Solutions to Group B Problems

1. You are given an array of n values. Determine which value appears most frequently in the array. If there is a tie, you can return any of the most-frequently-occurring values.

Brute force: For each array element, count how many times that array element appears in the array and store the most-frequent element found at each point. At the end, return that element.

- **Time complexity:** There are O(n) array elements to check and for each we'll spend O(n) time counting all the other elements of the array. This requires a total of O(n^2) time.
- **Space complexity:** Only O(1) space is required (temporary variables, the loop counter, and the current guess.)

Sorting: Sort the array, placing all copies of duplicate elements in the array next to one another. This problem then boils down to finding the longest range of equal values in the array, which can be done by scanning the array from left to right, at each point checking whether the current element is equal to the previous element (in which case, the current range extends for another step) or whether it is different (in which case the previous range ends and a new range starts). When doing this, don't forget that a range might end at the end of the array and so the end-of-array case needs to be handled specially.

- **Time complexity:** Sorting the array takes time O(n log n). Finding the longest range of equal values then takes time O(n) for a total time of O(n log n).
- **Space complexity:** Sorting the array takes space either O(1) (heapsort), O(log n) (quicksort on expectation) or O(n) (mergesort), depending on what algorithm is used.

Hashing: Build a hash table mapping elements to their frequencies. Scan over the array and for each element, if it's not in the hash table, add it to the hash table with frequency one. Otherwise, increment the counter stored in the hash table. While doing this, track the highest encountered frequency and the most frequency value found so far. At the end, output that value.

- **Time complexity:** This algorithm requires O(n) hash table reads and writes, so it will take O(n) on expectation.
- **Space complexity:** Storing elements and counters in the hash table requires O(n) total space.

2. You are given a *unimodal array* of real numbers. A unimodal array is one that consists of a strictly increasing sequence followed by a strictly decreasing sequence. For example, the array [1, 3, 7, 15, 29, 23, 14, 13, 12] is unimodal. Find the maximum value in the array.

Brute-force: Do a linear scan over the array to find the maximum value. This takes time O(n) and requires space O(1).

Modified binary search: If the array has only one element, that element must be the maximum element, Otherwise, look at the middle element and the element immediately after it. There are three cases to consider:

- If the first middle element is smaller than the second, then those elements are part of the increasing portion of the array. The maximum thus appears after the first middle element, so discard all elements up to and including the first middle element and repeat.
- If the first middle element is greater than the second, then those elements are part of the decreasing portion of the array. The maximum thus appears before the second middle element, so discard all elements including and after the second middle element and repeat.
- If the first middle element is equal to the second, then the first middle element must be the end of the increasing sequence and the second middle element must be the start of the decreasing sequence. Therefore, both elements have value equal to the maximum, so return that element.

Since the array size drops by roughly one half each time, this process eventually terminates.

- **Time complexity:** Each iteration of the algorithm does O(1) work and then discards half of the array. This means that there are O(log n) iterations, each doing O(1) work, so the total work done is O(log n).
- **Space complexity:** We can track the extent of the current subarray by using two pointers to indicate the leftmost and rightmost elements under consideration. Other than that, only O(1) temporary variables are needed, so the space usage is O(1).

3. Given two strings, determine whether those strings are anagrams of one another.

Sorting: Sort the two strings and check whether they compare equal to one another. Two strings are anagrams if and only if they have the same number of copies of each character, so two strings are anagrams of one another if and only if they are identical when sorted.

- **Time complexity:** Sorting the characters using a standard sorting algorithm will take time O(n log n), and checking whether the strings are equal will, from that point, take time O(n). This route gives a time complexity of O(n log n). However, if we assume that there are only finitely many possible characters, we can sort the strings either using counting sort or radix sort in time O(n), which would make the algorithm have time complexity O(n).
- **Space complexity:** Using an O(n log n) sort, the space complexity will be either O(1), O(log n), or O(n) depending on the sorting algorithm. Using radix sort, the space complexity would be either O(1) or O(n) depending on whether it's an MSD or LSD radix sort. Using counting sort, the space complexity will be O(n).

Hashing: Create a frequency table mapping characters to the number of times they appear in each string. Iterate over each string to populate the tables, then iterate over the tables to confirm that they're identical. The frequency table could be implemented using a hash table or as a standard array if the number of possible characters is known to be small.

