Chapter 8 - Recursion and Dynamic Programming

// **How to approach?**

- Recursive solutions are built off of solutions to subproblems

// **Bottom-Up Approach** → start by solving the problem for a simple case (eg. list of one element) → then we figure out how to solve it for two elements and so on → built solution on top of previous cases

// **Top-Down Approach** → think about how we can divide the problem for case N into subproblems

// **Half-and-Half Approach** → divide the data set in half (eg. binary search, merge sort)

// **Recursive vs. Iterative solutions** → recursive algorithms can be very space inefficient → if algorthm recurses to a depth of n → it uses at least O(n) memory

// **Dynamic Programming & Memoization** → finding the overlapping subproblems (repeated calls) then caching those resutls for future repeated calls

// Fibonacci Numbers → compute the nth Fibonacci number

```
int fibonacci(int i) {
	if (i == 0) {
		return 0;
	}
	if (i == 1) {
		return 1;
	}
	return fibonacci(i - 1) + fibonacci(i - 2);
}
```

// The number of nodes in the tree will represent the runtime → each node has 2 children → if we do this n times → O(2 ^ n) (in reality it's O(1.6 ^ n) since the right subtree is smaller)

// **Top-Down Dynamic Programming (or Memoization)** → we simply cache the results of fibonacci(i) between calls

```
int fiboncacci(int n) {
	return fibonacci(n, new int[n + 1]);
}

int fibonacci(int i, int[] memo) {
	if (i == 0 || i == 1) {
		return i;
	}
	if (memo[i] == 0) {
		memo[i] = fibonacci(i - 1, memo) + fibonacci(i - 2, memo);
	}
	return memo[i];
}
```

// The tree now jusy shoots straight down → 2n nodes → O(n) runtime

// **Bottom-Up Dynamic Programming** → do the same things as the recursive memoized approach but in reverse

```
int fibonacci(int n) {
	if (n == 0) {
		return 0;
	} else if (n == 1) {
		return 1;
	}
	int[] memo = new int[n];
	memo[0] = 0;
	memo[1] = 1;
	for (int i = 2; i < n; i++) {
		memo[i] = memo[i - 1] + memo[i - 2];
	}
	return memo[n - 1] + memo[n - 2];
}
```

// **Triple Step** → run up the stairs with either 1, 2 or 3 steps at a time. How many possible ways can the child climb the stairs?

```
int countWays(int n) {
	if (n < 0) {
		return 0;
	} else if (n == 0) {
		return 1;
	} else {
```

```
		return countWays(n - 1) + countWays(n - 2) + countWays(n - 3);
	}
}
```

// Runtime is roughly O(3 ^ n)

// Tipically we use a HashMap<Integer, Integer> for caching → keys will be exactly from 1 to n

```
int countWays(int n) {
	int[] memo = new int[n + 1];
	Arrays.fill(memo, -1);
	return countWays(n, memo);
}

int countWays(int n, int[] memo) {
	if (n < 0) {
		return 0;
	} else if (n == 0) {
		return 1;
	} else if (memo[n] > -1) {
		return memo[n];
	} else {
		memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
				countWays(n - 3, memo);
	}
	return memo[n];
}
```

// int will overflow, long will prolongue it, could use BigInteger?

// **Robot in a Grid** → grid with r rows and c columns → find path from top left to bottom right moving only right and down and some cells are off limits

```
public class QuestionA {

	public static ArrayList<Point> getPath(boolean[][] maze) {
		if (maze == null || maze.length == 0) {
			return null;
		}
		ArrayList<Point> path = new ArrayList<Point>();
		if (getPath(maze, maze.length - 1, maze[0].length - 1, path)) {
			return path;
		}
```

```
		return null;
	}

	public static boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path) {
		// If out of bounds or not available return
		if (row < 0 || col < 0 || !maze[row][col]) {
			return false;
		}

		boolean isAtOrigin = (row == 0) && (col == 0);

		if (isAtOrigin || getPath(maze, row, col - 1, path) || getPath(maze, row - 1, col,
	path)) {
			Point p = new Point(row, col);
			path.add(p);
			return true;
		}

		return false;
	}
}
```

// Optimize the solution by remmebering failed points in a HashSet

// **Magic Index** → sorted array of distinct integer → write a method to find a magic index if one exists

// Brute force → not using the fact the array is sorted

```
int magicIndex(int[] array) {
	for (int i = 0; i < array.length; i++) {
		if (i == array[i]) {
			return i;
		}
	}
	return -1;
}
```

