

# Chat history

[ChatGPT](#)

[ChatGPT](#)

[Explore GPTs](#)

## Today

[Game Over Detection](#)

[Candy Crush Clone p5.js](#)

## Yesterday

[Vijaykanth vs Terrorists Action](#)

## Previous 7 Days

[Dystopian Video Game Battle](#)

[Dhanush Nayanthara Army Scene](#)

Upgrade plan More access to the best models

ChatGPT

Share



**You said:**

set of images as img1.jpg img2.jpg. .... so on for a grid of random rectangular polygon of castle shape generated randomly is given and movement like candy crush game for similar candy given so rules of candy crush give me a p5 js script and sparkling animation on similar check of candy need a clone code on p5js and put game assets in a folder so all images generated randomly for candy and grid in simple colours and write the code also give me

**ChatGPT said:**

ChatGPT

Below is a Candy Crush-style game clone in p5.js, where randomly generated rectangles resembling castle-like polygons are used as the "candies." When matches of three or more similar "candies" occur, they disappear with a sparkling animation, and new "candies" are generated to fill the grid.

## Setup:

- Place all game assets (if any specific assets) in a folder named `assets`.
- If no assets exist, the script will generate random colors for the "candies."

Here's the code:

## Code:

javascript

Copy code

```
// Grid and candy variables let grid; let cols = 8; // Number of columns let rows = 8; // Number of rows let tileSize = 60; // Size of each tile let candies = []; let sparkleFrames = 10; // Number of frames for sparkle animation let sparkleParticles = []; // Preload images for candies function preload() { for (let i = 0; i < 5; i++) { candies.push(loadImage(`assets/candy${i + 1}.png`)); // Candy images in "assets" folder } } function setup() { createCanvas(cols * tileSize, rows * tileSize); grid = createGrid(cols, rows); } function draw() { background(30); // Draw the grid drawGrid(); // Handle sparkles for (let i = sparkleParticles.length - 1; i >= 0; i--) { let sparkle = sparkleParticles[i]; sparkle.display(); sparkle.update(); // Remove particle when it's finished if (sparkle.finished) { sparkleParticles.splice(i, 1); } } } // Create a grid of candies function createGrid(cols, rows) { let grid = []; for (let y = 0; y < rows; y++) { let row = []; for (let x = 0; x < cols; x++) { row.push(floor(random(candies.length))); // Random candy type } grid.push(row); } return grid; } // Draw the grid function drawGrid() { for (let y = 0; y < rows; y++) { for (let x = 0; x < cols; x++) { let candyIndex = grid[y][x]; if (candyIndex !== null) { image(candies[candyIndex], x * tileSize, y * tileSize, tileSize, tileSize); } } } } // Check for matches function checkMatches() { let matches = []; // Check rows for matches for (let y = 0; y < rows; y++) { for (let x = 0; x < cols - 2; x++) { if ( grid[y][x] !== null && grid[y][x] === grid[y][x + 1] && grid[y][x] === grid[y][x + 2] ) { matches.push({ x, y }); matches.push({ x: x + 1, y }); matches.push({ x: x + 2, y }); } } } // Check columns for matches for (let x = 0; x < cols; x++) { for (let y = 0; y < rows - 2; y++) { if ( grid[y][x] !== null && grid[y][x] === grid[y + 1][x] && grid[y][x] === grid[y + 2][x] ) { matches.push({ x, y }); matches.push({ x, y: y + 1 }); matches.push({ x, y: y + 2 }); } } } return matches; } // Create sparkle animation class Sparkle { constructor(x, y) { this.x = x; this.y = y; this.life = sparkleFrames; this.finished = false; } update() { this.life--; if (this.life <= 0) { this.finished = true; } } display() { noStroke(); fill(255, 255, random(100,
```

```

255), this.life * 25); ellipse( this.x + random(-10, 10), this.y + random(-10,
10), random(5, 10) ); } } // Trigger sparkle animation function
triggerSparkle(matches) { matches.forEach(({ x, y }) => { for (let i = 0; i < 20;
i++) { sparkleParticles.push( new Sparkle(x * tileSize + tileSize / 2, y *
tileSize + tileSize / 2) ); } }); } // Handle mouse click function mousePressed()
{ // Detect matches let matches = checkMatches(); if (matches.length > 0) { //
Trigger sparkle animation triggerSparkle(matches); // Remove matched candies
matches.forEach(({ x, y }) => { grid[y][x] = null; }); // Collapse grid
collapseGrid(); } } // Collapse the grid after matches function collapseGrid() {
for (let x = 0; x < cols; x++) { for (let y = rows - 1; y >= 0; y--) { if
(grid[y][x] === null) { // Find the next non-null candy above for (let k = y - 1;
k >= 0; k--) { if (grid[k][x] !== null) { grid[y][x] = grid[k][x]; grid[k][x] =
null; break; } } // If no candy found above, generate a new candy if (grid[y][x]
=== null) { grid[y][x] = floor(random(candies.length)); } } } } }

```

## How It Works:

1. Grid Initialization: A grid of random "candies" is created.
2. Match Checking: Matches of 3 or more similar candies are detected.
3. Sparkle Animation: Matches trigger a sparkling effect before disappearing.
4. Grid Collapse: Candies above the matches collapse downward, and new candies are generated to fill the grid.

