Section Handout Addendum

During staff meeting on Monday, the section leaders mentioned that many of you were eager to see more recursive backtracking questions. Here are a bunch of additional problems and their solutions. All of them are drawn from previous section handouts, midterms, and final exams.

Problem 1: Scheduling Movies

The Academy Awards are fast approaching, and you've been so busy with school and summer internships that you've not seen a single movie all year. You've decided to set aside a single Saturday to see a predefined list of nominated movies, and you're hoping there's some way to fit them all in. Given a dictionary of movies (each bundling the title, the length in minutes, and its various start times into a single **`struct`**) and the titles you want to see, write a function that decides whether or not it's possible to see everything you want to see in any given day. Of course, you can never be in two different theaters at the same time, but for simplicity, assume that you can attend a movie that begins at the same time another movie ends. Write a function that returns **`true`** if and only if it's possible to see all the movies you want to see, and **`false`** otherwise. Notice that you needn't generate the schedule itself—just a yes or no as to whether a schedule exists.

Your routine should return as soon as it can produce a **`true`** or **`false`**, as all of our recursive backtracking examples have.

```
struct interval {
   int start;
   int end;
};

static bool intervalsOverlap(const interval& one, const interval& two) {
   return ((one.start >= two.start && one.start < two.end) ||
           (two.start >= one.start && two.start < one.end));
}

struct movie {
   string name;
   int duration;           // in minutes
   Vector<int> showTimes;  // each in minutes since midnight
};

static bool canSchedule(Vector<string>& titles, Map<string, movie>& schedule);
```

Problem 2: Arithmetic Puzzles

Determine whether it's possible to combine all of the numbers in the given **`Vector<int>`** using simple addition, subtraction, and multiplication. It's possible, for example, to combine the numbers 2, 4, 6 and 8 to get 0. Here's how:

8 – 6 * 2 – 4 equals 0

Given the numbers 1, 4, 7, and 9, we can form the number 20. Here's how:

4 – 1 * 9 – 7 equals 20

Note that all operators have equal precedence, and no parentheses are allowed.

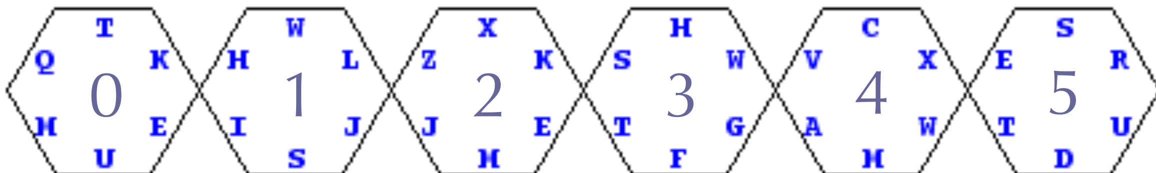
Write the **`combine`** function, which takes a **`Vector<int>`** of numbers and a target value and returns **`true`** if and only if all of the operands can be combined using **equal-precedent** addition, subtraction, and multiplication to form the given target. By equal precedent, we mean don't respect the normal order of operations where multiplication takes higher precedence than addition and subtraction. Just assume all expressions are evaluated from left to right (so that, for example, 4 – 2 * 2 is 4 and not 0.)

```
static bool combine(Vector<int>& numbers, int target);
```

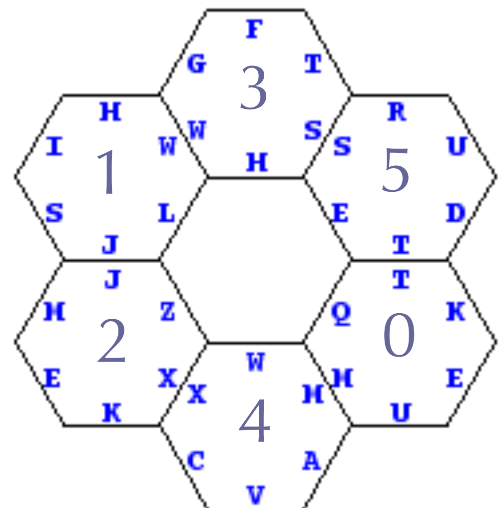
Problem 3: Beehive Puzzle

You're given six hexagonal games pieces, where each piece has a series of six letters, one per edge, around its perimeter. You're interested in finding an arrangement of the six pieces so that all six wrap a hollow center, and all letters on adjacent edges match.

