Chapter 10 - Sorting and Searching

// Common Sorting Algorithms

- Bubble sort → runtime O(n^2), memory O(1)
- Selection sort → runtime O(n^2), memory O(1)
- Merge sort → runtime O(n * log n), memory depends → split array in half, sort both halves and merge the result → at merge copy elements from both arrays in a temporary one at each index grabbing the smallest element first → memory is O(n)
- Quick Sort → runtime O(n * log n) average and O(n ^ 2) → memory is O(log n)
- Radix Sort → runtime O(k * n)
- Searching Algorithms → binary search

// **Sorted Merge** → insert largest elements to the back of the array

```
void merge(int[] a, int[] b, int lastA, int lastB) {
	int indexA = lastA - 1;
	int indexB = lastB - 1;

	int indexMerged = lastA + lastB - 1;

	while (indexB >= 0) {
		if (indexA >= 0 && a[indexA] > b[indexB]) {
			a[indexMerged] = a[indexA];
			indexA--;
		} else {
			a[indexMerged] = b[indexB];
			indexB--;
		}
		indexMerged--;
	}
}
```

// **Group Anagrams**

```
public static void sort(String[] array) {
	HashMapList<String, String> mapList = new HashMapList<String, String>();

	/* Group words by anagram */
	for (String s : array) {
		String key = sortChars(s);
```

```
            mapList.put(key, s);
        }

        /* Convert hash table to array */
        int index = 0;
        for (String key : mapList.keySet()) {
            ArrayList<String> list = mapList.get(key);
            for (String t : list) {
                array[index] = t;
                index++;
            }
        }
}

public static String sortChars(String s) {
        char[] content = s.toCharArray();
        Arrays.sort(content);
        return new String(content);
}
```

// **Search in rotated array →**

```
int search(int a[], int left, int right, int x) {
        int mid = (left + right) / 2;

        if (x == a[mid]) {
            return mid;
        }

        if (right < left) {
            return -1;
        }

        if (a[left] < a[mid]) {  // Left is normally ordered
            if (x >= a[left] && x < a[mid]) {
                return search(a, left, mid - 1, x);
            } else {
                return search(a, mid + 1, right, x);
            }
        } else if (a[mid] < a[left]) {  // Right is normally ordered
            if (x > a[mid] && x <= a[right]) {
                return search(a, mid +1, right, x);
            } else {
```

```
                return search(a, left, mid - 1, x);
            }
        } else if (a[left] == a[mid]) {  // Left or right half is all repeats
            if (a[mid] != a[right]) {
                return search(a, mid + 1, right, x);
            } else {  // We have to search both halves
                int result = search(a, left, mid - 1, x);
                if (result == - 1) {
                    return search(a, mid + 1, right, x);
                } else {
                    return result;
                }
            }
        }
        return -1;
}
```

// **Sorted Search**

```
public class Question {

    int search(Listy list, int value) {
        int index = 1;
        while (list.elementAt(index) != -1 && list.elementAt(index) < value) {
            index *= 2;
        }
        return binarySearch(list, value, index / 2, index);
    }

    int binarySearch(Listy list, int value, int low, int high) {
        int mid;

        while (low <= high) {
            mid = low + (high - low) / 2;
            int middle = list.elementAt(mid);
            if (middle > value || middle == -1) {
                high = mid - 1;
            } else if (middle < value) {
                low = mid + 1;
            } else {
                return mid;
            }
        }
```

```
        return -1;
    }
}
```

// **Sparse Search**

```
public class Question {

    int search(String[] strings, String str, int first, int last) {
        if (first > last) {
            return -1;
        }
        int mid = (last + first) / 2;

        if (strings[mid].isEmpty()) {
            int left = mid - 1;
            int right = mid + 1;

            while (true) {
                if (left < first && right > last) {
                    return -1;
                } else if (right <= last && !strings[right].isEmpty()) {
                    mid = right;
                    break;
                } else if (left >= first && !strings[left].isEmpty()) {
                    mid = left;
                    break;
                }
                right++;
                left--;
            }
        }

        if (str.equals(strings[mid])) {
            return mid;
        } else if (strings[mid].compareTo(str) < 0) {
            return search(strings, str, mid + 1, last);
        } else {
            return search(strings, str, first, mid - 1);
        }
    }
}
```