## Assets:

- Save candy images in the `assets` folder, named as `candy1.png`, `candy2.png`, ..., `candy5.png`.
- Alternatively, replace the image-drawing code with colored rectangles for simpler visuals.

Let me know if you need further clarification or customization!

**You said:**

🌸 p5.js says: It looks like there was a problem loading your image. Try checking if the file path (<https://preview.p5js.org/assets/candy2.png>) is correct, hosting the file online, or running a local server.

+ More info: <https://github.com/processing/p5.js/wiki/Local-server>

Event {isTrusted: true, type: "error", target: null, currentTarget: null, eventPhase: 0...}

🌸 p5.js says: It looks like there was a problem loading your image. Try checking if the file path (<https://preview.p5js.org/assets/candy4.png>) is correct, hosting the file online, or running a local server.

+ More info: <https://github.com/processing/p5.js/wiki/Local-server>

Event {isTrusted: true, type: "error", target: null, currentTarget: null, eventPhase: 0...}

🌸 p5.js says: It looks like there was a problem loading your image. Try checking if the file path (<https://preview.p5js.org/assets/candy3.png>) is correct, hosting the file online, or running a local server.

+ More info: <https://github.com/processing/p5.js/wiki/Local-server>

Event {isTrusted: true, type: "error", target: null, currentTarget: null, eventPhase: 0...}

🌸 p5.js says: It looks like there was a problem loading your image. Try checking if the file path (<https://preview.p5js.org/assets/candy5.png>) is correct, hosting the file online, or running a local server.

+ More info: <https://github.com/processing/p5.js/wiki/Local-server>

Event {isTrusted: true, type: "error", target: null, currentTarget: null, eventPhase: 0...}

🌸 p5.js says: It looks like there was a problem loading your image. Try checking if the file path (<https://preview.p5js.org/assets/candy1.png>) is correct, hosting the file online, or running a local server.

+ More info: <https://github.com/processing/p5.js/wiki/Local-server>

Event {isTrusted: true, type: "error", target: null, currentTarget: null, eventPhase: 0...} generate all images by itself randomly put asset of candy and everything seperatly and call

**ChatGPT said:**

ChatGPT

To avoid issues with external assets, we can generate the candy images programmatically as simple geometric shapes or colored rectangles. This way, no external images are required, and everything is self-contained.

Here's a self-contained p5.js script that generates random "candies" as simple colored shapes with a unique design. These will be randomly assigned to the grid, and the game will function without the need for external assets.

---

## Complete Code

javascript

Copy code

```
// Grid and candy variables let grid; let cols = 8; // Number of columns let rows = 8; // Number of rows let tileSize = 60; // Size of each tile let candyColors = []; // Colors for different candies let sparkleParticles = []; let sparkleFrames = 10; // Sparkle animation frames function setup() { createCanvas(cols * tileSize, rows * tileSize); // Generate random candy colors for (let i = 0; i < 5; i++) { candyColors.push(color(random(50, 255), random(50, 255), random(50, 255))); } // Initialize the grid grid = createGrid(cols, rows); } function draw() { background(30); // Draw the grid drawGrid(); // Handle sparkles for (let i = sparkleParticles.length - 1; i >= 0; i--) { let sparkle = sparkleParticles[i]; sparkle.display(); sparkle.update(); // Remove particle when it's finished if (sparkle.finished) { sparkleParticles.splice(i, 1); } } } // Create a grid of random candies function createGrid(cols, rows) { let grid = []; for (let y = 0; y < rows; y++) { let row = []; for (let x = 0; x < cols; x++) { row.push(floor(random(candyColors.length))); // Random candy type } grid.push(row); } return grid; } // Draw the grid function drawGrid() { for (let y = 0; y < rows; y++) { for (let x = 0; x < cols; x++) { let candyType = grid[y][x]; if (candyType !== null) { drawCandy(x * tileSize, y * tileSize, candyType); } } } } // Draw a single candy at the specified position function drawCandy(x, y, type) { noStroke(); fill(candyColors[type]); rect(x + 10, y + 10, tileSize - 20, tileSize - 20, 10); // Rounded rectangles for candies } // Check for matches function checkMatches() { let matches = []; // Check rows for matches for (let y = 0; y < rows; y++) { for (let x = 0; x < cols - 2; x++) { if ( grid[y][x] !== null && grid[y][x] === grid[y][x + 1] && grid[y][x] === grid[y][x + 2] ) { matches.push({ x, y }); matches.push({ x: x + 1, y }); matches.push({ x: x + 2, y }); } } } // Check columns for matches for (let x = 0; x < cols; x++) { for (let y = 0; y <
```

```

rows - 2; y++) { if ( grid[y][x] !== null && grid[y][x] === grid[y + 1][x] &&
grid[y][x] === grid[y + 2][x] ) { matches.push({ x, y }); matches.push({ x, y: y +
1 }); matches.push({ x, y: y + 2 }); } } } return matches; } // Create sparkle
animation class Sparkle { constructor(x, y) { this.x = x; this.y = y; this.life =
sparkleFrames; this.finished = false; } update() { this.life--; if (this.life <=
0) { this.finished = true; } } display() { noStroke(); fill(255, random(150, 255),
random(150, 255), this.life * 25); ellipse( this.x + random(-10, 10), this.y +
random(-10, 10), random(5, 10) ); } } // Trigger sparkle animation function
triggerSparkle(matches) { matches.forEach(({ x, y }) => { for (let i = 0; i < 20;
i++) { sparkleParticles.push( new Sparkle(x * tileSize + tileSize / 2, y *
tileSize + tileSize / 2) ); } }); } // Handle mouse click function mousePressed()
{ // Detect matches let matches = checkMatches(); if (matches.length > 0) { //
Trigger sparkle animation triggerSparkle(matches); // Remove matched candies
matches.forEach(({ x, y }) => { grid[y][x] = null; }); // Collapse grid
collapseGrid(); } } // Collapse the grid after matches function collapseGrid() {
for (let x = 0; x < cols; x++) { for (let y = rows - 1; y >= 0; y--) { if
(grid[y][x] === null) { // Find the next non-null candy above for (let k = y - 1;
k >= 0; k--) { if (grid[k][x] !== null) { grid[y][x] = grid[k][x]; grid[k][x] =
null; break; } } // If no candy found above, generate a new candy if (grid[y][x]
=== null) { grid[y][x] = floor(random(candyColors.length)); } } } } } }

