Section Solutions 3

Problem One: Change We Can Believe In

Here's a recursive algorithm for greedy change making. The idea is to find the biggest coin we can that doesn't exceed the total, to use it, then greedily make change for what's left.

```
/**
 * Given a collection of denominations and an amount to give in change, returns
 * the largest denomination that's no larger than the target amount.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @return The highest-denomination coin we can use.
 * @throws If, for some reason, the total is smaller than the smallest coin.
 */
int highestCoinNotExceeding(int cents, const Set<int>& coins) {
    int highest = -1; // Sentinel; will be overwritten
    for (int coin: coins) {
        if (coin <= cents && coin > highest) highest = coin;
    }

    /* In the event that there is no coin we can use at all, report an error
     * because something's wrong.
     */
    if (highest == -1) error("No valid coin found?");
    return highest;
}

/**
 * Given a collection of denominations and an amount to give in change, returns
 * the number of coins required using the greedy change making approach.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @return The number of coins needed to make change using the greedy approach.
 * @throws If it's not possible to make change for this total.
 */
int greedyChangeFor(int cents, const Set<int>& coins) {
    /* Base case: You need no coins to give change for no cents. */
    if (cents == 0) {
        return 0;
    }
    /* Recursive case: use the biggest coin you can and make change for what's
     * left. This uses one coin, plus whatever's needed for the remainder.
     */
    else {
        return 1 + greedyChangeFor(cents - highestCoinNotExceeding(cents, coins),
                                   coins);
    }
}
```

Now, for the non-greedy version. The idea behind this is the following: if we need to make change for zero cents, the best option is (still) to use 0 coins. Otherwise, we need to give back at least one coin. What's the first coin we should hand back? We don't know which one it is, but we can say that it's got to be one of the coins from our options and that that coin can't be worth more than the total. So we'll try each of those options in turn, see which one ends up requiring the fewest coins for the remainder, then go with that choice. The code for this is really elegant and is shown here:

```
/**
 * Given a collection of denominations and an amount to give in change, returns
 * the minimum number of coins required to make change for it.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @return The minimum number of coins needed to make change.
 */
int fewestCoinsFor(int cents, const Set<int>& coins) {
    /* Base case: You need no coins to give change for no cents. */
    if (cents == 0) {
        return 0;
    }
    /* Recursive case: try each possible coin that doesn't exceed the total as
     * as our first coin.
     */
    else {
        int bestSoFar = INT_MAX; // Sentinel, will be overwritten.
        for (int coin: coins) {
           /* If this coin doesn't exceed the total, try using it. */
           if (coin <= cents) {
               int ifUsed = fewestCoinsFor(cents - coin, coins);
               if (ifUsed < bestSoFar) {
                  bestSoFar = ifUsed;
               }
           }
        }
        return bestSoFar + 1;
    }
}
```

We asked whether memoization would be appropriate here, and the answer is "yes, definitely!" Imagine, for example, that we're using this algorithm on US coins, and we want to see the fewest number of coins required to make change for 10¢. Our options include first using a dime, first using a nickel, and first using a penny. Both of those latter two routes will eventually want to know the best way to make change for 5¢, the case where we use a nickel immediately needs to know this, and the case where we first use a penny will want to know how to do this for 9¢, which eventually needs to know 8¢, etc. down to 5¢. Without using memoization, we'd end up with a ton of redundant computation, which would slow things down dramatically. With memoization, this will be lightning fast for most numbers!

Here's what this might look like:

```
/**
 * Given a collection of denominations and an amount to give in change, returns
 * the minimum number of coins required to make change for it. This uses a table
 * to memoize its results.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @param memoizedResults A table mapping numbers of cents to the minimum number
 *                        of coins needed to make that total, for memoization.
 * @return The minimum number of coins needed to make change.
 */
int fewestCoinsForMemoized(int cents, const Set<int>& coins,
                           Map<int, int>& memoizedResults) {
    /* Base case: You need no coins to give change for no cents. */
    if (cents == 0) {
        return 0;
    }
    /* Base case: if we already know the answer, just return it! */
    else if (memoizedResults.containsKey(cents)) {
        return memoizedResults[cents];
    }
    /* Recursive case: try each possible coin that doesn't exceed the total as
     * as our first coin.
     */
    else {
        int bestSoFar = INT_MAX; // Sentinel, will be overwritten.
        for (int coin: coins) {
           /* If this coin doesn't exceed the total, try using it. */
           if (coin <= cents) {
               int ifUsed = fewestCoinsForMemoized(cents - coin, coins,
                                                   memoizedResults);
               if (ifUsed < bestSoFar) {
                  bestSoFar = ifUsed;
               }
           }
        }

        /* Store the result for later. */
        int result =  bestSoFar + 1;
        memoizedResults[cents] = result;
        return result;
    }
}

/**
 * Given a collection of denominations and an amount to give in change, returns
 * the minimum number of coins required to make change for it.
 *
 * @param cents How many cents we need to give back.
 * @param coins The set of coins we can use.
 * @return The minimum number of coins needed to make change.
 */
int fewestCoinsFor(int cents, const Set<int>& coins) {
    Map<int, int> memoizedResults;
    return fewestCoinsForMemoized(cents, coins, memoizedResults);
}
```

