Solutions to Group A Problems

1. You have an array of n integers. Find the largest range $[a, b]$ where every integer in the range appears at least once in the array.

Brute-force: Take the minimum and maximum value in the array; call them *min* and *max*. The largest possible range must be a subrange of [*min*, *max*]. For each possible subrange, check whether all numbers in the range are present somewhere in the array.

- **Time complexity:** We'll need to introduce a new variable (let's call it R) representing the range of values present in the array (that is, $max - min$). It takes time $O(n)$ to find the minimum and maximum values in the array. The number of possible subranges is $O(R^2)$, since there are R possible values for a start and end of the range, and testing each value in a particular range takes time $O(nR)$, since there are $O(R)$ possible values to test and each time you may have to scan over n array elements. This gives a total time complexity of $O(nR^3)$, which is not particularly good.
- **Space complexity:** This approach is very space-efficient, however. We only need O(1) space to store the min and max values and O(1) space to store the current range, so the space complexity is O(1).

Sorting: Sort the array. The problem now becomes the following: what is the longest range of consecutive values in the array, ignoring duplicates? This can be done by starting a pointer at the start of the array and continuously advancing it forward until it either hits the end of the array or reaches a value in the array that is at least 2 greater than the previous element. (Note that you don't want to stop if you find two consecutive equal values, since duplicates don't matter.) Record the size and endpoints of the largest array found this way to determine the largest range.

- **Time complexity:** Sorting the array can be done in time $O(n \log n)$ using any standard efficient sorting algorithm (quicksort, heapsort, mergesort, etc.) Afterwards, the scan over the array takes linear time. This gives a total time complexity of $O(n \log n)$.
- **Space complexity:** The linear scan requires only O(1) space. Sorting may take either O(1) space (heapsort), $O(\log n)$ space (quicksort on expectation), or $O(n)$ (mergesort).

Hashing: Create a hash table of all the values in the array. The goal will now be, for each entry in the array, to determine the length of the longest range it belongs to. For each array element x, determine whether $x - 1$ is in the hash table. If not, x is the lower end of a long range. Check whether $x + 1$, $x + 2$, $x + 3$, etc. are in the hash table until one of these values isn't present in the table. The last number in the hash table gives the upper end of the range starting at x. The largest range found this way is the answer.

- **Time complexity:** Building a hash table with n elements takes (expected) $O(n)$ time. The next step then takes time $O(n)$ for the following reason: for each element, we do O(1) work checking whether it's the lower end of a range. If it isn't, we're done. Otherwise, we count upward to the end of the range. This process visits each element in the hash table exactly once because each range is scanned only once, and it takes O(1) time to scan each element because hash table lookups run in time O(1). The total time is thus $O(n)$, on expectation.
- **Space complexity:** We need to maintain a hash table with n elements, which requires $O(n)$ space.

2. You are given an array that is the rotation of a sorted array (that is, it was formed by taking a sorted array and moving some number of elements from the end of the array to the front). For example, the array [3, 4, 6, 9, 0, 1] is a rotated array. Given some value k, determine whether k is contained in the array.

Brute-force: Do a linear scan over the elements of the array. This takes time O(n) and uses O(1) space, but isn't very efficient.

As a note: if the array can contain duplicates, it's impossible to do better than this algorithm in the worst-case. After all, you could get the array [1, 1, 1, …, 1, 0, 1, 1, 1, …, 1] for any length n and with the 0 positioned arbitrarily. There would be no way to tell where the 0 was unless you visited every possible element of the array to rule out the possibility that it was there.

Modified binary search: If no duplicates are allowed, the following algorithm is a variation on binary search. If the array has two or fewer elements, do a linear search to see if the element is present. Otherwise, look at the first and last elements of the array. If the first element is less than the last element, then the array in question is sorted and you can do a standard binary search for k.

In the interesting case, the first element of the array is greater than the last element of the array. This means that the array has been rotated by some number of steps. Call the part of the array from the beginning of the array to the rotation point the "large" part of the array and the other part the "small" part. Now compare the middle element to the first element. If the middle element greater than the first element, then the middle element belongs to the "large" part of the array. If not, belongs to the "small" part. Then:

- If the middle element belongs to the large part and the value of k is between the first and middle elements of the array, then if k exists in the array, it lies in the first half of the array, which happens to be a sorted range. Do a standard binary search for k in the first half of the array.
- If the middle element belongs to the large part and the value of k is not between the first and middle elements of the array, then if k exists, it's either in the tail end of the large part of the array or it's in the small part (which happens to be the second half of the array). Recursively apply this algorithm to the second half of the array.
- If the middle element belongs to the small part and the value of k is between the the middle and end of the array, then if k exists it's in the second half of the array, which is sorted. Do a standard binary search for k in the first half of the array.
- If the middle element belongs to the large part and k the value of k isn't between the middle and end of the array, then k either belongs to the part of the small array before the middle element or to the large part of the array, which happens to be the elements in the first half of the array. Recursively apply this algorithm to the first half of the array.