```

---

## Features:

1. Self-Contained "Candies": Generates colored rounded rectangles instead of external assets.
2. Sparkle Animation: Animates sparkle particles when a match occurs.
3. Matching Logic: Detects matches in rows and columns.
4. Grid Collapse: Candies fall down to fill gaps, and new ones are generated at the top.

## To Test:

- Copy the code into the p5.js Web Editor.

- Run the sketch to see the game in action. No additional files are required!

**You said:**

give candy gradient finish and give according to these rules rewrite the code "// Grid and candy variables

```
let grid;
let cols = 8; // Number of columns
let rows = 8; // Number of rows
let tileSize = 60; // Size of each tile
let candyColors = []; // Colors for different candies
let sparkleParticles = [];
let sparkleFrames = 10; // Sparkle animation frames

function setup() {
  createCanvas(cols * tileSize, rows * tileSize);

  // Generate random candy colors
  for (let i = 0; i < 5; i++) {
    candyColors.push(color(random(50, 255), random(50, 255), random(50, 255)));
  }

  // Initialize the grid
  grid = createGrid(cols, rows);
}

function draw() {
  background(30);

  // Draw the grid
  drawGrid();

  // Handle sparkles
  for (let i = sparkleParticles.length - 1; i >= 0; i--) {
    let sparkle = sparkleParticles[i];
    sparkle.display();
    sparkle.update();

    // Remove particle when it's finished
    if (sparkle.finished) {
      sparkleParticles.splice(i, 1);
    }
  }
}

// Create a grid of random candies
```

```

function createGrid(cols, rows) {
  let grid = [];
  for (let y = 0; y < rows; y++) {
    let row = [];
    for (let x = 0; x < cols; x++) {
      row.push(floor(random(candyColors.length))); // Random candy type
    }
    grid.push(row);
  }
  return grid;
}

```

```

// Draw the grid
function drawGrid() {
  for (let y = 0; y < rows; y++) {
    for (let x = 0; x < cols; x++) {
      let candyType = grid[y][x];
      if (candyType !== null) {
        drawCandy(x * tileSize, y * tileSize, candyType);
      }
    }
  }
}

```

```

// Draw a single candy at the specified position
function drawCandy(x, y, type) {
  noStroke();
  fill(candyColors[type]);
  rect(x + 10, y + 10, tileSize - 20, tileSize - 20, 10); // Rounded rectangles for candies
}

```

```

// Check for matches
function checkMatches() {
  let matches = [];

  // Check rows for matches
  for (let y = 0; y < rows; y++) {
    for (let x = 0; x < cols - 2; x++) {
      if (
        grid[y][x] !== null &&
        grid[y][x] === grid[y][x + 1] &&
        grid[y][x] === grid[y][x + 2]
      ) {
        matches.push({ x, y });
      }
    }
  }
}