// Binary search similarities

```
int magic(int[] array) {
	return magic(array, 0, array.length - 1);
}
```

```
int magic(int[] array, int start, int end) {
	if (end < start) {
		return -1;
	}
	int mid = start + (end - start) / 2;
	if (array[mid] == mid) {
		return mid;
	} else if (array[mid] > mid) {
		return magic(array, start, mid - 1);
	} else {
		return magic(array, mid + 1, end);
	}
}

// What if the elements are not distinct?

int magic(int[] array) {
	return magic(array, 0, array.length);
}

int magic(int[] array, int start, int end) {
	if (end < start) {
		return -1;
	}

	int midIndex = start + (end - start) / 2;
	int midValue = array[midIndex];

	if (midValue == midIndex) {
		return midIndex;
	}

	int leftIndex = Math.min(midIndex - 1, midValue);
	int left = magic(array, start, leftIndex);

	if (left > 0) {
		return left;
	}

	int rightIndex = Math.max(midIndex + 1, midValue);
	int right = magic(array, rightIndex, end);
```

```
        return right;
}
```

// **Power Set** → write a method to return all subsets of a set

// How many subsets do we actually have → 2 ^ n
// Each of the n elements are going to be contained in half of all subsets → n * 2 ^ (n - 1)

P3 = P2 + Pd → Pd = (P2 + a3)

```
ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set, int index) {
        ArrayList<ArrayList<Integer>> allSubsets;

        if (set.size() == index) {
                allSubsets = new ArrayList<ArrayList<Integer>>();
                allSubsets.add(new ArrayList<Integer>());
        } else {
                allSubsets = getSubsets(set, index + 1);
                int item = set.get(index);
                ArrayList<ArrayList> moreSubsets = new ArrayList<ArrayList<Integer>>();
                for (ArrayList<Integer> subset : allSubsets) {
                        ArrayList<Integer> newSubset = new ArrayList<Integer>();
                        newSubset.addAll(subset);
                        newSubset.add(item);
                        moreSubsets.add(newSubset);
                }
                allSubsets.addAll(moreSubsets);
        }
        return allSubsets;
}
```

// **Walkthrough**

1 2 3 4

set.size() = 4;
index = 0

--> getSubsets(set, 1)

set.size() = 4
index = 1

--> getSubsets(set, 2);

set.size() = 4
index = 2

--> getSubsets(set, 3)

set.size() = 4
index = 3

--> getSubsets(set, 4)

set.size() = 4
index = 4

all = {}

--> set.size() = 4, index = 3, all = {}

item = 4

more = empty

subset for all subsets -->
new = subset + item = {4}
more = {4}

all = {}, {4}

--> set.size() = 4, index = 2, all = {}, {4}

item = 3

more = empty

subset for all subsets -->
new = {} + 3
new = {3}

more = {3}

new = {4} + 3
new = {4, 3}

more = {3}, {4, 3}

all = {}, {3}, {4}, {3, 4}
….

// Solve the problem using **Combinatorics** → we iterate through the numbers from 0 to 2 ^ n and translate the binary representation into sets

```
ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set) {
	ArrayList<ArrayList<Integer>> all = new ArrayList<>();
	int max = 1 << set.size();

	for (int k = 0; k < max; k++) {
		ArrayList<Integer> subset = convertIntToSet(k, set);
		all.add(subset);
	}

	return all;
}

ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
	ArrayList<Integer> subset = new ArrayList<>();

	int index = 0;
	for (int k = x; k > 0; k >>= 1) {
		if ((k & 1) == 1) {
			subset.add(set.get(index));
		}
		index++;
	}
	return subset;
}
```

// **Recursive Multiply** → multiply two positive integer without using the * operator

```
int minProduct(int a, int b) {
	int bigger = a < b ? b : a;
	int smaller = a < b ? a : b;
	return minProductHelper(smaller, bigger);
}
```

```
int minProductHelper(int smaller, int bigger) {
	if (smaller == 0) {
		return 0;
	} else if (smaller == 1) {
		return bigger;
	}

	int s = smaller >> 1;  // divide by 2
	int side1 = minProduct(s, bigger);
	int side2 = side1;

	if (smaller % 2 == 1) {
		side2 = minProductHelper(smaller - s, bigger);
	}
	return side1 + side2;
}
```

// **We have duplicated work** → we can do better by caching resutls

```
int minProduct(int a, int b) {
	int bigger = a < b ? b : a;
	int smaller = a < b ? a : b;

	int[] memo = new int[smaller + 1];
	return minProduct(smaller, bigger, memo);
}


int minProduct(int smaller, int bigger, int[] memo) {
	if (smaller == 0) {
		return 0;
	} else if (smaller == 1) {
		return bigger;
	} else if (memo[smaller] > 0) {
		return memo[smaller];
	}

	int s = smaller >> 1;  // divide by 2
	int side1 = minProduct(s, bigger, memo);
	int side2 = side1;

	if (smaller % 2 == 1) {
		side2 = minProduct(smaller - s, bigger, memo);
```

```
        }
        memo[smaller] = side1 + side2;
        return memo[smaller];
}
```

