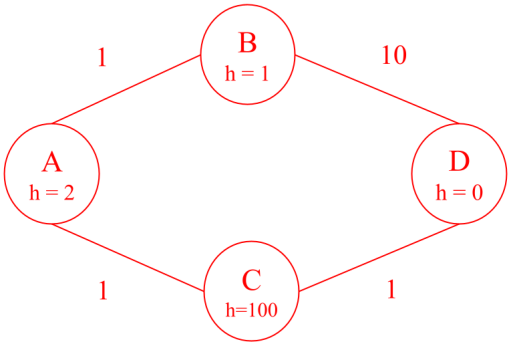




(b) "No matter what heuristic you use, A* search will always find the correct shortest path."

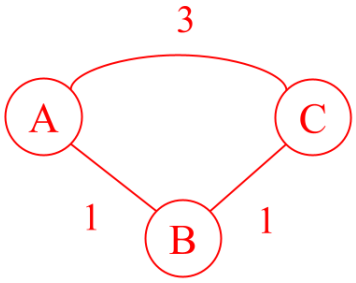
This false statement can be disproved with the below example. Here, A* would incorrectly return A - B - D as the shortest path from A to D. Starting at A, we would add B to the queue with priority 1+1 (the known distance to B, as well as our estimated distance from B to the goal), and we would add C to the queue with priority 1+100 (the known distance to C, as well as our estimated distance from C to the goal). We then pop B off the queue, and add D to the queue with priority 11+0 (the known distance to D, as well as the estimated distance from D to the goal). Our queue now contains C with priority 101, and D with priority 11, so we pop D off the queue and complete our search, returning A - B - D as the shortest path instead of the correct answer: A - C - D.

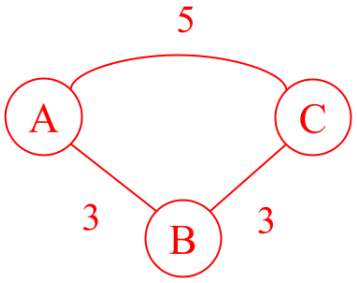
In general, A* is only guaranteed to be correct if the heuristic is good–specifically, it should be both admissible and consistent (note that applying admissibility and consistency are out of scope for this class, you only need to know the definition). In the example given, our heuristic is neither admissible nor consistent.



(c) "If you add a constant factor to each edge in a graph, Dijkstra's algorithm will return the same shortest paths tree."

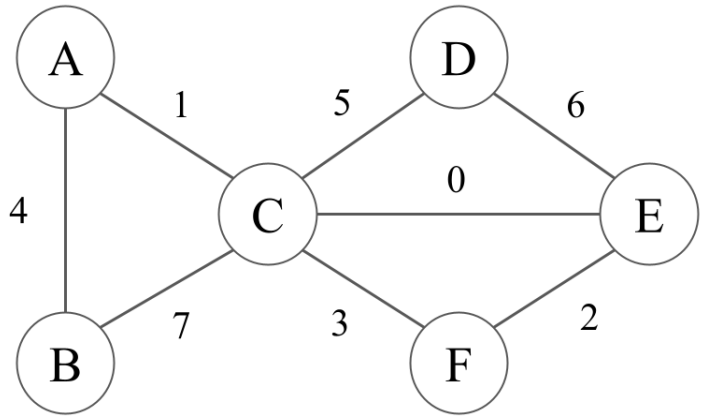
This false statement can be disproved with the example below, where we add a constant $c = 2$ to every edge. Adding a constant factor per edge will disadvantage paths with more edges. In our example, though A - B - C had more edges, in our original graph it still has shorter total path cost, at $1 + 1 = 2$. On the other hand, A - C had fewer edges but larger total path cost, 3. After adding a constant factor, however, the A - B - C path cost was $(1 + 2) + (1 + 2) = 6$ and the A - C had a path cost of $(2 + 3) = 5$. So before the addition, Dijkstra's shortest path tree would have said the shortest path from A to C was A - B - C, but afterwards it would say A - C.