```



```

        matches.push({ x: x + 1, y });
        matches.push({ x: x + 2, y });
    }
}

// Check columns for matches
for (let x = 0; x < cols; x++) {
    for (let y = 0; y < rows - 2; y++) {
        if (
            grid[y][x] !== null &&
            grid[y][x] === grid[y + 1][x] &&
            grid[y][x] === grid[y + 2][x]
        ) {
            matches.push({ x, y });
            matches.push({ x, y: y + 1 });
            matches.push({ x, y: y + 2 });
        }
    }
}

return matches;
}

// Create sparkle animation
class Sparkle {
    constructor(x, y) {
        this.x = x;
        this.y = y;
        this.life = sparkleFrames;
        this.finished = false;
    }

    update() {
        this.life--;
        if (this.life <= 0) {
            this.finished = true;
        }
    }

    display() {
        noStroke();
        fill(255, random(150, 255), random(150, 255), this.life * 25);
        ellipse(

```

```

        this.x + random(-10, 10),
        this.y + random(-10, 10),
        random(5, 10)
    );
}
}

```

```

// Trigger sparkle animation
function triggerSparkle(matches) {
    matches.forEach(({ x, y }) => {
        for (let i = 0; i < 20; i++) {
            sparkleParticles.push(
                new Sparkle(x * tileSize + tileSize / 2, y * tileSize + tileSize / 2)
            );
        }
    });
}

```

```

// Handle mouse click
function mousePressed() {
    // Detect matches
    let matches = checkMatches();
    if (matches.length > 0) {
        // Trigger sparkle animation
        triggerSparkle(matches);
    }
}

```

```

// Remove matched candies
matches.forEach(({ x, y }) => {
    grid[y][x] = null;
});

```

```

// Collapse grid
collapseGrid();
}
}

```

```

// Collapse the grid after matches
function collapseGrid() {
    for (let x = 0; x < cols; x++) {
        for (let y = rows - 1; y >= 0; y--) {
            if (grid[y][x] === null) {
                // Find the next non-null candy above
                for (let k = y - 1; k >= 0; k--) {
                    if (grid[k][x] !== null) {

```

```

        grid[y][x] = grid[k][x];
        grid[k][x] = null;
        break;
    }
}

// If no candy found above, generate a new candy
if (grid[y][x] === null) {
    grid[y][x] = floor(random(candyColors.length));
}
}
}
}
} "from typing import List

```

class Solution:

```

    def candyCrush(self, board: List[List[int]]) -> List[List[int]]:
        # Dimensions of the board
        num_rows, num_cols = len(board), len(board[0])

        # Flag to indicate if we should continue crushing candies
        should_crush = True

        # Keep crushing candies until no more moves can be made
        while should_crush:
            should_crush = False # Reset the flag for each iteration

            # Mark the candies to be crushed horizontally
            for row in range(num_rows):
                for col in range(num_cols - 2):
                    candy_value = abs(board[row][col])
                    # Check if three consecutive candies have the same value
                    if candy_value != 0 and candy_value == abs(board[row][col + 1]) ==
abs(board[row][col + 2]):
                        should_crush = True # We will need another pass after crushing
                        # Mark the candies for crushing by negating their value
                        board[row][col] = board[row][col + 1] = board[row][col + 2] = -candy_value

            # Mark the candies to be crushed vertically
            for col in range(num_cols):
                for row in range(num_rows - 2):
                    candy_value = abs(board[row][col])
                    # Check if three consecutive candies vertically have the same value

```

```

        if candy_value != 0 and candy_value == abs(board[row + 1][col]) == abs(board[row
+ 2][col]):
            should_crush = True # We will need another pass after crushing
            # Mark the candies for crushing
            board[row][col] = board[row + 1][col] = board[row + 2][col] = -candy_value

# Drop the candies to fill the empty spaces caused by crushing
if should_crush:
    for col in range(num_cols):
        # Pointer to where the next non-crushed candy will fall
        write_row = num_rows - 1
        for row in range(num_rows - 1, -1, -1):
            # If the candy is not marked for crushing, bring it down
            if board[row][col] > 0:
                board[write_row][col] = board[row][col]
                write_row -= 1
        # Fill the remaining spaces at the top with zeros
        while write_row >= 0:
            board[write_row][col] = 0
            write_row -= 1

# Return the modified board after all possible crushes are completed
return board " usethiswrite in p5 js

```

ChatGPT said:

ChatGPT

Here's an updated p5.js implementation with gradient candies and logic based on the

`Solution.candyCrush` approach you provided. The game will follow the candy crushing rules, detect matches, and include gradient finishes for the candies.

---

## Complete Code with Gradient Candies and Improved Logic

javascript

Copy code

```

// Grid and candy variables let grid; let cols = 8; // Number of columns let rows =
8; // Number of rows let tileSize = 60; // Size of each tile let candyColors = [];
// Gradient colors for different candies let sparkleParticles = []; let
sparkleFrames = 10; // Sparkle animation frames function setup() {
createCanvas(cols * tileSize, rows * tileSize); // Generate gradient colors for

