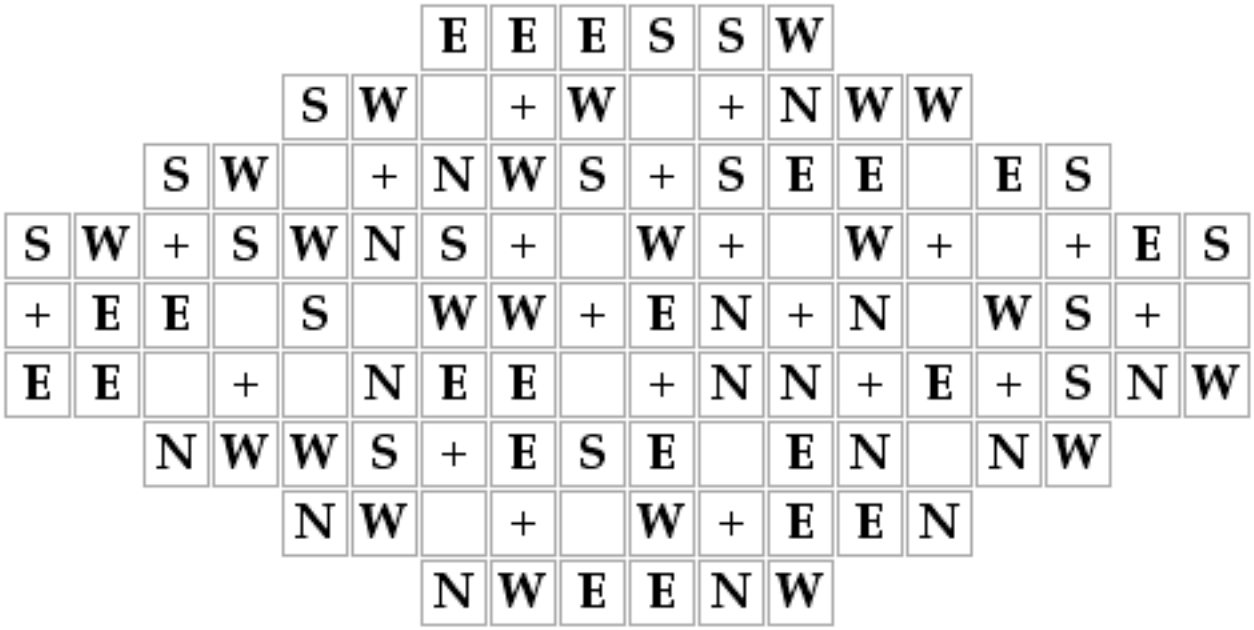
Section Handout

Discussion Problem 1: NEWS Puzzles

NEWS puzzles are maze-like constructions where the goal is to navigate from one location to another. You're free to wander up, down, left, and right as you please, except that moves are constrained by a set of rules. Here they are:

- If a particular coordinate is labeled with N, E, W, or S, then the next coordinate must reside due north, east, west, or south, respectively.
- If a particular coordinate is labeled with a +, then the next coordinate might reside in any of the four major compass directions.
- If a particular coordinate isn't labeled, then that coordinate is considered a bridge, and you must pass straight through that coordinate—that is, if you move east into an unlabeled coordinate, you must move east again.



The puzzle itself is modeled as a **Grid<rule>**, where a **rule** is defined here:

```
enum rule {
   North, East, West, South,
   Bridge,   // rule identifying a coordinate as a bridge.
   Any,      // rule allowing movement in any of the four compass directions
   OffLimits // "rule" asserting that the coordinate isn't part of the puzzle
};
```

Combine the above with a reasonable definition of a **coord**

```
struct coord {
   int row;
   int col;
};
```

and a sure-to-be-useful helper function

```
static coord nextCoord(const coord& c, rule r) {
   coord next = c;
   switch (r) {
      case North: next.row--; break;
      case East: next.col++; break;
      case West: next.col--; break;
      case South: next.row++; break;
   }

   return next;
}
```

and you're equipped to implement a recursive backtracking routine which decides whether a path from one coordinate to another exists while respecting all rules. For simplicity, you should assume that:

- the **coord** type magically works with all relational operators, including **<** and **==**.
- the initial location houses either an N, E, W, S, or +, so there's no confusion how to initiate a search.

Implement the **pathExists** function, which returns **true** if and only if one can travel from **start** to **finish** while respecting the rules. You needn't return the path itself—just the **true** or the **false**.

```
static bool pathExists(const Grid<rule>& maze,
                       const coord& start, const coord& finish);
```

Discussion Problem 2: Stable Counting Sort

Consider the following data structure:

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

where it's understood the **key** field stores an integer between 0 and 100, inclusive.

Write the **stableCountingSort** function, which accepts a reference to a **Vector<entry>** and sorts it, not by calling insertion sort or merge sort or any of the sorting routines we've studied in lecture, but by implementing a **stable counting sort**. Counting sort—at least the way you're to implement it for this problem—would build a **Vector** of 101 **Queue<entry>**s—one for each of the possible keys—and distribute all of the incoming entries to the proper queues. Then all of the queues can be depleted—from index 0 through index 100—to lay down a sorted sequence into the original **Vector<entry>**. If two entries have the same key, then their relative order in the original sequence must be preserved. (This algorithm runs in **O(n)** time, where **n** is the size of the sequence. Why do you think it's faster than even merge sort?)

```
static void stableCountingSort(Vector<entry>& v);
```

Discussion Problem 3: Order Statistics

Implement a routine called **rfind**, which searches an unsorted **Vector<int>** and returns the element of the specified **rank**. Requesting the element of rank 0 is a request to return the smallest element, and requesting an element of rank n - 1, where n is the length of the sequence, is a request to return the largest.

The classic implementation of **rfind** leverages the implementation of **quicksort**'s **partition**, which we looked at in lecture. It should first create a clone of the vector (so that the original one can be left unmodified), and then **partition** the clone around a pivot as **quicksort** would. By examining the **partition**'s return value, we can decide which part of the partitioned sequence—the part preceding the return index, or the part coming after it—should be iteratively examined in order to discover what element ranks as the **rank**th element overall.

```
static int rfind(const Vector<int>& v, int rank);
```

For an arbitrary permutation of a collection of distinct numbers, the algorithm you'll be writing runs in **O(n)** time in the best case, and **O(n^2)** in the worst case.

Lab Problem 1: Interpolation Search

Interpolation search, like binary search, can be used to efficiently search a sorted vector of numbers. Binary search always compares the median element—that is, the element at the 50th percentile point—to decide which half to discard and which half to continue searching. Interpolation search doesn't just choose the median for comparison, but instead chooses a value based on how close the search term is to the smallest and largest numbers. For instance, if you're searching for the number 200 in a sorted vector, and the vector contains numbers from 0 up through 1000, you might choose to compare your 200 not to the *median* element, but the one at the 20th percentile point. As needed, and using the same technique, you recur or iterate on an increasingly smaller range until you find, or fail to find, a match.

Interpolation search mimics how we'd search a physical phone book for, say, the last name **Crandall**—we wouldn't really open it up to the middle and launch a binary search. We'd open it up near the front—maybe 15% of the way in—and spawn something of an interpolation search instead.

Implement the **`search`** function, which takes a **`Vector<double>`**—sorted from low to high—and a key, and returns the index of the key within the vector, or -1 if it's not present. You must implement search using the interpolation technique described above.

```
static int search(Vector<double>& values, double key);
```

In the best case, the algorithm runs in **O(log log n)** time, and in the worst case, runs in **O(n)** time. What types of vectors and search values bring out the best and worst case behavior?