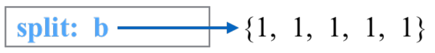
Backtracking

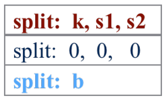
```
/** = "elements of b can be placed into
  *   2 bags whose sums are equal." */
static boolean split(int[] b) {
   return split(b, 0, 0, 0);
}
```

```
/** Values of b[0..k-1] have been placed into
   * two bags that sum to s1 and s2. Return
   * true iff b[k..] can be placed into the bags
   * so that the 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]);
}
```

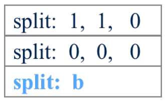
The call stack contains a frame for the call `split(b)`, with `b` containing a pointer to the array containing five 1's.



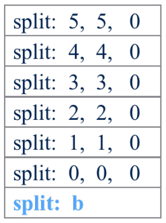
We execute the call that is in the body of `split`. A frame for the call is placed on the call stack and the argument values are stored in the parameters. In the frame for the call, we show parameters `k`, `s1`, and `s2`. We don't show `b`. It is a pointer to the array with five 1's.



Now execute the body of `split`. Start with the if-statement. Since `k` is 0 and `b.length` is 5, execution of the if-statement terminates. Next, execute the return statement. We evaluate the first call on `split`, putting a frame for the call on the call stack. In the new frame, `k` is 1, which means that `b`[0] has been placed in a bag. Parameter `s1` is 1, indicating that `b`[0] has been placed in the first bag, and the second bag is still empty.



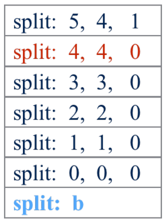
Next, execute the body of `split`. This work as before. The if-statement has no effect, and the return statement is executed, causing another frame for the call to be placed on the call stack. This time, `k` is 2 and the first bag contains two values. In fact, we can see that this will happen three more times, until `k` in the top frame is 5. All five values have been placed in the first bag!



Backtracking

At this point, `k` = 5 in the frame for the call, so `k` = `b.length`, and the if-condition of the if-statement in the body of split is true. Parameter `s1` is not equal to `s2`, so the frame for the call is popped and false is returned. This is the first example of *backtracking*; a choice was made when `k` was 4, and it didn't work.

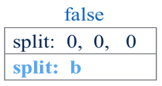
So, the second choice, putting `b`[`k`] into the second bag, is tried. We make the frame where `k` is 4 red to indicate that the second call on split in the return statement is begin executed.



Backtracking again —and again

Again, at this point `k` = `b.length`, the frame for the call is popped and the value of `s1` == `s2` is returned —it is again false. Backtracking occurs in order to attempt a different choice. But this time, it is the second call on split that returned false, so the frame for the call is popped and the value of `s1` == `s2` is returned —again it is false. More backtracking.

Eventually, the call `split`(`b`, 1, 1, 0) will return false.



This will cause the second call on `split` to be executed, with the first 1 being put in the second bag. Now, a lot of calls and backtracking will take place before the call `split`(`b`, 0, 0, 0) returns false.

Worst-case number of calls on split

We modified function `split` to also calculate how many calls on `split` are made. See our revised method in the demo code that accompanies entry backtracking in JavaHyperText. When the values in array `b` of size `n` cannot be split into two bags so that their sums are the same, 2^(`n`+1) calls are made! Thus, split take time O(2^`n`) in the worst case.

We could have predicted this. Each value can be placed in bag 1 or bag 2. There are 2^`n` sequences 1's and 2's of length `n`, and each has to be tested by split.