```

```

candies for (let i = 0; i < 5; i++) { let color1 = color(random(50, 200),
random(50, 200), random(50, 200)); let color2 = color(random(200, 255),
random(200, 255), random(200, 255)); candyColors.push([color1, color2]); } //
Initialize the grid grid = createGrid(cols, rows); } function draw() {
background(30); // Draw the grid drawGrid(); // Handle sparkles for (let i =
sparkleParticles.length - 1; i >= 0; i--) { let sparkle = sparkleParticles[i];
sparkle.display(); sparkle.update(); // Remove particle when it's finished if
(sparkle.finished) { sparkleParticles.splice(i, 1); } } } // Create a grid of
random candies function createGrid(cols, rows) { let grid = []; for (let y = 0; y
< rows; y++) { let row = []; for (let x = 0; x < cols; x++) {
row.push(floor(random(candyColors.length))); // Random candy type }
grid.push(row); } return grid; } // Draw the grid function drawGrid() { for (let y
= 0; y < rows; y++) { for (let x = 0; x < cols; x++) { let candyType = grid[y][x];
if (candyType !== null) { drawCandy(x * tileSize, y * tileSize, candyType); } } }
} // Draw a single candy with gradient function drawCandy(x, y, type) { let
gradient = candyColors[type]; let color1 = gradient[0]; let color2 = gradient[1];
// Create a gradient effect for (let i = 0; i < tileSize - 20; i++) { let inter =
map(i, 0, tileSize - 20, 0, 1); let col = lerpColor(color1, color2, inter);
stroke(col); line(x + 10, y + 10 + i, x + tileSize - 10, y + 10 + i); } // Add a
highlight noStroke(); fill(255, 255, 255, 50); ellipse(x + tileSize / 2, y +
tileSize / 3, tileSize / 3, tileSize / 4); } // Check for matches and mark candies
for removal function checkMatches() { let matches = []; // Check rows for matches
for (let y = 0; y < rows; y++) { for (let x = 0; x < cols - 2; x++) { let
candyType = Math.abs(grid[y][x]); if ( candyType !== null && candyType ===
Math.abs(grid[y][x + 1]) && candyType === Math.abs(grid[y][x + 2]) ) {
matches.push({ x, y }); matches.push({ x: x + 1, y }); matches.push({ x: x + 2, y
}); grid[y][x] = grid[y][x + 1] = grid[y][x + 2] = -candyType; } } } // Check
columns for matches for (let x = 0; x < cols; x++) { for (let y = 0; y < rows - 2;
y++) { let candyType = Math.abs(grid[y][x]); if ( candyType !== null && candyType
=== Math.abs(grid[y + 1][x]) && candyType === Math.abs(grid[y + 2][x]) ) {

```

```

matches.push({ x, y }); matches.push({ x, y: y + 1 }); matches.push({ x, y: y + 2
}); grid[y][x] = grid[y + 1][x] = grid[y + 2][x] = -candyType; } } } return
matches; } // Collapse the grid after matches function collapseGrid() { for (let x
= 0; x < cols; x++) { let writeRow = rows - 1; for (let y = rows - 1; y >= 0; y--)
{ if (grid[y][x] > 0) { grid[writeRow][x] = grid[y][x]; writeRow--; } } while
(writeRow >= 0) { grid[writeRow][x] = floor(random(candyColors.length));
writeRow--; } } } // Trigger sparkle animation function triggerSparkle(matches) {
matches.forEach(({ x, y }) => { for (let i = 0; i < 20; i++) {
sparkleParticles.push( new Sparkle(x * tileSize + tileSize / 2, y * tileSize +
tileSize / 2) ); } }); } // Create sparkle animation class Sparkle {
constructor(x, y) { this.x = x; this.y = y; this.life = sparkleFrames;
this.finished = false; } update() { this.life--; if (this.life <= 0) {
this.finished = true; } } display() { noStroke(); fill(255, random(150, 255),
random(150, 255), this.life * 25); ellipse( this.x + random(-10, 10), this.y +
random(-10, 10), random(5, 10) ); } } // Handle mouse click function
mousePressed() { let matches = checkMatches(); if (matches.length > 0) {
triggerSparkle(matches); // Remove matched candies matches.forEach(({ x, y }) => {
grid[y][x] = null; }); collapseGrid(); } }

```

---

## Key Features:

1. Gradient Candies:
  - Each candy is drawn with a gradient effect using two colors.
  - Highlights are added to simulate a shiny candy-like finish.
2. Match Detection:
  - Detects matches in rows and columns using the `candyCrush` logic.
  - Marks candies for removal using negative values.
3. Grid Collapse:
  - Cascades candies downward and fills empty spaces with new candies.
4. Sparkle Animation:
  - Adds sparkling effects at positions where candies are crushed.

## Try It:

Copy and paste this code into the p5.js Web Editor and run it. Adjust `cols`, `rows`, or `tileSize` to customize the grid size or visuals.

### You said:

Start Here

Flowchart

Speedrun


Templates

Go Pro Special Offer

dark mode toggle

More

 Log In

 Unlock 50% off right now — claim your discount instantly!

You are on level Student

Search...