7 Minimalist Moles

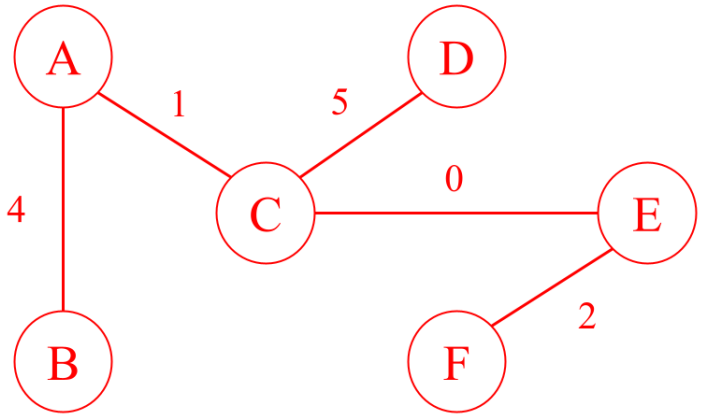
Mindy the mole wants to dig a network of tunnels connecting all of their secret hideouts. There are a set few paths between the secret hideouts that Mindy can choose to possibly include in their tunnel system, shown below. However, some portions of the ground are harder to dig than others, and Mindy wants to do as little work as possible. In the diagram below, the numbers next to the paths correspond to how hard that path is to dig for Mindy.



(a) How can Mindy figure out a tunnel system to connect their secret hideouts while doing minimal work?

This problem can be solved by finding a minimum spanning tree! This will connect all the secret hideouts (a tree will include all nodes in the graph) and it will take the minimum total work, since an MST will have the minimum total edge weight.

(b) *Extra:* Find a valid MST for the graph above using Kruskal's algorithm, then Prim's. For Prim's algorithm, take A as the start node. In both cases, if there is ever a tie, choose the edge that connects two nodes with lower alphabetical order. Both Prim's and Kruskal's give the MST below.

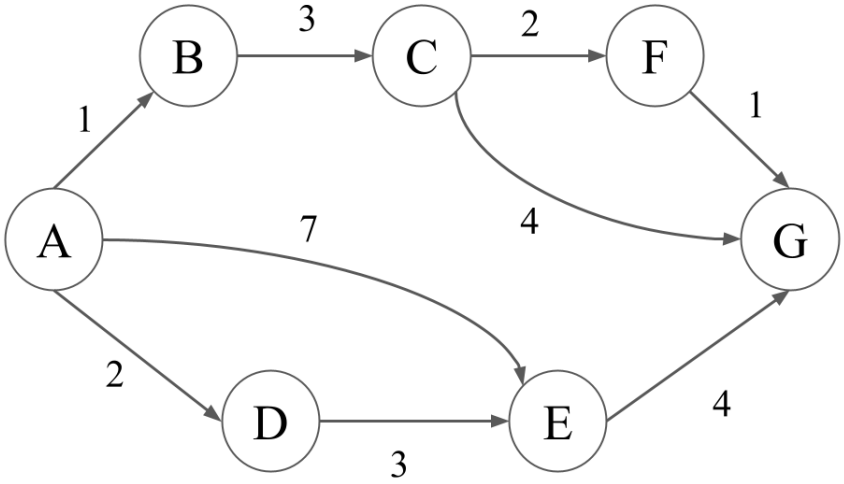


(c) *Extra:* Are the above MSTs different or the same? Is there a different tie-breaking scheme that would change your answer?

In this particular case, the trees for Prim's and Kruskal's are the same. There is NOT a different tie breaking scheme that would change this answer, because there is only one possible correct MST when all the edge weights are distinct! However, if a tree has edges with duplicate weights, then **it would be possible for Prim's and Kruskal's to give different answers**.

8 The Shortest Path To Your Heart

For the graph below, let `g(u, v)` be the weight of the edge between any nodes `u` and `v`. Let `h(u, v)` be the value returned by the heuristic for any nodes `u` and `v`.



