UC Berkeley – Computer Science M25112-A
CS61B: Data Structures

Final, Spring 2015 - SOLUTIONS, BETA. PLEASE POST TO PIAZZA IF YOU SPOT ANY BUGS (of which there is almost certainly at least one)

This test has 14 questions worth a total of 60 points. The exam is closed book, except that you are allowed to use three (front-and-back) handwritten pages of notes. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement out below in the blank provided and sign. You may do this before the exam begins.**

"I have neither given nor received any assistance in the taking of this exam."

_____
_____

	Points		Points
0	1	8	4
1	6	9	4.5
2	3	10	7
3	4.5	11	0
4	7.5	12	8
5	2	13	5
6	2.5		
7	5		

Signature: _____

Your Name: _____Jug_____

Your Student ID: _____ket_____
Three-letter Login ID: ___chu___
Login of Person to Left: ___pfr____
Login of Person to Right: __ien___
Exam Room: _____d_____

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful.
- Enjoy a long glorious summer.
- Please only submit regrade requests if you're pretty sure you're correct.

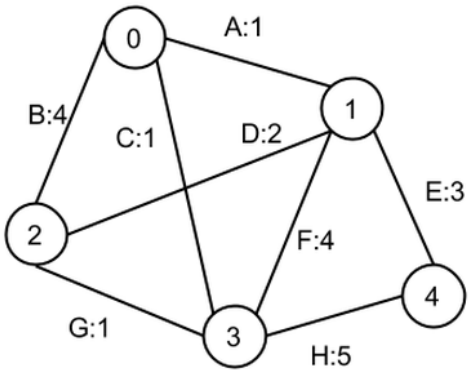
Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹_____☺].
After exam: [☹_____☺].

0. Another point. (1 points). Write your name, login, and ID on the front page. Write your exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your login in the corner of every page. ☺

1. Basic Operations (6 Points).

a. For the graph below, use Kruskal's algorithm to find the MST. The number on each edge is the weight, and the letter is a unique label you should use in your answer to specify that edge. Provide the edges **in the order they'd be found** by Kruskal's algorithm. Break any ties using the alphabetical label. Use the blanks below. You may not need all blanks. **Give your answer in terms of the alphabetical labels,** e.g. if Kruskal's starts with the edge between vertices 3 and 4, you would write H in the first blank.



__A__ __C__ __G__ __E__ ____ ____

Solution: Kruskal's algorithm works by choosing the edge of lightest weight in the graph and adding it to the MST if adding that edge does not create a cycle. Knowing this and the fact that we break ties by alphabetical order, we get the following solution

Grading: Full points were given for writing the correct edges that Kruskal's finds in the correct order. If you found the correct MST that Kruskal's gives but the edges were in the wrong order, then partial credit was given.

b. Repeat part a, but using Prim's algorithm, starting from vertex **#3**. **As before, give your answer in the order added and in terms of the alphabetical labels.** You may not need all of the blanks.

__C__ __A__ __G__ __E__ ____ ____

Solution: Prim's algorithms works by expanding the MST by adding the "closest" to the MST. That is we pick the edge of lightest weight if that edge has one vertex in the MST and it does not create a cycle. Thus we get the following solution.

Grading: Full points were given for writing the correct edges that Prim's finds in the correct order. If you found the correct MST that Prim's gives but the edges were in the wrong order, then partial credit was given.

c. Suppose we have the array [4, 1, 6, 2, 3, 7, 3, 4]. Give a valid partitioning of this array, using the leftmost item as a pivot. Any partitioning scheme is fine.

__1__ __2__ __3__ __3__ __4__ __4__ __6__ __7__

Solution: We are partitioning based on the number **4**. We put all of the numbers smaller than 4 to the left of the partitioned array, and numbers greater to the right. Thus we get the following partition

Grading: Full points were given if everything to the left of 4 was less than or equal to 4 and everything to the right of 4 was greater than or equal to 4. The order did not matter. No partial credit was given for this question.

d. Give the array that results if we use transform the array [1, 2, 4, 9, 3, 7, 5] into a max heap. Use the sinking based heapify process that we described in class (possibly helpful reminder: this process also happens to be the first step of heap sort).

__9__ __3__ __7__ __2__ __1__ __4__ __5__

Solution: Heapify is taking all of the nodes in the heap and from the bottom up, sinking those nodes to its correct position. Note that this is not the same as taking the elements of the array and adding them into an empty heap one by one. The correct heap is

Grading: Full points were given if you produced the heap that is given by heapify. Partial credit was given if you had a valid max heap.

2. **Basic Operations Turbo Edition (3 Points).**

a. Draw a binary tree that has a pre-order and in-order traversal of A, B, C, D, E.

Solution:

A preorder traversal on a binary tree is a recursive algorithm that visits itself, then its left child (and recursively visits all of its children), then its right child (and recursively all of its children). An inorder traversal, on the other hand, is a recursive algorithm that visits its left child (and all of its children), then itself, then its right child (and all of its children).

Given this information, the following tree is a valid example:

```
[A]
  \
  [B]
    \
    [C]
      \
      [D]
        \
        [E]
```

Grading:

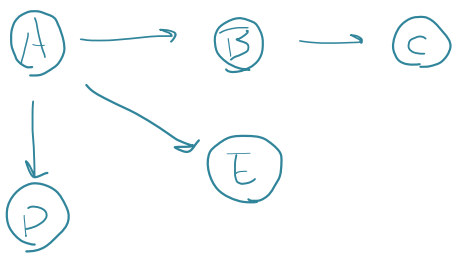
We awarded full credit for any tree that satisfied both the preorder and inorder traversal requirements. Students who drew a tree that satisfied only one of the traversal requirements received partial credit for their efforts. Note that some forms of the exam asked for an example with numbers rather than letters; students who did not follow this were not awarded credit.

Some students submitted two trees: one for the preorder traversal and one for the inorder traversal. In this scenario, credit for one of the trees was given.

b. Draw a **directed** graph whose DFS pre-order traversal is A, B, C, D, E, and whose DFS post-order traversal is C, B, D, E, A. Assume that ties are broken alphabetically.

Solution:

A DFS preorder traversal is depth-first-search, where the traversal order is based on when we visit a node. A DFS postorder traversal is where the traversal order is based on when we exit a node, i.e. have finished visiting all of its neighbors. Given this information, the following graph is a valid example:



Grading:

We awarded full credit for any *directed* graph that satisfied the constraints. Students with submissions that satisfied only one of the traversal orders specified received partial credit for their efforts. Note that some forms of the exam asked for an example with numbers rather than letters; students who did not follow this were not awarded credit.

