# Section Solution

### Discussion Problem 1 Solution: NEWS Mazes

Because the maze isn't necessarily rectangular, we refine what it means to be "in bounds" by checking to see if a supplied **coord** is part of the **Grid** and houses an actionable **rule**. (Note that because **maze** is a reference to an immutable **Grid**, we need to use **get** instead of the more convenient **operator[][]** notation.)

```
static bool inBounds(const Grid<rule>& maze, const coord& curr) {
   return
      maze.inBounds(curr.row, curr.col) &&
      maze.get(curr.row, curr.col) != OffLimits;
}
```

A five-argument version of **pathExists**—at least as I implement it—takes the **rule** that navigated us into the current coordinate (as it might dictate how we move away from this coordinate if the current coordinate houses a **Bridge**), and it also references a **Set** of previously visited coordinates, so we don't get caught up in some infinite loopy recursion. It's this five-argument version actually does the depth-first search via recursion backtracking, and the three-argument version—the one you've been instructed to implement—wraps a call to it.

```
static bool pathExists(const Grid<rule>& maze,
                       const coord& curr, const coord& finish,
                       rule prevRule, Set<coord>& visited) {

   if (!inBounds(maze, curr)) return false;
   if (visited.contains(curr)) return false;
   if (curr == finish) return true;

   rule currRule = maze.get(curr.row, curr.col);
   Vector<rule> possibilities;
   switch (currRule) {
      case North: case East: case West: /* fall through */
      case South: possibilities += currRule; break;
      case Bridge: possibilities += prevRule; break;
      case Any: possibilities += North, East, West, South; break;
      default: /* do nothing */ break;
   }

   if (currRule != Bridge) visited.add(curr);
   foreach (rule r in possibilities) {
      coord neighbor = nextCoord(curr, r);
      if (pathExists(maze, neighbor, finish, r, visited)) {
         return true;
      }
   }

   visited.remove(curr);
   return false;
}
```

Note that we never add **Bridge**s to the visited **Set**, because it's possible we pass through a **Bridge** multiple times on our way to the finish coordinate. One might think that's a problem, but because it's the only one we allow to be visited multiple times, we're protected from loops, because some non-**Bridge** would need to precede some **Bridge** in the loop, and that non-**Bridge** would eventually be in the **visited Set**.

Of course, there's the wrapper call, which constructs an empty **Set** and supplies it to the five-argument version. We also contrive a **prevRule** of **Any**, knowing that it'll be ignored, since we're promised that **start** houses a non-**Bridge**, non-**OffLimits** rule.

```
static bool pathExists(const Grid<rule>& maze,
                       const coord& start, const coord& finish) {
    Set<coord> visited;
    return pathExists(maze, start, finish, Any, visited);
}
```

**Discussion Problem 2 Solution: Stable Counting Sort**

```
struct entry {
   int key;
   string name;
};

static void stableCountingSort(Vector<entry>& v) {
   Queue<entry> buckets[101];
   for (int i = 0; i < v.size(); i++) {
      buckets[v[i].key].enqueue(v[i]);
   }

   v.clear();
   for (int k = 0; k < 101; k++) {
      while (!buckets[k].isEmpty()) {
         v.add(buckets[k].dequeue());
      }
   }
}
```

Why is the running time only **O(n)**? Because each element is enqueued and eventually dequeued exactly once, and **enqueue** and **dequeue** run in constant—or O(1)—time. All comparison sorts run in time **O(n log n)** time or more, but the counting sort approach—which has advance knowledge that the elements being sorted fall into one of 101 different categories —can be implemented to run even more quickly. In general, we should rely on quicksort for the vast majority of our sorting needs. But occasionally you can make use of a counting sort-like algorithm if you know the range of possible values is small and detect it really makes a difference in running time.

Truth be told, you should almost always use quicksort. C's **qsort** and C++'s **sort** functions implement quicksort, and 99.9% of C and C++ programmers just use it instead of hand-coding anything custom.

**Discussion Problem 3 Solution: Order Statistics**

The version of partition used below is precisely the same as that shown in lecture, save for the addition of the first line, which guards against the situation where it's asked to partition a sub-vector of size 1 (quicksort only calls partition on sub-vectors of size 2 or more, so the guard wasn't needed there).

```
static int partition(Vector<int>& v, int low, int high) {
   if (low == high) return low; // guard against arrays of size 0 or 1
   int pivot = v[low], left = low + 1, right = high;
   while (true) {
      while (left < right && v[right] >= pivot) right--;
      while (left < right && v[left] < pivot) left++;
      if (left == right) break;
      swap(v[left], v[right]);
   }

   if (v[left] >= pivot) return low;
   swap(v[low], v[left]);
   return left;
}
```

That **`partition`** returns the overall rank of its pivot within the original vector is particularly good for **`rfind`**. The pivot's rank, combined with the fact that **`partition`** moves all things small (compared to the pivot, anyway) to the left and all things large to the right, allows us to iteratively reframe the **`rfind`** call in terms of an ever-decreasing range of indices.

```
static int rfind(const Vector<int>& v, int rank) {
   if (rank < 0 || rank >= v.size())
      error("Bogus rank of " + integerToString(rank) + " passed to rfind.");

   Vector<int> copy = v; // don't change client's vector
   int low = 0;
   int high = v.size() - 1;
   while (true) {
      int mid = partition(copy, low, high);
      if (rank == mid) return copy[mid];
      if (rank < mid) high = mid - 1;
      else low = mid + 1;
   }
}
```

**Lab Problem 1 Solution: Interpolation Search**

The following forms the core of my own solution, but there are many legitimate variations on my approach.

```
static int interpolate(Vector<double>& values, double key, int lh, int rh) {
   if (values[lh] >= values[rh]) {
      error("Shouldn't call interpolate when endpoints are the same.");
   }

   double full = values[rh] - values[lh];
   double partial = key - values[lh];
   double percentile = partial / full;

   int width = rh - lh + 1;
   return lh + percentile * width;  // downcast to int on the way out
}

static int search(Vector<double>& values, double key) {
   int lh = 0;
   int rh = values.size() - 1;

   while (lh <= rh && key >= values[lh] && key <= values[rh]) {
      if (values[lh] == values[rh]) return lh;
      if (key == values[rh]) return rh;
      int pos = interpolate(values, key, lh, rh);
      if (values[pos] == key) return pos;
      if (values[pos] > key) rh = pos - 1;
      else lh = pos + 1;
   }

   return -1;
}
```