Problem Two: Weights and Balances

Imagine that we start off by putting the amount to be measured (call it n) on the left side of the balance. This makes the imbalance on the scale equal to n. Imagine that there is some way to measure n. If we put the weights on the scale one at a time, we can look at where we put that first weight (let's suppose it weighs w). It must either

- go on the left side, making the net imbalance on the scale $n + w$,
- go on the right side, making the net imbalance on the scale $n - w$, or
- not get used at all, leaving the net imbalance n.

If it is indeed truly possible to measure n, then one of these three options has to be the way to do it, even if we don't know which one it is. The question we then have to ask is whether it's then possible to measure the new net imbalance using the weights that remain – which we can determine recursively! On the other hand, if it's not possible to measure n, then no matter which option we choose, we'll find that there's no way to use the remaining weights to make everything balanced!

If we're proceeding recursively, which we are here, we need to think about our base case. There are many options we can choose from. One simple one is the following: imagine that we don't have any weights at all, that we're asked to see whether some weight is measurable using no weights. In what circumstances can we do that? Well, if what we're weighing has a nonzero weight, we can't possibly measure it – placing it on the scale will tip it to some side, but that doesn't tell us how much it weighs. On the other hand, if what we're weighing is completely weightless, then putting it on the scale won't cause it to tip, convincing us that, indeed, it is weightless! So as our base case, we'll say that when we're down to no remaining weights, we can measure n precisely if $n = 0$. With that in mind, here's our code:

```
/**
 * Given an amount, a list of weights, and an index, determines whether it's
 * possible to measure n using the weights at or after the index given by
 * startPoint.
 *
 * @param amount The amount to measure, which can be positive, negative or 0.
 * @param weights The weights available to us.
 * @param index The starting index into the weights Vector.
 * @return Whether the amount can be measured using the weights from the specified
 *         index and forward.
 */
bool isMeasurableRec(int amount, const Vector<int>& weights, int index) {
  if (index == weights.size()) {
      return amount == 0;
  } else {
      return isMeasurableRec(amount,                   weights, index + 1) ||
             isMeasurableRec(amount + weights[index], weights, index + 1) ||
             isMeasurableRec(amount - weights[index], weights, index + 1);
  }
}

bool isMeasurable(int amount, const Vector<int>& weights) {
    return isMeasurableRec(n, weights, 0);
}
```

Problem Three: Filling a Region

The key insight behind this problem is the following: if we're trying to fill a region black and the pixel we start from is black, then we're done – we only flood-fill if the pixel is white. Otherwise, if we're trying to fill a region and the pixel we're reading is white, then we mark it black, then start a flood fill from each of the four pixels immediately adjacent to it. This is shown here:

```
/**
 * Flood fills a region of an image with black pixels. Specifically, this
 * operation turns white pixels black, then propagates the fill to neighboring
 * pixels.
 *
 * @param image The monochrome image to update.
 * @param row The starting row
 * @param col The starting column.
 */
void floodFillFrom(Grid<bool>& image, int row, int col) {
    if (image.inBounds(row, col) && !image[row][col]) {
        image[row][col] = true;
        floodFillFrom(image, row, col + 1);
        floodFillFrom(image, row, col - 1);
        floodFillFrom(image, row + 1, col);
        floodFillFrom(image, row - 1, col);
    }
}
```

Problem Four: Weekend Hedonism

The key idea behind this problem is that for each event, you either pick it (in which case you rule out all the other events that overlap it) or you don't (in which case you're free to pick any other events). As a base case, when there aren't any events to pick from, you can have zero fun on the weekend. Awww.