3. **Best and Worst (4.5 Points).** Give bests and worsts as requested below.

a. Give an input that results in worst case runtime for insertion sort. Give your answer as an array of 5 integers, written in the blanks below.

__5__ __4__ __3__ __2__ __1__

Recall that insertion sort maintains two sections of the array: a sorted portion and an unsorted portion. The first element of the unsorted portion is swapped into place in the sorted portion by comparing its value with its left neighbor and swapping until the element we are sorting is larger than its left neighbor. Thus, to maximize the number of swaps, we want any array which is in purely descending order, because each element will have to be swapped to the very front.

b. Given an initially empty BST, give an insertion order for A, B, C, D, E, F, G that minimizes height. Assume we are not performing any balancing operations.

__D__ __B__ __F__ __A__ __C__ __E__ __G__

Recall a similar question from discussion: the ideal tree is constructed by recursively inserting the median of each subtree.

There were many different possible correct insertion orders such as DBACFEG. The important thing was to represent the tree as a directed graph where each parent node has a directed edge to its children. Any valid topological sort order of this DAG (directed acyclic graph) would be a valid insertion order.

c. Suppose we want to build an array based stack that supports push and pop. In class, we learned how appropriate resizing rules can empower data structures with excellent amortized runtime. Suppose our stack has the following two rules:

- Before each push, if the array **is full, double** the size before pushing.
- After each pop, if the array is less than **half full, halve** the size of the array.

What is the worst-case amortized runtime for a sequence of pushes and pops? Give your answer in $\Theta(\cdot)$ notation in terms of N, the maximum size of the array that is reached during the stack's lifetime.

The worst case scenario is if we fill up the array with push operations, then do a push, then a pop, then a push, then a pop, etc. This will cause the array to resize with every operation in that final sequence. Arrays "resize" in $\Theta(N)$ time because you must allocate the new array, then copy over each of the elements. Thus, each operation runs in amortized $\Theta(N)$ time.

4. Comparative Data Structures and Algorithms (7.5 points).

a. What is the size of the largest binary min heap that is also a valid BST? Draw an example assuming all keys are letters from the English alphabet. **Assume the heap has no duplicate elements.**

Note: For Morning and Evening Form B, see 4(b) for your solution. Also, some of the forms explicitly asked for letters while others asked for numbers.

Solution:

Recall that a heap must satisfy two properties:

(1) Heap-Order Property: In a min-heap, this means that each node is smaller than each of its children.
(2) Completeness: there are no "holes" in the heap as you go through the tree in a level-order traversal.

Recall that a binary search tree (BST) satisfies the binary search invariant: all elements in the left subtree are less than the key in the root, and all elements in the right subtree are greater than the key in the root.
Combining these facts, we know that we can only have elements in the right subtree, but this violates the completeness requirement for a heap. Thus, the maximum size is **1**, and a valid example is shown below:

```
[A]
```

Grading:

Broadly, there were two components to this problem:

(1) +0.75pts if you wrote down the correct size of 1. If you didn't write down a size, but your example clearly shows that the size is 1, we gave you credit here as well. If you wrote down an incorrect size and gave an example that shows a size of 1, you did not receive credit here.
(2) +0.75pts if you wrote down a valid example. Note that if you were asked to provide an example with letters but you gave one with numbers, you did not receive credit.

A few students gave multiple answers, one of which was the correct answer of 1, the other of which was a different size *along with assumptions about relaxing the constraints for a heap or a binary search tree*. If there were also corresponding examples, we awarded a total of 1.0 point.

b. What is the size of the largest binary max heap that is also a valid BST? Draw an example assuming all keys are letters from the English alphabet. **Assume the heap has no duplicate elements.**

Note: For Morning and Evening Form B, see 4(a) for your solution. Also, some of the forms explicitly asked for letters while others asked for numbers.

Solution:

See the detailed explanation above, with the following modification: in a max-heap, each node is larger than each of its children for the heap-order property. Because of this, we are able to have one element in the left subtree, and this does not violate the completeness property as before. Thus, the maximum size is **2** and a valid example is shown below:

```
  [B]
  /
[A]
```

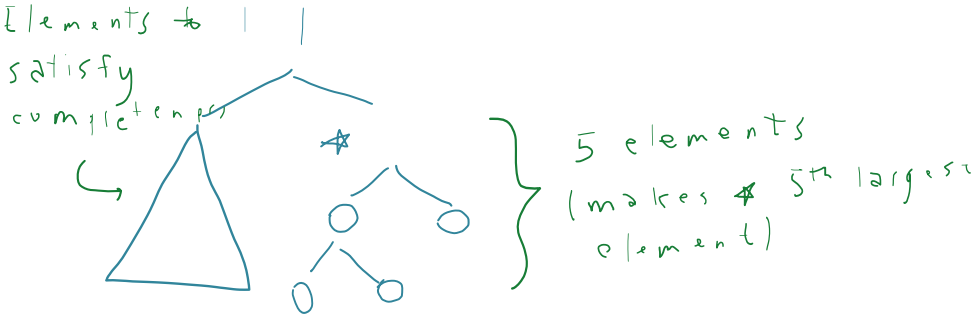
Grading: Same grading scheme as 4(a).

c. What is the size of the largest binary min heap in which the fifth largest element is a child of the root? You do not need to draw an example. **Assume the heap has no duplicate elements.**

Solution:

During the exam, we posted a clarification such that we were interested in "size" as the number of elements. Based on this, the solution is **13**.

We know that the size of the subtree rooted at the fifth largest element is 5, including that element. This gives us the following subtree:



Given this information, the tree structure we get looks something like this:

```
          [1]
       ___/ \____
     [2]         [9*]
     /  \        /  \
  [3]    [4]   [10] [11]
  / \    / \   / \
[5] [6][7] [8][12][13]
```

(continued, next page)

Grading: This question was largely all-or-nothing. The exception was if students specified a height of **3** *and* provided an example that allowed us to see that they understood what the correct solution was (i.e. if they drew the above diagram, or wrote out an explanation which we deemed to show sufficient understanding), we awarded some partial credit as a point adjustment. This was extremely rare, and as such, it is not reflected in all of the rubrics across the different forms.

d. Give an example of when you'd use Quicksort over Mergesort.

There are many situations where we might use Quicksort: When we want faster overall runtime, better memory usage, when we don't care about stability, when we have many duplicates (can use 3-way quick sort (from textbook)).

e. Give an example of when you'd use a Trie-based map instead of a HashMap.