Each iteration of the algorithm takes time O(1) and throws away half of the array at each step, so the runtime is O(log n). The space complexity is only O(1) because we can just store the endpoints of the subarray that we're currently considering.

3. You are given an array of n numbers. Given a number k, determine whether the array contains two numbers whose sum is exactly k.

Brute-force: Check each pair of elements in the array. If any two of them sum up to k, output yes. Otherwise, output no.

- **Time complexity:** There are O(n^2) pairs to check and each can be checked in time O(1). The total time complexity is therefore O(n^2).
- **Space complexity:** This only uses O(1) space, since we just need to store the indices that we're checking.

Sorting: Sort the array into ascending order. There are then two possible solutions to consider:

- For each element x of the array, do a binary search to see if $x - k$ exists in the array. If so, output yes. If no element's complement is found, output no. This step will take time O(n log n) since it does n binary searches and will use only O(1) space.
- Maintain two pointers to the first and last elements of the array. Sum those values together. If the sum is greater than k, move the right pointer one step to the left. If the sum is less than k, move the left pointer one step to the right. If the sum is exactly k, output yes. If at any point the pointers meet, output no. This step takes O(n) time and uses O(1) space.

The total time complexity, using either approach, is O(n log n) due to the sorting step and uses space proportional to the space used by the sorting algorithm (either O(1) for heapsort, O(log n) for quicksort, or O(n) for mergesort).

Hashing: Scan over the elements of the array from left to right. For each element x found this way, see if $x - k$ is in the hash table. If so, output yes. Otherwise, add x to the hash table and go to the next element. If you reach the end of the array without finding a pair that works, output no.

- **Time complexity:** This does O(1) work per array element and visits each array element exactly once for a total time complexity of O(n).
- **Space complexity:** O(n) space is required to store the hash table.

4. You are given a list of n closed intervals of real numbers. Find the smallest nonnegative integer that does not belong to any of those intervals.

Brute-force: Starting at 0 and counting upward, check each integer to see if it's not contained in any of the intervals. As soon as a value is found that's not contained in any of the intervals, output it.

- **Time complexity:** Let R be the value found this way. There are then O(R) iterations of doing O(n) work to test each interval, for a total time complexity of O(nR).
- **Space complexity:** Only O(1) space is needed.

Sorting: Sort the intervals from left to right by their left endpoint, breaking ties arbitrarily. Let the candidate answer be 0. Starting from the leftmost interval and moving to the right, do the following:

- If the candidate value is less than the current interval's left endpoint, output the candidate value.
- If the candidate value is greater than or equal to the current interval's left endpoint, set the candidate value to be the maximum of the current candidate value and the smallest integer greater than the current interval's right endpoint.

If this process doesn't terminate before all intervals are processed, output the candidate value as the result.

- **Time complexity:** Sorting the intervals takes time O(n log n) and the scanning step then only takes O(n) extra time, since we do O(1) work for up to O(n) intervals. The total time complexity is therefore O(n).
- **Space complexity:** Depending on the sorting algorithm used, the space complexity will either be O(1) (heapsort), O(log n) (quicksort), or O(n) (heapsort).

5. The Fibonacci sequence is the sequence that starts with 0, 1 and where each successive term is the sum of the two previous terms. The Fibonacci numbers start off 0, 1, 1, 2, 3, 5, 8, 13, 21, … . You are given a list of n positive integers. Determine, for each number in the list, whether it's a Fibonacci number. When analyzing your algorithm, you should come up with a runtime in terms of n, the number of integers given, and U, the largest number given in the array.

Brute force: Check, for each number in the array, whether it's a Fibonacci number by computing terms of the Fibonacci sequence until either (a) the value is generated or (b) a larger Fibonacci number is produced.

- **Time complexity:** The Fibonacci numbers grow exponentially quickly, so for each number in the array this will take time O(log U). Since there are n numbers total, this will take time O(n log U).
- **Space complexity:** Outputting the answer for all n values will take O(n) space just to store the result. Aside from that, only O(1) auxiliary storage is required.

Hashing: Find the largest value in the array. Compute all Fibonacci numbers up to and including that value and put them into a hash table. Then, for each element of the array, if the number is in the hash table, output that it's a Fibonacci number. Otherwise, output that it's not a Fibonacci number.

- **Time complexity:** It takes time O(n) to find the largest value and O(log U) time to then compute the Fibonacci numbers up to that value. From there, we spend O(n) more time looking up values in the hash table. The total time complexity is therefore O(n + log U).
- **Space complexity:** Outputting the answer for all n values will take O(n) time just to store the result. Aside from that, O(log U) space is required to store the Fibonacci numbers in the hash table.