Below, the pseudocode for Dijkstra's and A* are both shown for your reference throughout the problem.

Dijkstra's Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, 0)
PQ.add(v, infinity) # (all nodes except A).

distTo = {} # map
distTo[A] = 0
distTo[v] = infinity # (all nodes except A).

while (not PQ.isEmpty()):
  popNode, popPriority = PQ.pop()

  for child in popNode.children:
    if PQ.contains(child):
      potentialDist = distTo[popNode] +
        edgeWeight(popNode, child)
      if potentialDist < distTo[child]:
        distTo.put(child, potentialDist)
        PQ.changePriority(child, potentialDist)
```

A* Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, h(A))
PQ.add(v, infinity) # (all nodes except A).

distTo = {} # map
distTo[A] = 0
distTo[v] = infinity # (all nodes except A).

while (not PQ.isEmpty()):
  poppedNode, poppedPriority = PQ.pop()
  if (poppedNode == goal): terminate

    for child in poppedNode.children:
      if PQ.contains(child):
        potentialDist = distTo[poppedNode] +
          edgeWeight(poppedNode, child)

        if potentialDist < distTo[child]:
          distTo.put(child, potentialDist)
          PQ.changePriority(child, potentialDist + h(child))
```

(a) Run Dijkstra's algorithm to find the shortest paths from A to every other vertex. You may find it helpful to keep track of the priority queue and make a table of current distances.

`B = 1 ; C = 4 ; D = 2 ; E = 5 ; F = 6 ; G = 7`

Explanation:

For the best explanation, it is recommended to check the slideshow linked on the website or watch the walkthrough video, as the text explanation is verbose.

We will maintain a priority queue and a table of distances found so far, as suggested in the problem and pseudocode. We will use {} to represent the PQ, and (()) to represent the distTo array.

```
{A:0, B:inf, C:inf, D:inf, E:inf, F:inf, G:inf}. (()).

Pop A.
{B:inf, C:inf, D:inf, E:inf, F:inf, G:inf}. ((A: 0)).

changePriority(B, 1). changePriority(D, 2). changePriority(E, 7).
{B:1, D:2, C:inf, E:7, F:inf, G:inf}. ((A: 0)).

Pop B.
{D:2, C:inf, E:7, F:inf, G:inf}. ((A: 0, B: 1)).

changePriority(C, 4).
{D:2, C:4, E:7, F:inf, G:inf}. ((A: 0, B: 1)).

Pop D.
{C:4, E:7, F:inf, G:inf}. ((A: 0, B: 1, D: 2)).

changePriority(E, 5).
{C:4, E:5, F:inf, G:inf}. ((A: 0, B: 1, D: 2)).

Pop C.
{E:5, F:inf, G:inf}. ((A: 0, B: 1, D: 2, C: 4)).

changePriority(F, 6). changePriority(G, 8).
{E:5, F:6, G:8}. ((A: 0, B: 1, D: 2, C: 4)).

Pop E.
{F:6, G:8}. ((A: 0, B: 1, D: 2, C: 4, E: 5)).

potentialDistToG = 9, which is worse than our current best known distance to G. No updates made.

Pop F.
{G:8}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6)).

potentialDistToG = 7. changePriority(G, 7).
{G:7}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6)).

Pop G.
{}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6, G: 7)).
```

(b) *Extra:* Given the weights and heuristic values for the graph above, what path would A* search return, starting from A and with G as a goal?

A* would return $A - B - C - F - G$. The cost here is 7.

Explanation: A* runs in a very similar fashion to Dijkstra's. We got the same answer for the shortest path to G, though we actually explored less unnecessary nodes in the process (we never popped D and E off the queue). The main difference is the priority in the priority queue. For A*, whenever computing the priority (for the purposes of the priority queue) of a particular node n, always add $h(n)$ to whatever you would use with Dijkstra's. Additionally, note that A* will be run to find the shortest path to a particular goal node (as our heuristic is calculated as our estimate to our specific goal node), whereas Dijkstra's may be run with a specific goal, or it may be run to find the shortest paths to ALL nodes. In the solutions above, we found the shortest paths to all nodes, but if we only needed to know the shortest path to E, for example, we could have stopped after visiting E.