Likewise, there are many cases where we might prefer a Trie: When we want to look for near misses (e.g. spell checking), for autocomplete or other string prefix operations, when we expect to have mismatches early on in a string, to save space for certain pathological inputs with large degrees of overlap at the front of the string, when we want to support ordered operations, to implement alphabet sort from proj3 (not a great answer since a map isn't the right abstraction, but technically correct).

f. Why did we bother introducing heaps for implementing priority queues instead of just using left-leaning red-black binary search trees?

Since you can't read my mind about our motivations for heaps, we were pretty friendly about correct answers. The best answers are:

- Uses less memory.
- Easier to implement and/or understand.
- Pedagogically interesting.
- Handles duplicates gracefully.
- Example of an interesting, useful, and simpler tree invariant.
- As a stepping stone to heaport.

Less good, but arguably correct were:

- Faster overall runtime (far from obvious).
- Faster insert / delete (though this is far from obvious).
- As a building block for the development of asymptotically superior faster heaps (e.g .fibonacci heap, brodal queue)
- Tuning arity (number of children) allows for performance tuning for graph algorithms (not a primary reason to introduce heaps!).
- Heap construction is faster (but construction of a PQ from a collection of existing items isn't the most common thing).
- Better balance (but not really my motivation in introducing the heap. LLRBs are pretty good at balance!)

Common incorrect answers:

- **Faster getMax / getMin / peek**: It is almost trivial to augment a BST so that you can peek at the "top" item. You simply create a min pointer that updates every time you insert or delMax. There is no asymptotic cost. While a naïve BST implementation WOULD be log N for peek / getMax / getMin, this is not the reason that we introduced heaps in class since it's simple to make this operation constant time. We explicitly discussed many other more important and interesting reasons that heaps are better.
- Better asymptotic performance for operations (actually the same).
- Constant delMax or insert for a heap (actually log).
- Simply saying "no need to balance". This isn't really true. Heaps do some work to stay balanced.
- Requires constant space for heapification: This doesn't say enough.

Hier ist eine keine Tausendfüßler Zone.

5. Syntax Mastery (2 points). Give the output of `main`. You may not need all lines provided.

```
reorder (Morning A)
rearward (Morning B)
neonernr (Evening A)
xeoxerxr (Evening B)
```

```
public class Sklarp implements Iterable<Character>, Iterator<Character> {
    public char[] contents;
    public char magicCharacter;
    public int k;
    public Sklarp(char[] s, Character c) {
        contents = s;
        magicCharacter = c;
        k = 0;
    }

    public Iterator<Character> iterator() {
        return this;
    }

    public boolean hasNext() {
        return k < contents.length;
    }

    public Character next() {
        if (k % 3 == 0) {
            contents[k] = magicCharacter;
        }
        char returnChar = contents[k];
        k += 1;
        return returnChar;
    }

    public void remove() { throw new UnsupportedOperationException(); }

    public static void main(String[] args) {
        Sklarp g = new Sklarp("Zeoidei".toCharArray(), 'r');
        for (Character c : g) {
            System.out.print(c);
        }
        System.out.println();
        for (Character c : g) {
            System.out.print(c);
        }
        System.out.println();   }   }
```

6. Runtime Analysis (2.5 points). For each of the pieces of code below, give the **worst case** runtime in $\Theta(\cdot)$ notation as a function of N. Your answer should be as simple as possible (i.e. avoid unnecessary constants, lower order terms, etc.). If the worst case is an infinite loop, write an infinity symbol in the blank. Assume there is no limit to the size of an int (otherwise technically they're all constant).

Note: Different forms had similar functions but were named differently and were in different orders. The ideas behind the solutions are the same. The answers below are very mathematical treatments. Alternately, you can use the intuitive ideas developed in lecture. Each of these pieces of code (except f4) followed an exact pattern from lecture.

Note: For this problem, you were required to simplify your answer as much as possible. Students who did not do this were hit with a one-time penalty across the entire question.

_$\Theta(N^2)$__

```
public static void f1(int N) {
      int sum = 0;
      for (int i = N; i > 0; i -= 1) {
          for (int j = 0; j < i; j += 1) {
              sum += 1;
          }
      }
}
```

The inner loop runs i number of times for each value of i, and each of those runs takes $\Theta(1)$ time. Because we decrement the value of i by 1 each time, i will take on every value from 1 to N inclusive. Thus, we end up with:

$$\Theta(1 + 2 + \cdots + N) = \Theta\left(\frac{N(N+1)}{2}\right) = \Theta(N^2)$$

_$\Theta(N)$___

```
public static void f2(int N) {
      int sum = 0;
      for (int i = 1; i <= N; i = i*2) {
          for (int j = 0; j < i; j += 1) {
              sum += 1;
          }
      }
}
```

The inner loop runs i number of times for each value of i, and each of those runs takes $\Theta(1)$ time. In this case, i doubles with each iteration of the outer loop. Being a little sloppy with my math, we have:

$$\Theta\left(\sum_{k=1}^{\lfloor \log_2 N \rfloor} 2^k\right) = \Theta\left(\frac{1 - 2^{\lfloor \log_2 N \rfloor + 1}}{1 - 2} - 2^0\right) \approx \Theta\left(\frac{1 - 2N}{-1} - 1\right) = \Theta(2N - 2) = \Theta(N)$$

_$\Theta(N^2)$__

```
public static void f3(int[] a) {
      if (a.length == 0) { return; }
      int N = a.length;
      int[] newA = new int[N-1];
      for (int i = 0; i < newA.length; i += 1) {
```

```
            newA[i] = a[i];
        }
        f3(newA);
    }
```

The easiest way to explain this question is to use a diagram to show the amount of work that is being done at each level of recursion:

```
|--- N ---|      Amount of Work at this level:
*** ... ***      N
...
****             4
***              3
**               2
*                1
```

The amount of work that is being done at each level comes from needing to copy each element of the array `a` into `newA`. Thus, we observe the same idea as in `f1`:

$$\Theta(1 + 2 + \cdots + N) = \Theta\left(\frac{N(N+1)}{2}\right) = \Theta(N^2)$$

_$\Theta(\log(N))$

```
public static void f4(int N) {
    int x = N;
    while (x > 0) {
        x = x >>> 1;
    }
}
```

Effectively, this function is right-shifting the bits until all of the bits become zero. Note that the problem asked you to assume that there is no limit to the size of an int. The number of iterations of the while loop is related to the number of bits □ in the integer, and we know that □ = $\lceil \log_2(\square) \rceil$.