Here's what this looks like in code:

```
Vector<Event> tailOf(Vector<Event> v);
bool overlapsWith(const Event& e1, const Event& e2);
Vector<Event> unconflicted(const Vector<Event>& events, const Event& event);
Vector<Event> valueOf(const Vector<Event>& events);

/**
 * Given a list of potential events to attend, returns the set of events that
 * results in the greatest overall happiness.
 *
 * @param events The master list of events.
 * @return The events you should do to maximize your happiness.
 */
Vector<Event> bestWeekendGiven(const Vector<Event>& events) {
   /* Base Case: If there are no events to attend, your only option is to do
    * nothing. :-(
    */
   if (events.isEmpty()) {
      return {}; // Empty list
   }
   /* Otherwise, see what happens if you do and do not do the first event. */
   else {
     /* If you pick the first event, you can't pick anything that conflicts with
      * it, but you get the benefit of doing that event.
      *
      * For simplicity, we're using C++'s nice auto keyword, which sets the type
      * of the variable based on the type of the thing initializing it.
      */
     auto withFirst = bestWeekendGiven(unconflicted(events, event[0]) + events[0];

     /* Or you can skip the first event, giving you the best of what's left. */
     auto withoutFirst = bestWeekendGiven(tailOf(events));

     /* Return whichever is better. */
     return valueOf(withFirst) > valueOf(withSecond)? withFirst : withSecond;
}

/**
 * Given a Vector, returns a new vector formed by dropping off the first element.
 *
 * @param v The input list.
 * @return A list of everything except the first event.
 */
Vector<Event> tailOf(Vector<Event> v) {
    v.remove(0);
    return v;
}
```

```
/**
 * Given two events, reports whether they overlap with one another.
 *
 * @param e1 The first event
 * @param e2 The second event
 * @return Whether they overlap.
 */
bool overlapsWith(const Event& e1, const Event& e2) {
   /* Two events overlap if the start time of one event is sandwiched between
    * the start and end times of the other event.
    */
   return (e1.startTime >= e2.startTime && e1.startTime <= e2.endTime) ||
          (e2.startTime >= e1.startTime && e2.startTime <= e1.endTime);
}

/**
 * Given a list of events and a potentially conflicting event, returns a Vector
 * of the events that don't conflict with the event in question.
 *
 * @param events The list of all events
 * @param event The event in question.
 * @return All events that don't conflict with it.
 */
Vector<Event> unconflicted(const Vector<Event>& events, const Event& event) {
   Vector<Event> result;
   for (const auto& thisEvent: events) { // Use auto; see below
       if (!overlapsWith(thisEvent, event)) {
          result += thisEvent;
       }
   }
   return result;
}

/**
 * Given a list of events, returns the total happiness derived from those events.
 *
 * @param events A list of events to attend.
 * @return The happiness you'll get from attending those events.
 */
Vector<Event> valueOf(const Vector<Event>& events) {
   int result = 0;
   for (const auto& event: events) { // See below for auto
      result += event.happiness;
   }
   return result;
}
```

Problem Five: Member of the Wedding

Here's one way to think about this problem. If everyone is already seated, you're done! Just score how good your seating arrangement is so far. If not, then pick someone who isn't seated and think about all the places you could seat her. She has to go somewhere, after all! Then take a look and see which of those choices ends up working out best overall and go with it! With that in mind, here's one possible solution:

```
/**
 * Given a list of guests, an index into that list, a partial seating chart,
 * and the capacity of each table, returns the best possible seating chart we
 * can make by placing the remaining people into seats.
 *
 * @param guests The guests at the wedding.
 * @param index The index of the next person to place.
 * @param seatsSoFar The partial seating chart.
 * @param tableCapacity How much seats there are at each table.
 * @return The optimal seating chart we can make by adding people in to the tables
 *         given the existing seating chart.
 */
Map<string, Vector<string>>
bestSeatingRec(const Vector<string>& guests, int index,
               const Map<string, Vector<string>>& seatsSoFar, int tableCapacity) {
    /* Base Case: If everyone's seated, whatever we have is the best option. */
    if (index == guests.size()) {
        return seatsSoFar;
    }
    /* Otherwise, figure out where we're going to put the next person. */
    else {
        Map<string, Vector<string>> bestSoFar;
        int bestScore = INT_MIN; // Sentinel; will be overwritten

        /* Try each table that still has space for this person. */
        for (string table: seatsSoFar) {
            if (seatsSoFar[table].size() < tableCapacity) {
                /* Put this person at this table. To do so, we'll make a copy of
                 * the seating chart and add this person to this table.
                 */
                auto newChart = seatsSoFar;
                newChart[table] += guests[index];

                /* See how well we can do. */
                auto thisChoice = bestSeatingRec(guests, index + 1, newChart,
                                                 tableCapacity);

                /* If this is the best we've found, remember that. */
                if (scoreFor(thisChoice) > bestScore) {
                    bestSoFar = thisChoice;
                    bestScore = scoreFor(thisChoice);
                }
            }
        }

        return bestSoFar;
    }
}
```

```
/**
 * Given a list of guests, tables, and table capacities, returns the best seating
 * arrangement for the wedding.
 *
 * @param guests The guests at the wedding.
 * @param tableNames The names of each of the tables.
 * @param tableCapacity How much seats there are at each table.
 * @return The optimal seating chart.
 */
Map<string, Vector<string>>
bestSeatingArrangementFor(const Vector<string>& guests,
                          const Vector<string>& tableNames,
                          int tableCapacity) {
    Map<string, Vector<string>> tables;
    for (string tableName: tableNames) {
        tables[tableName] = {}; // No one is initially sitting here
    }
    return bestSeatingRec(guests, 0, tables, tableCapacity);
}
```