Filter by tags

Getting Started

Overview

LeetCode Patterns: The Smart, Data-Driven Way to Prepare for Tech Interviews

How to Study

Math Basics

Runtime to Algo Cheat Sheet

Keyword to Algo Cheat Sheet

Basic Data Structures

Basic Data Structures and Algorithms

Stack Intro

Queue Intro

Hashmap Intro

Python Basic Data Structures

Java Basic Data Structures

JavaScript Basic Data Structures

C++ Basic Data Structures

Basic Algorithms

Intro to Sorting

Advanced Sorting Algorithms - Merge Sort | Quick Sort

Sorting Summary

Built-in Sort with Custom Comparator

Simulation

Simulation Coding Problems: Introduction and Strategies

Binary Search

Depth First Search

Backtracking

Breadth First Search

Graph

Two Pointers

Priority Queue / Heap

Dynamic Prog.

Adv. Data Structures

Miscellaneous

OOP Design

System Design

Company OAs

LeetCode Editorials

723. Candy Crush

Medium

Array

Two Pointers

Matrix

Simulation

Leetcode Link

Problem Description

This problem requires you to simulate the game "Candy Crush". Given a 2D grid representing a candy board of size  $m \times n$ , where each cell contains an integer that represents a different type of candy, your task is to implement an algorithm that performs a series of crushing actions according to the game's rules.

The crushing rules are as follows:



When three or more candies of the same type are adjacent either vertically or horizontally, they all get crushed at once, and their positions are set to zero, indicating an empty cell.

Once candies are crushed, gravity comes into play. Any candies above the empty cells will fall down to fill the empty spaces.

New candies do not enter the board from the top. Only existing candies drop down.

The crushing and dropping process continues until no more crushing can occur—i.e., no groups of three or more adjacent candies of the same type are left.

The goal is to keep applying these rules until the board reaches a stable state with no more possible crushes. The final stable board should be returned.

### Intuition

The solution is iterative and simulates the game's mechanics step by step. Here's how the intuition unfolds:

**Initial Check for Crushes:** Start by scanning the board to find any rows or columns of three or more candies that can be crushed. We keep a boolean flag, `run`, tracking if we performed a crushing operation during the current iteration. This flag indicates whether we need to continue processing the board.

**Marking Candies for Crushing:** When we find three or more candies in a row or column that match, we mark them by negating their value. It allows us to distinguish between candies that will be crushed and the rest of the board.

**Crushing Candies:** Once we are done checking the entire board and have marked all candies to be crushed, we perform the actual crushing by replacing the marked candies (negative numbers) with zeros.

**Implementing Gravity:** After crushing, simulate gravity by letting the uncrushed candies (positive numbers) fall down to fill any gaps. This is done by moving candies down within each column, starting from the bottom up.

**Repeating the Process:** Once we've completed a crush and gravity step, we need to check if more crushes can be made because the falling candies might create new sequences of three or more matching candies. We repeat the process as long as we keep crushing candies in each iteration. The `run` flag helps us to determine this; if set to `True`, at least one crushing occurred, so we need another pass.

**Stable Board:** Finally, when no more crushes occur in a whole pass (i.e., `run` is `False`), the board is stable, and we return it.

By iterating through these steps, the given algorithm ensures that all possible crushes are made and that the board reaches a stable state according to Candy Crush rules.

Learn more about Two Pointers patterns.

### Solution Approach

The implementation of the solution involves several important steps, each taking advantage of basic algorithms and data structures.

**Marking Candies to Crush:** We loop through the entire board, searching for sequences of three or more similar candies either horizontally or vertically. We use two nested loops for this - one for rows and one for columns. When we find such a sequence, we mark the candies by negating their values using `abs(board[i][j])` to indicate these will be crushed. The use of negation allows us to retain the candy type information, which is important to check for subsequent crushes in later iterations.

**Horizontal Crushing:** This part checks for three or more adjacent candies horizontally. The following snippet from the code performs horizontal checks and marks the candies:

```
for i in range(m):
    for j in range(n - 2):
        if (board[i][j] != 0
            and abs(board[i][j]) == abs(board[i][j + 1])
            and abs(board[i][j]) == abs(board[i][j + 2])):
            run = True
            board[i][j] = board[i][j + 1] = board[i][j + 2] = -abs(board[i][j])
```

**Vertical Crushing:** Similar to the horizontal checks, we perform vertical checks as well, identifying candies that should be crushed and marking them as negative numbers. Code snippet for the vertical check:

```
for j in range(n):
    for i in range(m - 2):
        if (board[i][j] != 0
            and abs(board[i][j]) == abs(board[i + 1][j])
            and abs(board[i][j]) == abs(board[i + 2][j])):
            run = True
            board[i][j] = board[i + 1][j] = board[i + 2][j] = -abs(board[i][j])
```

**Candies Falling (Gravity):** Once all possible candies are marked, we need to let all the unmarked (positive) candies fall down. We iterate through each column starting from the bottom row and move unmarked candies down to the 'curr' position, which represents the next available uncrushed position from the bottom. The code for simulating gravity is as follows:

```
for j in range(n):
    curr = m - 1
    for i in range(m - 1, -1, -1):
        if board[i][j] > 0:
```

```
        board[curr][j] = board[i][j]
        curr -= 1
    while curr > -1:
        board[curr][j] = 0
        curr -= 1
```

Repeating Until Stable: The outer while run: loop allows us to repeat this process until no further candies can be crushed. The run flag is set to True whenever a crush happens, prompting another iteration. When an iteration completes with no candies crushed (run is False), the board is stable, and the loop will terminate.