9 Sorta Interesting, Right?

(a) What does it mean to sort "in place", and why would we want this?

In general, we consider a sorting operation to be done in place if it does not require significant extra space. "Significant extra space" is often defined as linear or greater, with respect to the number of elements we are sorting. For example, if our sorting algorithm requires making a whole new array and copying elements over to it, then it is NOT in place because we had to allocate significant space for this new array. Some algorithms that can be implemented in place are selection sort, insertion sort, and heap sort. Mergesort technically can be implemented in place, but it's rather complex.

Doing operations in place is beneficial because we typically want to use as little memory/computer resources as possible. Though in this class, we focus on time efficiency, space efficiency is important too in the real world!

(b) What does it mean for a sort to be "stable"? Which sorting algorithms that we have seen are stable?

If given a list with equivalent elements (according to whatever metric we're using to compare items), a stable sorting algorithm will leave those elements in the same order as they were originally. Some stable sorts we've seen are insertion sort and merge sort.

(c) Which algorithm would run the fastest on an already sorted list?

Insertion sort would run the fastest on an already sorted list, because there would be no inversions! Then at every step, insertion sort would see that there is no inversion and simply move on to the next element. After going through the whole list like this, insertion sort would terminate, resulting in linear time!

(d) Given any list, what is the ideal pivot for quicksort?

The ideal pivot will be the median, or the element that WOULD be at the exact halfway point if the list were to be sorted. This is because it will split the list exactly in half after partitioning around it. When we are able to break our subproblem perfectly in half at every time step, it gives is the recursive tree height of $\log(n)$. With n work done at each level, we achieve $\theta(n \log n)$ runtime in this best case where we pick the median at each step. Note that the worst case pivot choice is choosing the minimum or maximum element (apply the same logic we've used above to see why!), in which we'd have $\theta(n^2)$! Then technically, the upper bound or Big O limit on quicksort is $O(n^2)$. On average though, with random pivot selection, we often see behavior that closely resembles our best case.

(e) So far, in class, we've mostly applied our sorts to lists of numbers. In practice, how would we typically make sure our sorts can be applied to other types?

In practice, if we want our items to be sortable by a comparison sort, we make sure they implement the `Comparable` interface! This involves defining a `compareTo()` method, such that a sorting algorithm has a way to compare our elements, just as naturally as it can compare numbers. This is only relevant for comparison sorts (all of the sorts in this discussion), but not counting sorts (e.g. radix sorts).

10 Zero One Two-Step

Note: this question is a popular LeetCode question, sometimes called the "Dutch National Flag" problem.

(a) Given an array that only contains 0's, 1's and 2's, write an algorithm to sort it in linear time. You may want to use the provided helper method, `swap`.

The solution below is designed in the style of comparison swaps we have seen thus far (indeed, we see it has flavor similar to quicksort with 1 as the pivot, and other sorts that use swapping). Note that there is also a completely valid counting sort approach, but counting sorts are not the focus of this discussion.

```
public static int[] specialSort(int[] arr) {
    int front = 0;
    int back = arr.length - 1;
    int curr = 0;

    while (curr <= back) {
        if (arr[curr] < 1) {
            swap(arr, curr, front);
            front += 1;
            curr += 1;
        } else if (arr[curr] > 1) {
            swap(arr,  curr, back);
            back -= 1;
        } else {
            curr += 1;
        }
    }
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

(b) We just wrote a linear time sort, how cool! Can you explain in a sentence or two why we can't always use this sort, even though it has better runtime than Mergesort or Quicksort?

While our algorithm is super cool, we were only able to write it because we knew there were exactly 3 possible values that could be in our array! For the general case, when the collections we're sorting have much more variety, we can't make these kinds of guarantees.