_$\Theta(N^2)$___

```
public static void f5(int N) {
    f1(N);            // If you left one or more of the answers
    f2(N);            // above blank, give your answer to f5 in
    f3(new int[N]);   // terms of your non-blank answers.
    f4(N);             }
```

Since we are simply running each of the functions one after another, we just add the runtimes together. In asymptotic analysis, we drop all of the non-dominating terms, so we have:

$\Theta(N^2 + N + N^2 + \log(N)) = \Theta(N^2)$

As long as your answer here was consistent with what you put for f1, f2, f3, and f4 (basically, if you picked the dominating term), you received full credit for this problem.

7. **Sorting Mechanics (5 points).**

a. Below, the leftmost column is an array of strings to be sorted. The column to the far right gives the strings in sorted order. Each of the remaining columns gives the contents of the array during some intermediate step of one of the algorithms listed below. Match each column with its corresponding algorithm. Use every answer exactly once. Write your answers in the blanks provided.

4873	1876	1874	1626	9573	2212	1626
1874	1874	1626	1874	7121	8917	1874
8917	2212	1876	1876	9132	7121	1876
1626	1626	1897	4873	6973	1626	1897
4982	3492	2212	4982	4982	9132	2212
9132	1897	3492	8917	8917	6152	3492
9573	4873	4873	9132	6152	4873	4873
1876	9573	4982	9573	1876	9573	4982
6973	6973	6973	1897	1626	6973	6152
1897	9132	6152	3492	1897	1874	6973
9587	9587	7121	6973	1874	1876	7121
3492	4982	8917	9587	3492	9877	8917
9877	9877	9132	2212	4873	4982	9132
2212	8917	9573	6152	2212	9587	9573
6152	6152	9587	7121	9587	3492	9587
7121	7121	9877	9877	9877	1897	9877
----	----	----	----	----	----	----
__1_	__3_	__6_	__2_	__4_	__5_	__7_

1: Unsorted 2: Merge 3: Quick 4: Heap 5: LSD 6: MSD 7: Sorted

Notes:

- Quicksort: Non-random and uses topmost item as pivot. We have deliberately omitted information about the partitioning strategy.
- Mergesort: Recursive implementation (a.k.a. top-down for textbook readers).
- Heapsort, LSD Radix Sort, and MSD Radix Sort: As described in class.

Each blank was worth 1 point. No partial credit awarded.

Column 2: (3) Quicksort.

We are given that quicksort will use the topmost item as the pivot, which we can tell from Column 1 is 4873. Notice that this column is still mostly unsorted, but we can tell that everything above 4873 (highlighted in blue) is less than 4873 an everything below is greater. This is the distinguishing feature of quicksort.

Column 3: (6) MSD Radix Sort.

Notice that the column seems to have been grouped together based on the most significant (thousands) digit, but within that group, there does not seem to be any particular order. This is clearly most-significant-digit (MSD) radix sort.

Column 4: (2) Mergesort.

Notice that every pair of numbers is in sorted order, but there is no particular ordering between the pairs. This is the result of using merge sort after one level of merging.

Column 5: (4) Heap Sort.

Heap sort uses a max heap and maintains the sorted portion at the back of the array, which I have highlighted in blue. Notice also that the topmost element of the column – 9573 – is the largest remaining item, hinting at the fact that this is indeed a max heap. Clearly, this is heap sort.

Column 6: (5) LSD Radix Sort.

Notice that we seem to have each of the elements sorted based on the ones and tens digits, but not yet by the hundreds digits. This is thus least-significant-digit radix sort.

8. **Choosing a Sort (4 Points).**

You're an imperial engineer doing some QA on a recent batch of droids that were produced. You have a supposedly sorted array of n Droid objects that implement Comparable. However, when looking through your array, you realize these aren't the droids you're looking for! Your supervisor tells you that the machine malfunctioned, and it's made at most k mistakes: There are no more than k inversions, where **we define an inversion as a pair of droids that is not in the right order.**[1]

For this question, we awarded 0.4 pts for having the correct sorting algorithm, and 0.4 pts for having a runtime that was consistent with the sorting algorithm you chose.

a. State the most efficient sorting algorithm and the big O runtime (giving the tightest bound with no unnecessary constants or lower order terms) to sort the droids if:

k = O(n)
Algorithm: __Insertion Sort_____

Runtime: __$O(N)$_____

Insertion sort has a worst-case runtime of $O(N^2)$ on general input, but remember that it works quite well if there are relatively few inversions: it runs in $O(N + k)$ in this setting. In this case, the runtimes for some of the popular answers we received would be:

Insertion Sort	$O(n)$
Merge Sort, Heap Sort, Quick Sort	$O(n \log n)$
Selection Sort	$O(n^2)$

k = O(n^2)
Algorithm: ___Merge Sort, Quick Sort, Heap Sort___

Runtime: ___$O(n \log n)$_____

Any n log n comparison sort would work here. Because we have so many inversions, insertion sort would run in $O(n + n^2) = O(n^2)$. We can do better with other comparison-based sorts, which can run in $O(n \log n)$. Radix sort would not work well here because you don't know anything about the comparisons.

k = log(n)
Algorithm: __Insertion Sort_____

Runtime: __$O(n)$_____

Insertion sort has a worst-case runtime of $O(N^2)$ on general input, but remember that it works quite well if there are relatively few inversions: it runs in $O(N + k)$ in this setting. In this case, the runtimes for some of the popular answers we received would be:

Insertion Sort	$O(n)$
Quick Sort	$O(n)$ or $O(n \log n)$
Merge Sort, Heap Sort	$O(n \log n)$
Selection Sort	$O(n^2)$

[1] In case you've forgotten the term "inversion", here's an example that may help: Suppose we have the array [0 1 1 2 3 4 8 6 9 5 7]. This array has 6 inversions: 8-6, 8-5, 8-7, 6-5, 9-5, 9-7.

b. Two weeks later, you're given another batch of Droids that are supposed to be sorted on a 32-bit int ID, an instance variable of Droid. The sorting machine hasn't been fixed yet and again has made at most k mistakes. State the most efficient sorting algorithm and the runtime (giving the tightest bound with no unnecessary constants or lower order terms) to sort the droids by int ID if:

`k = O(n^2)`
`Algorithm:` Radix Sort

`Runtime:` $O(n)$ or $O(n+r)$

We now know something about the Droids: their IDs. There is a bound on these IDs, so radix sort is a natural choice. Comparison-based sorts will generally be too slow (running at $O(n \log n)$ at best). Note that we did not award credit for runtime for given constants.