So, if you're given the following **`Vector`** of six pieces:



you should be able to permute and rotate as necessary in order to discover the arrangement on the right.



Write a function called **`arrangementExists`**, which takes a **`Vector`** of six **`string`**s, where each **`string`** consists of the six characters along the perimeter of one hexagonal piece. The order of the letters dictates the clockwise ordering along the perimeter of the piece, but the piece may be rotated by any multiple of 60 degrees if it'll help to produce a solution. **`arrangementExists`** reports back a **`true`** or a **`false`**—**`true`** if and only if some solution could be found. You don't need to return

what the solution is. You only need to return a yes or a no.

```
static bool arrangementExists(Vector<string>& pieces);
```

Solution 1: Scheduling Movies

```
static bool canScheduleMovie(const interval& proposed,
                             const Vector<interval>& scheduled) {
   for (int i = 0; i < scheduled.size(); i++) {
      const interval& scheduledInterval = scheduled[i];
      if (intervalsOverlap(proposed, scheduledInterval)) {
         return false;
      }
   }
   return true;
}

static bool canSchedule(Vector<string>& titles, int count,
                        Map<string, movie>& schedule, Vector<interval>& scheduled) {
   if (titles.size() == count) return true;
   if (!schedule.containsKey(titles[count])) return false;

   const movie& m = schedule[titles[count]];
   for (int i = 0; i < m.showTimes.size(); i++) {
      interval movieInterval = {
         m.showTimes[i], m.showTimes[i] + m.duration
      };
      if (canScheduleMovie(movieInterval, scheduled)) {
         scheduled.add(movieInterval);
         if (canSchedule(titles, count + 1, schedule, scheduled)) return true;
         scheduled.remove(scheduled.size() - 1);
      }
   }
   return false;
}

static bool canSchedule(Vector<string>& titles, Map<string, movie>& schedule) {
    Vector<interval> scheduled;
    return canSchedule(titles, 0, schedule, scheduled);
}
```

Solution 2: Arithmetic Puzzle

```
static bool combine(Vector<int>& numbers, int target, int current) {
   if (numbers.isEmpty()) return target == current;

   for (int i = 0; i < numbers.size(); i++) {
      int number = numbers[i];
      numbers.remove(i);
      if (combine(numbers, target, current + number) ||
          combine(numbers, target, current - number) ||
          combine(numbers, target, current * number)) return true;
      numbers.insert(i, number);
   }
   return false;
}
```

```
static bool combine(Vector<int>& numbers, int target) {
   for (int i = 0; i < numbers.size(); i++) {
      int current = numbers[i];
      numbers.remove(i);
      if (combine(numbers, target, current)) return true;
      numbers.insert(i, current);
   }
   return false;
}
```

Solution 3: Beehive Puzzle

```
static bool ae(Vector<string>& pieces, char start, char bridge) {
   if (pieces.size() == 0) return start == bridge;
   for (int j = 0; j < pieces.size(); j++) {
      string bridgingPiece = pieces[j];
      pieces.remove(j);
      for (int k = 0; k < 6;  k++) {
         if (bridgingPiece[k] == bridge &&
            ae(pieces, start, bridgingPiece[(k + 4) % 6])) return true;
      }
      pieces.insert(j, bridgingPiece);
   }
   return false;
}

static bool arrangementExists(Vector<string> pieces) {
   string anchor = pieces[0];
   pieces.remove(0);
   for (int k = 0; k < 6; k++) {
      if (ae(pieces, anchor[k], anchor[(k + 4) % 6])) return true;
   }
   return false;
}
```