- **Time complexity:** Building each frequency table will take time O(n) and comparing the frequency tables against one another will take time O(n) as well. This gives a total runtime of O(n).
- **Space complexity:** Each frequency table will require O(n) space, so O(n) space is required.

4. You are given a list of n closed intervals whose endpoints represent times in a day. Some of these intervals might overlap one another. We'll say that a set of intervals is *overlapping* if all of the intervals in the range overlap one another. For example, the set {[0, 4], [1, 3], [2, 8]} is overlapping, while the set {[0, 2], [1, 3], [2, 4], [3, 5]} is not (because [0, 2] and [3, 5] don't overlap). Find the maximum number of intervals from the list that overlap.

Brute-force: List off all subsets of the intervals and, for each set, check whether all the intervals in it are overlapping. Then find which set has the largest number of overlapping intervals and output the number of elements in that set.

- **Time complexity:** There are 2^n possible subsets of intervals and each set can be listed in time $O(n)$. It then takes time $O(n^2)$ using a naïve approach to check if the intervals in a set overlap, giving a total time complexity of $O(n^3 2^n)$.
- **Space complexity:** Only $O(n)$ space is actually required to store each set and that space can be reused from set to set, so only $O(n)$ space is required.

Sorting: For each interval $[a, b]$, create two "events:" an "open" event at time a and a "close" event at time b. Sort these events into ascending order of time, breaking ties by ordering "open" events ahead of "close" events. Scanning from left to right over these events then gives a "time-line" of when different intervals open and close. Start with a counter set to 0 and scan over the events from left to right. Whenever an "open" event occurs, increment the counter. Whenever a "close" event occurs, decrement the counter. The maximum value attained by this counter is then the maximum number of intervals that overlap at any time.

- **Time complexity:** Creating the intervals takes time $O(n)$ and the scan at the end takes time $O(n)$ as well. Sorting the events takes time $O(n \log n)$. Thus the total time complexity is $O(n \log n)$.
- **Space complexity:** Storing the events takes space $O(n)$, which dominates the space used by any of the standard sorting algorithms. Thus only $O(n)$ space is needed.

5. You are given a list of positive integers. Determine the smallest positive integer that cannot be written as a sum of some of those integers. For example, given the list [4, 8, 1, 2], the answer is 16. Given the list [1, 2, 5, 8], the answer is 4.

Brute-force: List off all subsets of the list and compute their sum, storing the results in a set. Then, count from 1 upward until a number is found that is not contained in the set and output that number.

- **Time complexity:** There are 2^n subsets of the list to consider, so there may be up to 2^n different sums possible. It's possible to generate each of the possible sums in time $O(2^n)$ total. Assuming the result is stored as either a bitvector or in a hash table, these sums can be stored in time $O(2^n)$. If R is the number returned, the total runtime will then be $O(2^n + R)$.
- **Space complexity**: Storing all 2^n possible sums uses up to $O(2^n)$ space.

Sorting: Sort the numbers into ascending order. Keep track of a variable *upTo* that stores the largest value of k for which all numbers 0, 1, 2, …, k are known to be possibly made as the sum of some numbers from the list. Initially, *upTo* is zero. Then, scan across the array from the left to the right. When encountering number x, compare *upTo* and x. If $x > upTo + 1$, then there is no possible way to make the number $upTo + 1$ using the numbers from the list, so the answer is $upTo + 1$. Otherwise, we know $x \leq upTo + 1$. Therefore, all the numbers 0, 1, 2, …, *upTo* can be formed as before, and the numbers x, $x + 1$, $x + 2$, $x + 3$, …, $x + upTo$ can be formed by forming one of the numbers 0, 1, 2, …, *upTo* and then adding in x. This means that all nonnegative integers between 0 and $upTo + x$ can be formed using the current set of array elements, so set *upTo* to $upTo + x$. If all array elements are processed without a missing value being found, return $upTo + 1$.

- **Time complexity:** Sorting the array takes time $O(n \log n)$ using any standard sorting algorithm. Once the array is sorted, we do O(1) work per element for each of the n array elements, so the runtime is $O(n \log n)$.
- **Space complexity:** Sorting the array takes space O(1), $O(\log n)$, or $O(n)$ depending on which algorithm is used. Other than that, only O(1) space is needed, so the total space complexity depends purely on the sorting algorithm chosen.