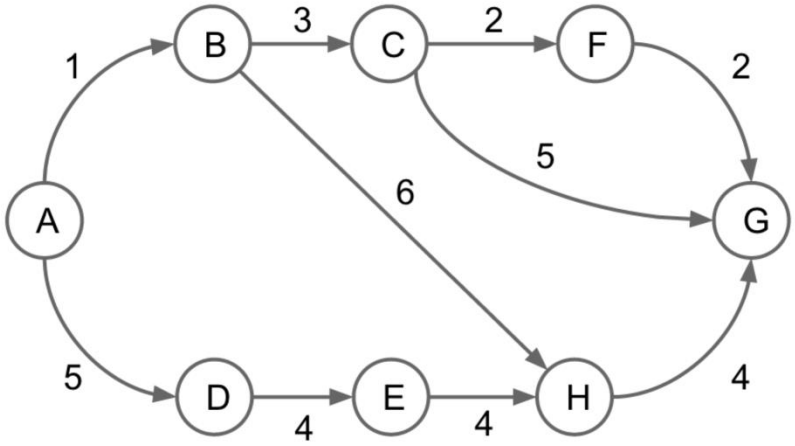
`k <= 5`
`Algorithm:` Insertion Sort, MSD Radix Sort, Bubble Sort

`Runtime:` $O(n)$

Insertion sort works well because there were very few inversions. MSD Radix Sort works well because of the reason specified in the previous part of (b). Bubble sort also works fine in this setting. All other comparison-based sorts will be too slow because they run at $O(n \log n)$ at best.

9. **Shortest Paths Algorithms (4.5 Points).**

a. For the graph below, give the order in which Dijkstra's Algorithm would visit each vertex, starting from vertex A.



__A_ __B__ __C__ __D__ __F__ __H__ __G__ __E__

For Djikstra's, we first insert all nodes in our fringe PQ, ordered by their distances from the source. Initially, these values are set to infinity, except the source, whose value is set to 0. We continually remove the smallest-valued vertex from the PQ and reset values of nodes in the PQ accordingly.

Before we begin:
Fringe: [A (0), B (inf), C (inf), D (inf), E (inf), F (inf), G (inf), H (inf)]

A: Lower cost paths found for B and D.
Fringe: [B (1), D (5), C (inf), E (inf), F (inf), G (inf), H (inf)]

B: Lower cost paths found for C and H.
Fringe: [C (4), D (5), H (7), E (inf), F (inf), G (inf)]

C: Lower cost paths found for F and G.
Fringe: [D (5), F (6), H (7), G (9), E (inf)]

D: Lower cost path found for E.
Fringe: [F (6), H (7), E (9), G (9)]

F: Lower cost path found for G:
Fringe: [H (7), G (8), E (9)]

H: No lower cost path found:
Fringe: [G (8), E (9)]
G: No children nodes

Fringe: [E (9)]

E: No lower cost path found (H was already removed from the fringe):
Fringe: [ ]

So, the ordering in which we visited nodes was: A, B, C, D, F, H, G, E.

b. Change one of the weights in the graph so that the shortest paths tree returned by Dijkstra's is not correct. Hint: we showed in class that Dijkstra's returns the correct SPT as long as all edges are non-negative.

   Set the weight of the edge connecting vertex __E__ and vertex __H__ to __<= -3__.

To make the shortest paths tree incorrect, we need to find an edge in the graph where the node on the opposite end of the node being visited (i.e. the neighboring node) doesn't get considered because it had already been visited.

This is because we should be able to disregard the edge weight because we should be able to assume that if it's in the closed set, the shortest path to that node had already been founded. However, this does not hold true for graphs that have non-negative edge weights.

E-H is the only edge for which this applies because H had already been taken out of the priority queue.

Now that we know the weight of the edge is not considered, we need to find a value that will invalidate the shortest paths tree. Since we know E-H is the correct edge, we have to find shortest paths that go through H. We see that A-B-H has a cost of 7, and A-B-H-G has a cost of 11. We know that A-D-E has a cost of 9, so we need to set the weight of E-H to a value such that A-B-H or A-B-H-G is no longer the shortest path to H or G. To do this, we can set E-H to any value <= -3, so A-D-E-H has a cost of <= 6, while A-B-H has a cost of 7.

c. Suppose we use the following heuristic:

```
h(A) = 2
h(B) = 2
h(C) = 20
h(D) = 2
h(E) = 6
h(F) = 2
h(G) = 0
h(H) = 2
```

Recall that A* is just Dijkstra's algorithm, except that vertices are given a priority equal to the sum of their Dijkstra priority (distance from the source) plus the heuristic distance, and also that we quit when the target is visited.

Give the path (not order visited) that A* returns from A to G, you may not need all of the blanks:

__A_ __B__ __H__ __G__

For A* search, we visit vertices in the order of the known cost to the node plus the heuristic for a particular node (i.e. the estimate for that node to the goal). When we've found our goal node, we're done! A* is guaranteed to yield the shortest path if the heuristic is admissible. This means that the value returned by the heuristic must never overestimate the actual cost of the path.

We see h(C) = 20, but the actual cost to G is 4, so the heuristic is not admissible, and A* is not guaranteed to return the correct shortest path (A-B-C-F-G). In fact, it won't!

Before we begin:
[A (2), B (inf), C, (inf), D (inf), E (inf), F (inf), G (inf), H (inf)]

A: Smaller estimates found for B and D.
[B (3), D (7), C (inf), E (inf), F (inf), G (inf), H (inf)]

B: Smaller estimates found for C and H.
[D (7), H (9), C (24), E (inf), F (inf), G (inf)]

D: Smaller estimate found for E.
[H (9), E (15), C (24), F (inf), G (inf)]

H: Smaller estimate found for G.
[G (11), E (15), C (24), F (inf)]

G: At goal node, done!

Shortest path to G, according to this heuristic, is A-B-H-G.

