Interview Question: The Two-Sum Problem

This is a classic algorithmic interview question. There are many different solution routes, each of which involves a different technique. This handout details the problem and gives a few different solution routes.

Problem Statement:

You are given an array of n integers and a number k. Determine whether there is a pair of elements in the array that sums to exactly k. For example, given the array [1, 3, 7] and $k = 8$, the answer is "yes," but given $k = 6$ the answer is "no."

Possible Follow-Up Questions:

- Can you modify the array? **Yes, that's fine**.
- Do we know something about the range of the numbers in the array? **No, they can be arbitrary integers**.
- Are the array elements necessarily positive? **No, they can be positive, negative, or zero**.
- Do we know anything about the value of k relative to n or the numbers in the array? **No, it can be arbitrary**.
- Can we consider pairs of an element and itself? **No, the pair should consist of two different array elements.**
- Can the array contain duplicates? **Sure, that's a possibility**.
- Is the array necessarily in sorted order? **No, that's not guaranteed.**
- What about integer overflow? **For simplicity, don't worry about this**.

What to Watch For

- Make sure their solution works on arrays of zero or one element (it should always return false.)
- Make sure their solution always returns a value.
- Make sure their solution works with both positive and negative numbers.
- Make sure their solution works if the only way to make the sum is to use an element that appears twice in the array (for example, [3, 3] with k=6)
- Make sure their solution works if the answer is "no" but $k/2$ is present in the array (for example, [1, 3] with $k = 6$) should return false.

Possible Solution Routes

Option 1: Brute force

One option is to just try all the pairs in the array and see if any of them add up to the number k. Since there are $\Theta(n^2)$ possible pairs, this takes $O(n^2)$ time in the worst-case. This solution uses only O(1) space, since no auxiliary structures are created. This is not considered a "good" solution because better options exist, but it's a correct solution.

Sample solution:

```
private boolean sumsToTarget(int[] arr, int k) {
      for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                  if (arr[i] + arr[j] == k) {
                        return true;
                  }
            }
      }
      return false;
}
```

Option 2: Hashing

Another option is to create a hash table of all the elements in the array. You can then scan over the array and check, for each element A[i], whether there's another element A[j] in the array where A[j] = k – A[i]. Make sure their solution correctly handles the case where there are duplicated elements in the array and doesn't accidentally let you pair an element with itself.

This solution runs in expected time $O(n)$ because n insertions and n lookups in a hash table takes expected time $O(n)$. It uses space $O(n)$. This is considered a "good" solution – there is no clear way to improve on both the time complexity or the space complexity simultaneously.

Sample solution:

```
private boolean sumsToTarget(int[] arr, int k) {
      HashSet<Integer> values = new HashSet<Integer>();
      for (int i = 0; i < arr.length; i++) {
            if (values.contains(k - A[i])) return true;
            values.add(A[i]);
      }
      return false;
}
```

Option 3: Sorting and Binary Search

A third option is to sort the array and use binary search. This is conceptually similar to the hashing approach except that instead of using a hash table, we sort the array elements and use binary search to test if a pair appears.

This solution runs in time O(n log n) because it takes O(n log n) time to sort the array using a standard sort and the cost of n binary searches is O(n log n). The space usage depends on the particular sorting algorithm used – quicksort will take O(log n) space, heap sort uses O(1) space. This is considered a "pretty good solution" because it can be sped up a bit in practice using a better sorting algorithm and a different observation about the sorted array (details later).

Sample solution:

```
private boolean sumsToTarget(int[] arr, int k) {
      Arrays.sort(arr);
      for (int i = 0; i < arr.length; i++) {
            int siblingIndex = Arrays.binarySearch(arr, k - A[i]);
            if (siblingIndex >= 0) { // Found it!
                  /* If this points at us, then the pair exists only if
                   * there is another copy of the element. Look ahead of
                   * us and behind us.
                   */
                  if (siblingIndex != i ||
                      (i > 0 && arr[i-1] == arr[i]) ||
                      (i < arr.length - 1 && arr[i+1] == arr[i])) {
                        return true;
                  }
            }
      }
      return false;
}
```

Option 4: Sorting and Walking Inward

A fourth option is to sort the array, then walk two pointers inward from the ends of the array, at each point looking at their sum. If it's exactly k, then we're done. If it exceeds k, then any sum using the larger element is too large, so we walk that pointer inwards. If it's less than k, then any sum using the lower element is too small, so we walk that pointer inwards.

The runtime of this algorithm depends on the sorting algorithm used. Using a standard sorting algorithm, this takes time O(n log n) due to the cost of sorting. Using something like radix sort, this takes time O(n log U), where U is the largest element of the array, because of the cost of the sort. This will take space O(log n) if you use something like quicksort or radix sort and O(1) if you use heapsort. This is a "good solution" because in practice it's faster than the sort-and-binary-search approach using a regular sort and is likely to be faster asymptotically if you use radix sort.

Sample solution:

```
private boolean sumsToTarget(int[] arr, int k) {
      Arrays.sort(arr);
      int lhs = 0, rhs = arr.length - 1;
      while (lhs < rhs) {
            int sum = arr[lhs] + arr[rhs];
            if      (sum == k) return true;
            else if (sum <  k) lhs++;
            else               rhs--;
      }
      return false;
}
```