JavaScript Extras

Problem 1 Preço Certo

A contestant plays a spinning game where:

- There is a wheel with **21 slots**, each containing a value ranging from **0 to 100**, in steps of 5:
$$\{0, 5, 10, \ldots, 95, 100\}$$
- The contestant spins the wheel **up to two times**.
  - If the **first spin is 100**, the player **does not spin again**.
  - Otherwise, they must spin a **second time**.
- The contestant's final score is the **sum of the two spins**.
  - If the sum is **greater than 100**, the contestant is **disqualified** (score = 0).
  - If the sum is **less than or equal to 100**, it becomes their final score.

Given the two spin results for a player (`r1` and `r2`), calculate the **probability that they will win**, assuming:

- All valid spin combinations are equally likely (and independent).
- The player wins if their final score is the **highest among all valid combinations**.
- In case of a **tie for the highest score**, it is considered a **draw** (re-spin), and **not a win**, hence, if we get 100 total score, it shouldn't output 1 as in guaranteed victory (consider all other ways you can get 100).
- A final score **over 100** results in **disqualification** (score = 0).

Solution:

```
function chanceToWin(r1, r2) {
  const values = [];
  for (let i = 0; i <= 100; i += 5) values.push(i);
  const firstSpinValues = values.filter(v => v !== 100); // 20
     values (because if you get 100 in the first one, you're not
     playing the second round)
  const secondSpinValues = values; // 21 values (if you played the
     first round and got something under 100, any other value is a
     possibility)
  const allFinalScores = [];
  // 420 combinations (when the first spin is not= 100)
  for (let i = 0; i < firstSpinValues.length; i++) {
    for (let j = 0; j < secondSpinValues.length; j++) {
      const first = firstSpinValues[i];
      const second = secondSpinValues[j];
      const sum = first + second;
      if (sum > 100) {
        allFinalScores.push(0); // disqualified
      } else {
        allFinalScores.push(sum);
      }
    }
  }
  // +1 combination: first spin = 100 -> automatic 100 (there are
     421 combinations in total)
  allFinalScores.push(100);
  const playerScore = (r1 === 100) ? 100 : (r1 + r2 > 100 ? 0 : r1 +
      r2);
  const wins = allFinalScores.filter(score => score < playerScore).
     length;
  return wins / allFinalScores.length;
}
```

Problem 2 Maximum Matrix Area

You are given an n x n matrix consisting of 0s and 1s. Your task is to find the largest submatrix that contains only 1s (a submatrix is a rectangular section of the matrix). The goal is to return the area (number of 1s) of this largest submatrix.

```
    function maximalRectangle(matrix) {
    if (!matrix.length || !matrix[0].length) return 0;
    const n = matrix.length;
    const m = matrix[0].length;
    let maxArea = 0;
    const heights = Array(m).fill(0);
```

```
    for (let row of matrix) {
        // Build histogram heights
        for (let j = 0; j < m; j++) {
            heights[j] = row[j] === 0 ? 0 : heights[j] + 1;
        }
        // Update max area with current row's histogram
        maxArea = Math.max(maxArea, largestRectangleArea(heights));
    }
    return maxArea;
}
function largestRectangleArea(heights) {
    const stack = [];
    let maxArea = 0;
    heights.push(0); // Add a sentinel value
    for (let i = 0; i < heights.length; i++) {
        while (stack.length && heights[i] < heights[stack[stack.
           length - 1]]) {
            const height = heights[stack.pop()];
            const width = stack.length ? i - stack[stack.length - 1]
                - 1 : i;
            maxArea = Math.max(maxArea, height * width);
        }
        stack.push(i);
    }
    heights.pop(); // Remove the sentinel value
    return maxArea;
}
```

Problem 3 Spiral Matrix

Given an m x n matrix, return all the elements of the matrix in spiral order.

```
    function spiralOrder(matrix) {
    const result = [];
    if (!matrix.length) return result;
    let top = 0;
    let bottom = matrix.length - 1;
    let left = 0;
    let right = matrix[0].length - 1;
    while (top <= bottom && left <= right) {
        // Left to right
        for (let i = left; i <= right; i++) {
            result.push(matrix[top][i]);
        }
        top++;
        // Top to bottom
```

```
        for (let i = top; i <= bottom; i++) {
            result.push(matrix[i][right]);
        }
        right--;
        // Right to left
        if (top <= bottom) {
            for (let i = right; i >= left; i--) {
                result.push(matrix[bottom][i]);
            }
            bottom--;
        }
        // Bottom to top
        if (left <= right) {
            for (let i = bottom; i >= top; i--) {
                result.push(matrix[i][left]);
            }
            left++;
        }
    }
    return result;
}
```

Problem 4 Find the intersection of Two arrays

Given two arrays, find their intersection. The result should include only unique elements.

```
    function intersection(nums1, nums2) {
    const set1 = new Set(nums1);
    const set2 = new Set(nums2);
    const result = [];
    for (let num of set2) {
        if (set1.has(num)) {
            result.push(num);
        }
    }
    return result;
}
```

Problem 5 Implement LRU Cache

Design and implement a data structure for a Least Recently Used (LRU) cache. It should support the following operations:

- get(key): Returns the value of the key if the key exists, otherwise returns -1;

- put(key, value): Inserts or updates the value of the key. If the cache reaches its capacity, it should invalidate the least recently used item before inserting a new one.

```
    class LRUCache {
    constructor(capacity) {
        this.capacity = capacity;
        this.cache = new Map();
    }
    get(key) {
        if (!this.cache.has(key)) return -1;
        const value = this.cache.get(key);
        this.cache.delete(key);
        this.cache.set(key, value); // Update order
        return value;
    }
    put(key, value) {
        if (this.cache.has(key)) {
            this.cache.delete(key);
        }
        this.cache.set(key, value);
        if (this.cache.size > this.capacity) {
            const firstKey = this.cache.keys().next().value;
            this.cache.delete(firstKey);
        }
    }
}
```