10. **Hashing and Doubly Linked Lists (7 points).** Mu Gulshan, a 61B student, was working on a DLList class. Unfortunately, Mu was transformed into an owl by a jealous Zeus before completing the implementation, so you'll need to pick up where Mu left off. For space reasons, the code for `insertFront` is hidden, but you can assume it works properly. **You may not need all lines.**

```
public class DLList {
    public int size = 0;
    public DNode sentinel = new DNode(null, null, null, null);

    /** Creates a new node and inserts it at the front of the DLL. */
    public void addToFront(String k, String v) {
        insertFront(new DNode(k, v, null, null));
    }

    /** Inserts the given node into the front of the DLL. */
    public void insertFront(DNode toInsert) { ... }

    /** Returns node containing k, or null if node doesn't exist. */
    public DNode getNode(String k) {
        // See next page
    }



    /** Unlinks the given node from its position in the list. */
    public void detachNode(DNode toDetach) {
        // See next page
    }

    public static class DNode {
        public String key, value;
        public DNode prev, next;
        public DNode(String k, String v, DNode p, DNode n) {
            key = k;
            value = v;
            prev = p;
            next = n;
        }
    }
}
```

Solution:

```
public DNode getNode(String k) {
    Dnode cur = sentinel.next;
    while (cur != null && !cur.key.equals(k)) {
        cur = cur.next;
    }
    return cur;
}
```

Comments: The intended use of sentinel was as a dummy node at the front of the list to make iterating through the list a bit easier (fewer null checks for instance). Some people interpreted it as being a dummy node at the end of the list, or the middle of the list, or the beginning and end of the list (i.e. a circular linked list). We accepted all of these interpretations as long as they were used consistently within `getNode` and `detachNode`.

In this problem we were fairly forgiving of minor errors, including syntax errors, off-by-one errors (such as terminating the traversal of the linked list when `cur.next` was null) and other minor mistakes (such as referring to prev/next as p/n, etc). We also allowed solutions that assume that k is never null and that none of the keys in the linked list are ever null (we even forgave solutions that assumed that `sentinel.key` is not null, even though this is false). In fact, the solution above makes such an assumption. It is a little bit annoying to write the method without some sort of assumption like that. Here is one possibility:

```
public DNode getNode(String k) {
    DNode cur = sentinel.next;
    while (cur != null && !(k == cur.key || (k != null &&
k.equals(cur.key)))) {
        cur = cur.next;
    }
    return cur;
}
```

Or if we are willing to use extra lines:

```
public DNode getNode(String k) {
    DNode cur = sentinel.next;
    while (cur != null) {
        if (k == cur.key || (k != null && k.equals(cur.key))) {
            return cur;
        }
        cur = cur.next;
    }
    return cur;
}
```

Common mistakes for this problem included forgetting to update the pointer, using `==` to compare strings instead of `.equals`, returning things other than `DNodes`, and destroying the list while iterating.

Some people also attempted recursive solutions, which rarely worked (and often destroyed the list).

Now on to `detachNode:`

```
public void detachNode(DNode toDetach) {
    toDetach.prev.next = toDetach.next;
    if (toDetach.next != null) {
        toDetach.next.prev = toDetach.prev;
    }
    size -= 1;
}
```

A few comments: The most common way for people to lose points on this problem was by failing to check that `toDetach.next` was not null (as would occur if `toDetach` was the last node in the list). It is not necessary to check this for `toDetach.prev` because of the sentinel. Solutions that assumed sentinel was at the end of the list in `getNode` lost points for not checking that `toDetach.prev` was not null. Solutions that assumed that the list was a circular linked list did not need to perform any null checks.

A few more notes. It is acceptable to assume that `toDetach` is a node in the linked list and is not the sentinel (though we did not take off points for checking these things of course). It is also not required to change the values of `toDetach.next` and `toDetach.prev` (though once again we didn't take off points for this). We also did not take off points for not decrementing the size variable (though it made us happy when we saw solutions that did). Also, it is not necessary to iterate through the list to find `toDetach` or to use `getNode` (though once again we didn't take off points for this).

Mu built the strange DLL class listed on the previous page as a support class for an optimized HashMap. The main idea is that recently used items are kept at the front of each list. In Mu's scheme, the HashMap (which maps Strings to Strings), behaves as follows:

1. Key/value pairs are stored in nodes in an array of buckets, where each bucket is a doubly linked list.
2. When a key/value pair is inserted into a bucket, we check to see if the key is already in the bucket.
   - If the key/value pair does not exist in the bucket, we create a new node (with that pair) at the front of the bucket.
   - If the key/value pair exists in a node in the bucket, we update that node's value, and move that node to the front.

Mu's last commit to the DLList class was the following method:

```
/** Returns node associated with string k or null if doesn't exist.
  * Moves item to the front if it does exist. */
public DNode getNodeAndPromote(String k) {
    DNode p = getNode(k);
    if (p == null) {  return null;  }
    detachNode(p);
    insertFront(p);
    return p;
}
```

Using this method (and others), fill in the put method below. **You may not need all lines.**

```
public class DLLHashMap {
     public DLList[] data;
     public DLLHashMap(int numBuckets) {
          data = new DLList[numBuckets];
          for (int i = 0; i < numBuckets; i += 1)
               data[i] = new DLList();
     }

      /** If k exists, update value and move to front. Otherwise add new
        * node (containing key and value) to front. */
     public void put(String k, String v) {
          int index = (k.hashCode() << 1 >>> 1) % data.length;
          DLList bucket = data[index];
          DNode p = bucket.getNodeAndPromote(k);
          if (p != null) {
               p.value = v;
          } else {
               bucket.addToFront(k, v);
          }
     }
}
```

There were two major sections to this question:

PART A (2 pts): Finding the correct bucket to check for the key (and to insert the key and value into if the key didn't exist). This also involved writing a good hash function to provide an even distribution in constant time with respect to the number of buckets of the `DLLHashMap`.

- + 0 – No bucket selection strategy or one that does not provide an even distribution of keys
  - Example 1: `int index = 0;`
  - Example 2: `int index = k.length() % data.length;`
- +0.5 – Linear (not constant) time bucket selection strategy that gives nice distribution of keys
  - Some solutions had a smart (but inefficient) idea to check all buckets for a key but insert new key-value pairs into the most empty bucket.
  - This leads to a good distribution of keys in the `HashMap`, but is quite inefficient (taking away one of the main advantages of using a `HashMap`).
- +0.5 – Uses `k.hashCode()` for bucket, but no attempt to coerce into a usable array index
  - Example: `int index = k.hashCode();`
  - Some solutions were on the right track with getting the hash code of the key, but did not ensure that they were turned into a usable index.
  - In many cases, the hash code of a string will be larger than the number of buckets, thus causing an `IndexOutOfBounds Exception`.
- +1.5 – Picks good bucket in constant time, but fails for negative numbers (or some other large range of hash codes)
  - Example: `int index = k.hashCode() % data.length;`
  - This solution is far more correct than the solution above as it works for all positive hash codes. It only misses an edge case specific to Java's mod operator.
  - Java's mod operator returns a number of the same sign as the input to the left of the mod. Thus, negative hash codes for the key will return negative indices.
  - Because this was a small edge case, we gave mostly all of the points for this case.
- +1.9 – Picks good bucket in constant time, but fails for `Integer.MIN_VALUE` or overflows on edge case
  - Example 1: `int index = ((k.hashCode() % data.length) + data.length) % data.length;`
  - Example 2: `int index = Math.abs(k.hashCode()) % data.length;`
  - These solutions are so close to working properly. Unfortunately, they fail on a couple edge cases. But we gave mostly all the points for this first hashing part.
  - Example 1: Fails when `data.length` is large enough that add it again to `k.hashCode()` again causes overflow.
  - Example 2: Fails when `k.hashCode()` is equal to `Integer.MIN_VALUE`. This is because there is no corresponding positive value for `Integer.MIN_VALUE`. Read the Java docs for `Math` if you are interested. Or take CS 61C!
- +2 – Found the appropriate bucket in constant time
  - Example: `int index = (k.hashCode() << 1 >>> 1) % data.length;`
  - This solution is perfect because it turns the hash code into a positive number.