// **We can still make it a bit faster** → if the number is even we double the half → if the number is odd we can double the half and add the bigger number → no need to cache as the calls will not be the same

// **Towers of Hanoi** → 3 towers, disks sorted in ascending order on tower 1

```
public class Tower {
        private Stack<Integer> disks;
        private int index;

        public Tower(int index) {
                this.disks = new Stack<>();
                this.index = index;
        }

        public int index() {
                return index;
        }

        public boolean add(int disk) {
                if (!disks.isEmpty() && disks.peek() <= disk) {
                        return false;
                }
                disks.push(disk);
                return true;
        }

        public void moveTopTo(Tower tower) {
                int top = disks.pop();
                tower.add(top);
        }

        public void moveDisks(int n, Tower destination, Tower buffer) {
                if (n <= 0) {
                        return;
                }
                moveDisks(n - 1, buffer, destination);
                moveTopTo(destination);
```

```
        buffer.moveDisks(n - 1, destination, this);
    }
}
```

// **Permutations without Dups** → write a method to compute all permutations of a string of unique characters

```
ArrayList<String> getPerms(String str) {
    if (str == null) {
        return null;
    }

    ArrayList<String> permutations = new ArrayList<>();
    if (str.length() == 0) {
        permutations.add("");
        return permutations;
    }
    char first = str.charAt(0);
    String remainder = str.substring(1);
    ArrayList<String> words = getPerms(remainder);
    for (String word : words) {
        for (int j = 0; j <= word.length(); j++) {
            String s = insertCharAt(word, first, j);
            permutations.add(s);
        }
    }
    return permutations;
}

String insertCharAt(String word, char c, int i) {
    String start = word.substring(0, i);
    String end = word.substring(i);
    return start + c + end;
}

// Another approach

ArrayList<String> getPerms(String str) {
    ArrayList<String> result = new ArrayList<>();
    getPerms("", str, result);
    return result;
}
```

```
void getPerms(String prefix, String remainder, ArrayList<String> result) {
	if (remainder.length() == 0) {
		result.add(prefix);
	}
	int len = remainder.length();
	for (int i = 0; i < len; i++) {
		String before = remainder.substring(0, i);
		String after = remainder.substring(i + 1, len);
		char c = remainder.charAt(i);
		getPerms(prefix + c, before + after, result);
	}
}
```

// **Perms with dups** → characters of strings are not necessarily unique

```
ArrayList<String> printPerms(String s) {
	ArrayList<String> result = new ArrayList<>();
	HashMap<Character, Integer> map = buildFreqTable(s);
	printPerms(map, "", s.length(), result);
	return result;
}

HashMap<Character, Integer> buildFreqTable(String s) {
	HashMap<Character, Integer> map = new HashMap<>();
	for (char c : s.toCharArray()) {
		if (!map.containsKey(c)) {
			map.put(c, 0);
		}
		map.put(c, map.get(c) + 1);
	}
	return map;
}

void printPerms(HashMap<Character, Integer> map, String prefix, int remaining,
ArrayList<String> result) {
	if (remaining == 0) {
		result.add(prefix);
		return;
	}

	for (Character c : map.keySet()) {
		int count = map.get(c);
```

```
			if (count > 0) {
				map.put(c, count - 1);
				printPerms(map, prefix + c, remaining - 1, result);
				map.put(c, count);
			}
		}
}
```

// **Parens** → print all combinations of n pairs of parentheses

```
void parens(ArrayList<String> list, int left, int right, char[] str, int index) {
	if (left < 0 || right < left) {
		return;
	}

	if (left == 0 && right == 0) {
		list.add(String.copyValueOf(str));
	} else {
		str[index] = '(';
		parens(list, left - 1, right, str, index + 1);

		str[index] = ')';
		parens(list, left, right - 1, str, index + 1);
	}
}

ArrayList<String> generateParens(int count) {
	char[] str = new char[count * 2];
	ArrayList<String> list = new ArrayList<>();
	parens(list, count, count, str, 0);
	return list;
}
```

// **Another way**

```
def parens(left, right, sequence):
	if left == 0 and right == 0:
		print(sequence)
	if left > 0:
		parens(left - 1, right + 1, sequence + "(")
	if right > 0:
		parens(left, right - 1, sequence + ")")
```