// **Missing int**

```
 public static int findOpenNumber(String filename) throws FileNotFoundException {
		int rangeSize = (1 << 20); // 2^20 bits (2^17 bytes)

		/* Get count of number of values within each block. */
		int[] blocks = getCountPerBlock(filename, rangeSize);

		/* Find a block with a missing value. */
		int blockIndex = findBlockWithMissing(blocks, rangeSize);
		if (blockIndex < 0) return -1;

		/* Create bit vector for items within this range. */
		byte[] bitVector = getBitVectorForRange(filename, blockIndex, rangeSize);

		/* Find a zero in the bit vector */
		int offset = findZero(bitVector);
		if (offset < 0) return -1;

		/* Compute missing value. */
		return blockIndex * rangeSize + offset;
	}

	/* Get count of items within each range. */
	public static int[] getCountPerBlock(String filename, int rangeSize) throws
FileNotFoundException {
		int arraySize = Integer.MAX_VALUE / rangeSize + 1;
		int[] blocks = new int[arraySize];

		Scanner in = new Scanner (new FileReader(filename));
		while (in.hasNextInt()) {
		   int value = in.nextInt();
		   blocks[value / rangeSize]++;
		}
		in.close();
		return blocks;
	}

	/* Find a block whose count is low. */
	public static int findBlockWithMissing(int[] blocks, int rangeSize) {
		for (int i = 0; i < blocks.length; i++) {
			if (blocks[i] < rangeSize){
				return i;
```

```
			}
		}
		return -1;
	}

	/* Create a bit vector for the values within a specific range. */
	public static byte[] getBitVectorForRange(String filename, int blockIndex, int rangeSize)
throws FileNotFoundException {
		int startRange = blockIndex * rangeSize;
		int endRange = startRange + rangeSize;
		byte[] bitVector = new byte[rangeSize/Byte.SIZE];

		Scanner in = new Scanner(new FileReader(filename));
		while (in.hasNextInt()) {
			int value = in.nextInt();
			/* If the number is inside the block that's missing
			 * numbers, we record it */
			if (startRange <= value && value < endRange) {
				int offset = value - startRange;
				int mask = (1 << (offset % Byte.SIZE));
				bitVector[offset / Byte.SIZE] |= mask;
			}
		}
		in.close();
		return bitVector;
	}

	/* Find bit index that is 0 within byte. */
	public static int findZero(byte b) {
		for (int i = 0; i < Byte.SIZE; i++) {
			int mask = 1 << i;
			if ((b & mask) == 0) {
				return i;
			}
		}
		return -1;
	}

	/* Find a zero within the bit vector and return the index. */
	public static int findZero(byte[] bitVector) {
		for (int i = 0; i < bitVector.length; i++) {
			if (bitVector[i] != ~0) { // If not all 1s
				int bitIndex = findZero(bitVector[i]);
```

```
                return i * Byte.SIZE + bitIndex;
            }
        }
        return -1;
    }
```

// **Find duplicates**

4 kb memory → 8 * 4 * 1024 > 32000 → we can create a bit vector with 32000 bits where each bit represents one integer

```
void duplicates(int[] array) {
    BitSet bs = new BitSet(32000);
    for (int i = 0; i < array.length; i++) {
        int num = array[i];
        int num0 = num - 1; // bitset starts at 0, numbers start at 1
        if (bs.get(num0)) {
            System.out.println(num);
        } else {
            bs.set(num0);
        }
    }
}

class BitSet {
    int[] bitset;

    public BitSet(int size) {
        bitset = new int[(size >> 5) + 1];
    }

    boolean get(int pos) {
        int wordNumber = (pos >> 5);  // divide by 32
        int bitNumber = (pos & 0x1F);  // x mod y = x & (y - 1) if y is a power of 2
        return (bitset[wordNumber] & (1 << bitNumber)) != 0;
    }

    void set(int pos) {
        int wordNumber = (pos >> 5);  // divide by 32
        int bitNumber = (pos & 0x1F);  // x mod y = x & (y - 1) if y is a power of 2
        bitset[wordNumber] |= 1 << bitNumber;
    }
}
```