PARTS B-D (1.5 pts): The rest of this question was about promoting (moving to front) and updating nodes that already existed and creating new nodes in the case that they didn't exist. We graded this part independently of finding a good bucket, so you could still get partial credit on this part.

- +0.5 – Part B (got and promoted node with key `k` in the correct context)
  - Example: `bucket.getNodeAndPromote(k);`
  - Any solution that had the line above (or a corresponding line) received credit for this.
- +0.5 – Part C (correctly added new node to front of exactly one list)
  - Example: `bucket.addToFront(k, v);`
  - We also gave credit to those who chose to use `DLList`'s `insertFront` method.
  - Credit was not given to solutions that inserted new nodes even though the node already existed in the `DLList` as that would create duplicate nodes with the same key – thus violating the `HashMap` invariant!
- +0.5 – Part D (update node if it already existed)
  - Example: `p.value = v;`
  - This point was dependent on getting pre-existing nodes properly.

DEDUCTIONS (-0.5 pts): We were actually extremely lenient on most typos/minor syntax errors. One of the only major syntax errors we deducted points for was violating the `DLLHashMap/DLList` abstraction barrier.

- -0.5 – Assumes `getNodeAndPromote` is a method of `DLLHashMap`
  - Example 1: `this.getNodeAndPromote(k);`
  - Example 2: `data.getNodeAndPromote(k);`
  - `addToFront`, `insertFront`, `getNode`, `detachNode`, and `getNodeAndPromote` are all methods in `DLList`.
  - We deducted from solutions that called any of these methods on a `DLLHashMap` (as in Example 1) or `DLList[]` (as in Example 2).

11. PNH (0 points). This remote surfing spot was "discovered" via Google Earth in the 2000s, and is known as one of the most magnificently long barreling waves in the world. Either of its common names (as given by its discoverers, or by the people already surfing it before it was discovered) is OK.

Skeleton Bay or Donkey Bay

12. Radix Sorting and Bits (8 Points).

a. In LSD radix sorting, we sort our inputs digit by digit, starting from the least significant digit and working our way leftwards. For each digit, we used counting sort. Conceptually, we can imagine that we used a subroutine `countSort(Something[] a, int k)`, which sorts the array `a` of something based on the `k`th digit of each object in `a`.

Suppose that we replaced counting sort with a special version of insertion sort `insertionSort(Something[] a, int k)` that insertion sorts based on only the kth digit of each `Something`. Would LSD radix sort still work? What would be the worst-case runtime in terms of N and W, where N is the number of keys and W is the width of the keys in digits? Justification is optional but may be considered for partial credit.

Correct (yes/no): Yes, Insertion Sort is stable
Runtime, big O: $O(WN^2)$

b. Same question as a, but using `heapSort(Something[] a, int k)` to sort on each digit instead of counting sort: Would this version of LSD radix sort yield the correct result? What would be the worst case runtime in terms of N and W? Justification is optional but may be considered for partial credit.

Correct (yes/no): No, Heap Sort is not stable
Runtime, big O: O(WN logN)

c. The absolute value function in Java can be implemented as:

```
public static int abs(int x) {
      if (x < 0) { return (~x + 1); }
      return x;
}
```

This function works correctly for every integer value in Java except one. Which value is this, and why? Recall that the `~x` operation flips all the bits of x.

Integer.MIN_VALUE (or -2^{31}) will return itself.

d. Suppose we are given an LSD radix sorting function described below.

```
public class LSDRadixSorter {
      /** Sorts input digit-by-digit. */
      public static void LSDRadixSort(Digitible[] a) { ... }
}
```

LSDSorter requires that the inputs implement the Digitable interface, described below:

```
public interface Digitible {
      /** Returns the number of digits. */
      public int numDigits();
      /** Returns the kth digit. */
      public int kthDigit(int k);
}
```

Fill in `kthDigit` so that the result of our LSD radix sort is the same as if we used a regular comparison based sort. You may not need all lines given.

```
public class KetchupFriend implements Comparable<KetchupFriend>, Digitable {
      public int ketchupLevel;     // assume non-negative
      public int rednessQuotient;  // assume non-negative
      public int compareTo(KetchupFriend other) {
            if (ketchupLevel > other.ketchupLevel) { return 1; }
            else if (ketchupLevel < other.ketchupLevel) { return -1; }
            if (rednessQuotient > other.rednessQuotient) { return 1; }
            else if (rednessQuotient < other.rednessQuotient) { return -1; }
            return 0;
      }

      public int numDigits() {
            return 64;
      }

      /** Returns kth character where k = 0 is the least
        * significant digit. */
      public int kthDigit(int k) {
            // See below for example solutions
      }
}
```

Here is the big picture before detailed example solutions:

- The `compareTo` method of `KetchupFriend` showed that we compare two `KetchupFriend` objects firstly on their `ketchupLevel` then on their `rednessQuotient` in the case of ties.
- If `KetchupFriend` is radix sortable (because it implements the `Digitable` interface), then the kth digit must be the digit to be sorted on during the kth pass of LSD radix sort.
- The really big hint was that the number of digits was 64. `ketchupLevel` and `rednessQuotient` are both `int`s, and Java `int`s are 32 bits long.
- The 0th digit (LSD) of `KetchupFriend` should return the 0th bit (LSB) of `rednessQuotient`, and the 63rd digit (MSD) of `KetchupFriend` should return the 31st bit (MSB) of `ketchupLevel`.
- Returning the correct "digits" for `KetchupFriend` ensures that the `KetchupFriend` objects are sorted the same way with comparison based sorts (that use the `compareTo` method) as with counting/radix based sorts (that use the `kthDigit` method).