Returning the Result: After the loop exits, we return the stabilized board. There are no extra data structures used outside of the input board, which is mutated in place to save space, and simple variables for iteration control.

By iterating over the marked candies and applying gravity as long as there are candies to crush, the algorithm ensures an accurate portrayal of the Candy Crush mechanism.

Ready to land your dream job?

Unlock your dream job with a 2-minute evaluator for a personalized learning plan!

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using a simple 3x3 grid to demonstrate the Candy Crush algorithm.

Suppose our initial 3x3 grid is as follows:

```
1 1 1
2 2 3
3 3 2
```

Following the solution approach:

Marking Candies to Crush: First, we look for horizontal and vertical sequences of three or more similar candies to mark for crushing. We see that the top row has three 1s horizontally.

After marking, our grid looks like this:

```
-1 -1 -1
2 2 3
3 3 2
```

There are no vertical sequences to mark, so we continue to the next step.

Horizontal and Vertical Crushing: Since we've already marked the candies in step 1, we now set those positions to zero, simulating the crush.

Our grid now looks like this:

0 0 0

2 2 3

3 3 2

Candies Falling (Gravity): We apply gravity, allowing the candies to fall down to fill the empty spaces. Starting from the bottom of each column, we move positive numbers down.

After gravity, our grid becomes:

0 0 0

0 0 0

2 2 3

Now we have uncrushed candies at the bottom with zeros representing the empty cells above them.

Repeating Until Stable: We need to check again if any new sequences of three or more similar candies have formed due to gravity. Since no such sequences exist, no further action is required, and the run flag remains False.

Returning the Result: With no further crushes possible, and the board in a stable state, this is the final board state:

0 0 0

0 0 0

2 2 3

And this is the state we return as our answer, illustrating a simple run-through of the Candy Crush algorithm.

Solution Implementation

Python

Java

C++

TypeScript

from typing import List

class Solution:

def candyCrush(self, board: List[List[int]]) -> List[List[int]]:

# Dimensions of the board

num\_rows, num\_cols = len(board), len(board[0])