// **Paint fill** → implement the paint fill function common in most image editors

```
enum Color {
	BLACK,
	WHITE,
	RED,
	YELLOW,
	GREEN;
}

boolean PaintFill(Color[][] screen, int r, int c, Color nColor) {
	if (screen[r][c] == nColor) {
		return false;
	}
	return PaintFill(screen, r, c, screen[r][c], nColor);
}

boolean PaintFill(Color[][] screen, int r, int c, Color oColor, Color nColor) {
	if (r < 0 || r >= screen.length || c < 0 || c > screen[0].length) {
		return false;  // out of bounds
	}

	if (screen[r][c] == oColor) {
		screen[r][c] = nColor;
		PaintFill(screen, r - 1, c, oColor, nColor);  // UP
		PaintFill(screen, r + 1, c, oColor, nColor);  // DOWN
		PaintFill(screen, r, c - 1, oColor, nColor);  // LEFT
		PaintFill(screen, r, c + 1, oColor, nColor);  // RIGHT
	}
	return true;
}
```

// **Coins** → make change

```
long makeChange(int[] coins, int money, int index, HashMap<String, Long> memo) {
	if (money == 0) {
		return 1;
	}

	if (index >= coins.length) {
		return 0;
	}
```

```
        String key = money + "-" + index;
        if (memo.containsKey(key)) {
                return memo.get(key);
        }

        int amountWithCoin = 0;
        long ways = 0;

        while (amountWithCoin <= money) {
                int remaining = money - amountWithCoin;
                ways += makeChange(coins, remaining, index + 1, memo);
                amountWithCoin += coins[index];
        }
        memo.put(key, ways);
        return ways;
}

long makeChange(int[] coins, int money) {
        return makeChange(coins, money, 0, new HashMap<String, Long>());
}
```

// **Eight Queens**

```
int GRID_SIZE = 8;

void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
        if (row == GRID_SIZE) {
                results.add(columns.clone());
        } else {
                for (int col = 0; col < GRID_SIZE; col++) {
                        if (checkValid(columns, row, col)) {
                                columns[row] = col;
                                placeQueens(row + 1, columns, results);
                        }
                }
        }
}

boolean checkValid(Integer[] columns, int row1, int column1) {
        for (int row2 = 0; row2 < row1; row2++) {
                int column2 = columns[row2];
                if (column1 == column2) {
                        return false;
```

```
            }
            int columnDistance = Math.abs(column2 - column1);
            int rowDistance = row1 - row2;
            if (columnDistance == rowDistance) {
                return false;
            }
        }
        return true;
}
```

// **Stack of Boxes**

```
int createStack(ArrayList<Box> boxes) {
        Collections.sort(boxes, new BoxComparator());
        int[] stackMap = new int[boxes.size()];
        return createStack(boxes, null, 0, stackMap);
}

int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
        if (offset >= boxes.size()) {
            return 0;
        }
        Box newBottom = boxes.get(offset);
        int heightWithBottom = 0;
        if (bottom == null || newBottom.canBeAbove(bottom)) {
            if (stackMap[offset] == 0) {
                stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
                stackMap[offset] += newBottom.height;
            }
            heightWithBottom = stackMap[offset];
        }

        int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);

        return Math.max(heightWithBottom, heightWithoutBottom);
}
```

// **Boolean Evaluation** → 0, 1, &, |, ^

```
public static int count = 0;
public static boolean stringToBool(String c) {
        return c.equals("1") ? true : false;
}
```

```
public static int countEval(String s, boolean result, HashMap<String, Integer> memo) {
	count++;
	if (s.length() == 0) return 0;
	if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
	if (memo.containsKey(result + s)) return memo.get(result + s);

	int ways = 0;

	for (int i = 1; i < s.length(); i += 2) {
		char c = s.charAt(i);
		String left = s.substring(0, i);
		String right = s.substring(i + 1, s.length());
		int leftTrue = countEval(left, true, memo);
		int leftFalse = countEval(left, false, memo);
		int rightTrue = countEval(right, true, memo);
		int rightFalse = countEval(right, false, memo);
		int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);

		int totalTrue = 0;
		if (c == '^') {
			totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
		} else if (c == '&') {
			totalTrue = leftTrue * rightTrue;
		} else if (c == '|') {
			totalTrue = leftTrue * rightTrue + leftFalse * rightTrue + leftTrue *
rightFalse;
		}

		int subWays = result ? totalTrue : total - totalTrue;
		ways += subWays;
	}

	memo.put(result + s, ways);
	return ways;
}

public static int countEval(String s, boolean result) {
	return countEval(s, result, new HashMap<String, Integer>());
}
```