Backtracking

```
/** = "elements of b can be placed into
 *    2 bags whose sums are equal." */
static boolean split(int[] b)
```

We write a function that determines whether the values of an **int** array can be placed into two bags whose sums are equal. For example, for b containing {3, 1, 5, 4, 1, 2} we can place them in two bags {3, 5} and {1, 4, 1, 2}, which both sum to 8. But putting the values {4, 2, 1} into two bags will always yield a bag with an even sum and a bag with an odd sum because there is only one odd value.

Now, look at the specification of split. The spec doesn't require us to construct the bags in data structures! We need *only* calculate *the sum of the values in each bag*! This will simplify everything tremendously.

Function `split` has only array `b` as parameter, so we think of writing another function `split` that will have more parameters and be recursive. We envision a recursive function in which at each depth of recursion one more value is put into one of the bags. This leads us to the specification shown (below). Values in `b`[0..`k`-1] have been placed in the two bags and their sums are in parameters `s1` and `s2`. Return true if the rest of the array values, starting at `b[k]`, can be placed in the bags so that their sums are equal; otherwise, return false.

With 4-parameter method `split` specified, we can complete the original method `split` with the body that indicates that no values have been placed in the bags, so the values in the two bags sum to 0.

```
/** The values of b[0..k-1] have been placed into two
  *  bags that sum to s1 and s2, respectively. Return true
  *  iff the rest of the values b[k..] can be placed into the
  *  two bags so that the two bags have the same sum. */
static boolean split(int[] b, int k, int s1, int s2)
```

```
/** = "elements of b can be placed into
 *    2 bags whose sums are equal." */
static boolean split(int[] b) {
   return split(b, 0, 0, 0);
}
```

At this point, stop and study the specification of the four-parameter `split` carefully. Make sure you understand it, as well as the call on it in the one-parameter function `split`.

We are now ready to write the body of 4-parameter method `split`.

The base case is easy. If `k` equals `b.length`, then all values have been placed in the bags, and true or false can be returned depending on whether the two bags have the same sum or not.

```
Base case:
  if (k == b.length) return s1 == s2;
```

For the recursive case, we have two choices: place `b[k]` in the first bag and place `b[k]` in the second bag. (These are given by the two calls given to the right.) Which one do we do? It doesn't really matter.

```
Choice 1:  split(b, k+1, s1 + b[k], s2)
Choice 2:  split(b, k+1, s1, s2 + b[k])
```

Suppose we call the first choice: put `b[k]` in `s1`. If this choice returns true, true can be returned. But if this choice returns false, the other choice must be called and its value returned.

Thus, we write a single return statement, which returns the value of the first one OR the value of the second one. Note: If the first one returns true, short circuit evaluation means that its value will be returned, without calling the second one, but if the first one returns false, the value of the second one is returned.

Isn't that a neat little algorithm?

```
/** = The values of b[0..k-1] have been placed into two
  *    bags that sum to s1 and s2, respectively. Return true
  *    iff the rest of the values b[k..] can be placed into the
  *    two bags so that the two bags have the same sum. */
static boolean split(int[] b, int k, int s1, int s2) {
   if (k == b.length) return s1 == s2;
   return split(b, k+1, s1 + b[k], s2)
          || split(b, k+1, s1, s2 + b[k]);
}
```