ketchupLevel	rednessQuotient
bits 32-63	bits 0-31

Example 1: Using a mask to get the correct bit of `ketchupLevel` or `rednessQuotient`

```
public int kthDigit(int k) {
      if (k < 32) {
            return (rednessQuotient >>> k) & 1;
      }
      return (ketchupLevel >>> (k – 32)) & 1;
}
```

- This solution shifts `rednessQuotient` or `ketchupLevel` (depending on k) till the kth bit is at the rightmost bit. Then it performs a bitwise and with the mask 1 to get the correct bit.
- A perfectly acceptable variation of this solution is to use the arithmetic right shift `(>>)` rather than the logical. This is because `rednessQuotient` and `ketchupLevel` are guaranteed to be positive.
- This solution is ideal because the size of the alphabet represented by each digit (aka. the number of buckets required for counting sort) is 2. (In shorthand notation, R = 2.) Forgetting the mask at the end leads to a technically correct solution in that LSD radix sort still performs the same way, but R = 2^31 (and LSD radix sort is thus potentially less efficient). We still gave full credit for this variation, however.

Example 2: Using shifts to get the correct bit of `ketchupLevel` or `rednessQuotient`

```
public int kthDigit(int k) {
      if (k < 32) {
            return (rednessQuotient << (31 - k)) >>> 31;
      }
      return (ketchupLevel << (63 - k)) >>> 31;
}
```

- This solution uses shifts rather than a bitmask to achieve the same thing as Example 1.
- The left shift first destroys the bits above the kth bit (more significant bits), and the logical right shift destroys the bits below the kth bit (less significant bits).
- Note that in this case, it was necessary to use the logical right rather than arithmetic right. It is possible that the kth bit is 1, in which case using the arithmetic right `(>>)` would result in a negative number rather than the correct bit.

Example 3: Returning only two useful digits and dummy digits for the rest of the numbers

```
public int kthDigit(int k) {
      if (k == 0) {
            return rednessQuotient;
      }
      return ketchupLevel;
}
```

- Surprisingly enough, something the staff had not anticipated was that a solution like this would actually work in the sense that LSD radix sort would still work properly.
- We debated for a while about giving it less credit than the previous two examples because it missed the heart of the question (treating each bit of `rednessQuotient` and `ketchupLevel` as digits for `KetchupFriend`).
- In the end, we decided to give it full credit because it fulfilled the specification that the result of LSD radix sort is the same as if we used a regular comparison based sort (using the `compareTo` method).
- However, many solutions similar to this example were docked points for returning undefined things for `k != 0` and `k != 1`.
- As a teaching point though, while this solution works, it is potentially less efficient than the prior two examples (depending on the internal implementation of our radix sort) because the digits represent an alphabet size of 2^31 rather than 2 (as would be represented by bits).

Example 4: Other types of solutions

- Finally, there were other types of solutions for alphabet size 10 that were given either full or partial credit depending on varying levels of correctness.
- I have not included those for the sake of brevity.

13. Shortest Paths Algorithm Design (5 points).

Warning: This problem is particularly challenging. **Do not start until you feel like you've done everything else you can.** We will be award very little partial credit for this problem. Solutions which are correct but do not meet our time and space requirements (below) will be not be awarded credit.

a. Design an efficient algorithm for the following problem: Given a weighted, directed graph G where the weights of every edge in G are all integers between 1 and 10, and a starting vertex s in G, find the distance from s to every other vertex in the graph (where the distance between two vertices is defined as the weight of the shortest path connecting them, or infinity if no such path exists).

Your algorithm must run faster than Dijkstra's to receive credit.

For every edge e in the graph, replace e with a chain of w-1 vertices (where w is the weight of e) where the two ends of the chain are the endpoints of e. Pictorially:

```
      3
x ---> y      ->      x ---> a ---> b ---> y
```

Then run BFS on the modified graph, keeping track of the distance from v to each vertex from the original graph.

Alternate solution:

Run Dijkstra's on the graph, but using a different implementation of priority queue. Namely, the priority queue will consist of an array of 11 linked lists. The linked lists will store vertices in the graph and the buckets in the array will correspond to distances from s. We will also keep track of a variable, counter, that represents our current position in the array. First we insert s into the 0th bucket and set counter to 0. Here is how the operations in this priority queue implementation will work:

remove min: If the bucket with index counter is nonempty, remove and return the first vertex in its linked list. Otherwise, increment counter until we find a nonempty bucket. If counter reaches 11, reset it to 0 and continue (so in effect our array is circular).

insert: given a vertex v and a distance d, insert the vertex into the beginning of the linked list at index d%11 in the bucket.

We will also keep track of which vertices we have visited so far, their best known distances from s, and the edge that led us to them. When we remove a vertex from the priority queue, if it had already been visited we will ignore it (thus we don't need to update the priorities of vertices in the priority queue, we just allow duplicates). Once the priority queue is empty, set the distances of all unvisited vertices to infinity.

This strategy works because the vertices we add to the priority queue at any time have a distance that is no more than 10 greater than the current distance.

Everybody who got the question correct had some variation on one of the above two strategies. The first strategy was a little less common, but usually resulted in fewer mistakes. We were fairly lenient with mistakes in solutions that basically had the right idea.

Other attempts:

A few people proposed using a topological sort of the graph. Unfortunately, this only works when the graph has no cycles (which was not necessarily the case here).

Some people also proposed using counting sort to sort the edges by weight and then use Kruskal's algorithm. There are two problems with this: one is that it only really makes sense to talk about spanning trees in the context of undirected, connected graphs. The other is that the minimum spanning tree of a graph does not always give you the shortest paths tree from a vertex.

There were a number of other attempts that didn't work, including people who tried to use modifications of DFS, BFS, Prim's, radix sorts, and tries. Other ideas included quantum computers, supercomputers, grumpy cats and magic.

b. Give the running time of your algorithm in terms of V and E (the number of vertices in the graph and the number of edges in the graph).

First solution: The new graph has at most 10 edges for every edge in the original graph, and at most $|V| + 9|E|$ vertices. So BFS runs in $\Theta(|V| + 9|E| + 10|E|) = \Theta(|V| + |E|)$ time.

Second solution: Remove min takes a constant amount of time (we just have to check if up to 11 buckets are empty and then remove the first element from a single linked list) and insert also takes constant time. We do up to E inserts and up to E remove mins (because of the possibility of duplicates). Thus the whole thing takes $O(V + E)$ time.