// **Sorted Matrix Search** → M x N matrix, each row and each column is sorted in ascending order, write a method to find an element

// Solution A → we can perform a binary search on each row → O (M * logN)

```
boolean findElement(int[][] matrix, int element) {
	int row = 0;
	int col = matrix[0].length - 1;

	while (row < matrix.length && col >= 0) {
		if (matrix[row][col] == element) {
			return true;
		} else if (matrix[row][col] > element) {
			col --;
		} else {
			row++;
		}
	}
	return false;
}
```

// Solution B → Binary Search

```
public class Coordinate implements Cloneable {

	public int row, column;

	public Coordinate(int r, int c) {
		row = r;
		column = c;
	}

	public boolean inBounds(int[][] matrix) {
		return row >= 0 && column >=0 &&
			row < matrix.length && column < matrix[0].length;
	}

	public boolean isBefore(Coordinate p) {
		return row <= p.row && column <= p.column;
	}

	public Object clone() {
		return new Coordinate(row, column);
```

```
	}

	public void setToAverage(Coordinate min, Coordinate max) {
		row = (min.row + max.row) / 2;
		column = (min.column + max.column) / 2;
	}
}

public static Coordinate partitionAndSearch(int[][] matrix, Coordinate origin, Coordinate dest,
Coordinate pivot, int x) {
		Coordinate lowerLeftOrigin = new Coordinate(pivot.row, origin.column);
		Coordinate lowerLeftDest = new Coordinate(dest.row, pivot.column - 1);
		Coordinate upperRightOrigin = new Coordinate(origin.row, pivot.column);
		Coordinate upperRightDest = new Coordinate(pivot.row - 1, dest.column);

		Coordinate lowerLeft = findElement(matrix, lowerLeftOrigin, lowerLeftDest, x);
		if (lowerLeft == null) {
			return findElement(matrix, upperRightOrigin, upperRightDest, x);
		}
		return lowerLeft;
	}

	public static Coordinate findElement(int[][] matrix, Coordinate origin, Coordinate dest, int
x) {
		if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
			return null;
		}
		if (matrix[origin.row][origin.column] == x) {
			return origin;
		} else if (!origin.isBefore(dest)) {
			return null;
		}

		/* Set start to start of diagonal and end to the end of the diagonal. Since
		 * the grid may not be square, the end of the diagonal may not equal dest.
		 */
		Coordinate start = (Coordinate) origin.clone();
		int diagDist = Math.min(dest.row - origin.row, dest.column - origin.column);
		Coordinate end = new Coordinate(start.row + diagDist, start.column + diagDist);
		Coordinate p = new Coordinate(0, 0);

		/* Do binary search on the diagonal, looking for the first element greater than x */
		while (start.isBefore(end)) {
```

```
                p.setToAverage(start, end);
                if (x > matrix[p.row][p.column]) {
                    start.row = p.row + 1;
                    start.column = p.column + 1;
                } else {
                    end.row = p.row - 1;
                    end.column = p.column - 1;
                }
            }

            /* Split the grid into quadrants. Search the bottom left and the top right. */
            return partitionAndSearch(matrix, origin, dest, start, x);
        }

        public static Coordinate findElement(int[][] matrix, int x) {
            Coordinate origin = new Coordinate(0, 0);
            Coordinate dest = new Coordinate(matrix.length - 1, matrix[0].length - 1);
            return findElement(matrix, origin, dest, x);
        }
```

// **Rank from stream**

implement BST with rank and put operations

// **Peaks and valleys**

```
public static void swap(int[] array, int left, int right) {
    int temp = array[left];
    array[left] = array[right];
    array[right] = temp;
}

public static void sortValleyPeak(int[] array) {
    for (int i = 1; i < array.length; i += 2) {
        if (array[i - 1] < array[i]) {
            swap(array, i - 1, i);
        }
        if (i + 1 < array.length && array[i + 1] < array[i]) {
            swap(array, i + 1, i);
        }
    }
}
```