# Flag to indicate if we should continue crushing candies

should\_crush = True

# Keep crushing candies until no more moves can be made

```

while should_crush:
    should_crush = False # Reset the flag for each iteration

    # Mark the candies to be crushed horizontally
    for row in range(num_rows):
        for col in range(num_cols - 2):
            candy_value = abs(board[row][col])
            # Check if three consecutive candies have the same value
            if candy_value != 0 and candy_value == abs(board[row][col + 1]) ==
abs(board[row][col + 2]):
                should_crush = True # We will need another pass after crushing
                # Mark the candies for crushing by negating their value
                board[row][col] = board[row][col + 1] = board[row][col + 2] = -candy_value

    # Mark the candies to be crushed vertically
    for col in range(num_cols):
        for row in range(num_rows - 2):
            candy_value = abs(board[row][col])
            # Check if three consecutive candies vertically have the same value
            if candy_value != 0 and candy_value == abs(board[row + 1][col]) == abs(board[row
+ 2][col]):
                should_crush = True # We will need another pass after crushing
                # Mark the candies for crushing
                board[row][col] = board[row + 1][col] = board[row + 2][col] = -candy_value

    # Drop the candies to fill the empty spaces caused by crushing
    if should_crush:
        for col in range(num_cols):
            # Pointer to where the next non-crushed candy will fall
            write_row = num_rows - 1
            for row in range(num_rows - 1, -1, -1):
                # If the candy is not marked for crushing, bring it down
                if board[row][col] > 0:
                    board[write_row][col] = board[row][col]
                    write_row -= 1
            # Fill the remaining spaces at the top with zeros
            while write_row >= 0:
                board[write_row][col] = 0
                write_row -= 1

    # Return the modified board after all possible crushes are completed
    return board

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the nested loops and the operations performed within them. Let's analyze it step by step.

The outer while loop that repeats until no more candies can be crushed ( $run = False$ ) will run until the board stabilizes, which relates to the dimensions of the board. In the worst case, this might be  $O(m * n)$  because each candy might need to fall all the way down the board in the most extreme case.

Inside the while loop, there are two sets of nested for loops: one for checking rows and one for checking columns for crushable candies. Both of these sets involve iterating over all elements on the board once, giving us a  $O(m * n)$  for each full scan.

Crushing candies and making them negative take  $O(1)$  time each, but since it's inside the nested loops, it doesn't add more than a constant factor to the overall complexity of scanning the board.

The final part within the while loop is another nested loop that handles gravity, making candies fall down. This, again, requires going over each column and, in the worst case, moving each candy down to the bottom, which takes  $O(m)$  for each column. Multiplying by  $n$  columns gives us  $O(m * n)$ .

Combining these, in the worst case, the while loop may run  $m * n$  times due to the possibility of each iteration only crushing one candy and making others fall one place. This is a pessimistic estimate, but it serves as an upper boundary. Thus, the time complexity of the nested loops, taken together, results in  $O((m * n) * (m * n))$  or  $O((m * n)^2)$ .

### Space Complexity

The space complexity of the code is:

$O(1)$  for the variables  $m$ ,  $n$ ,  $run$ ,  $curr$ , and  $i$ ,  $j$  inside loops, as they do not depend on the size of the board and use a constant amount of space.

The input board is modified in-place and no additional data structures of significant size are created or used in the process, meaning we do not use additional space proportional to the size of the input.

Hence, the space complexity of the algorithm is  $O(1)$ .

Note: While the actual number of iterations needed to stabilize the board might be much less than  $m * n$ , we are considering the worst-case time complexity here for completeness.

Learn more about how to find time and space complexity quickly using problem constraints.

Discover Your Strengths and Weaknesses: Take Our 2-Minute Quiz to Tailor Your Study Plan:

Question 1 out of 10

Which of the following problems can be solved with backtracking (select multiple)

Generating subsets

Generating random numbers

Generating permutations

Sorting integers

Recommended Readings

Tech Interview Pattern | Two Pointers Introduction

Tech Interview Pattern Two Pointers Introduction If you prefer videos here's a super quick introduction to Two Pointers 

iframe src https www youtube com embed xZ4AfXHQ1VQ title YouTube video player frameborder 0 allow accelerometer autoplay clipboard write encrypted media gyroscope picture in picture allowfullscreen

Two pointers is a common interview

LeetCode Patterns: The Smart, Data-Driven Way to Prepare for Tech Interviews

LeetCode Patterns Your Personal Dijkstra's Algorithm to Landing Your Dream Job The goal of AlgoMonster is to help you get a job in the shortest amount of time possible in a data driven way We compiled datasets of tech interview problems and broke them down by patterns This way we

Recursion Review

Recursion Recursion is one of the most important concepts in computer science Simply speaking recursion is the process of a function calling itself Using a real life analogy imagine a scenario where you invite your friends to lunch <https://algomonster.s3.us-east-2.amazonaws.com/recursion.jpg> You first

Want a Structured Path to Master System Design Too? Don't Miss This!

Coding Interview Strategies

Dive into our free, detailed pattern charts and company guides to understand what each company focuses on.

use this rules and give correct code

**ChatGPT said:**

ChatGPT

Here's the complete implementation of the Candy Crush problem in Python following the provided rules and explanation. The code ensures correctness by marking candies for crushing, simulating gravity, and iterating until the board is stable.

python

Copy code

```
from typing import List class Solution: def candyCrush(self, board:
List[List[int]]) -> List[List[int]]: # Dimensions of the board num_rows, num_cols
= len(board), len(board[0]) # Flag to indicate if we should continue crushing
candies should_crush = True # Keep crushing candies until no more moves can be made
while should_crush: should_crush = False # Reset the flag for each iteration # Mark
the candies to be crushed horizontally for row in range(num_rows): for col in
range(num_cols - 2): candy_value = abs(board[row][col]) # Check if three
consecutive candies have the same value if candy_value != 0 and candy_value ==
abs(board[row][col + 1]) == abs(board[row][col + 2]): should_crush = True # We
will need another pass after crushing # Mark the candies for crushing by negating
their value board[row][col] = board[row][col + 1] = board[row][col + 2] =
-candy_value # Mark the candies to be crushed vertically for col in
range(num_cols): for row in range(num_rows - 2): candy_value =
abs(board[row][col]) # Check if three consecutive candies vertically have the same
value if candy_value != 0 and candy_value == abs(board[row + 1][col]) ==
abs(board[row + 2][col]): should_crush = True # We will need another pass after
crushing # Mark the candies for crushing board[row][col] = board[row + 1][col] =
board[row + 2][col] = -candy_value # Drop the candies to fill the empty spaces
caused by crushing if should_crush: for col in range(num_cols): # Pointer to where
the next non-crushed candy will fall write_row = num_rows - 1 for row in
range(num_rows - 1, -1, -1): # If the candy is not marked for crushing, bring it
down if board[row][col] > 0: board[write_row][col] = board[row][col] write_row -=
1 # Fill the remaining spaces at the top with zeros while write_row >= 0:
board[write_row][col] = 0 write_row -= 1 # Return the modified board after all
possible crushes are completed return board
```



## Explanation of Code

1. Candy Marking:
  - Horizontal and vertical groups of three or more candies are identified using nested loops. Matching candies are marked for crushing by negating their values.
2. Candy Crushing:
  - Marked candies are set to zero to simulate the crushing process.
3. Gravity Simulation:
  - For each column, uncrushed candies "fall" to the lowest possible positions. The gaps above are filled with zeros.
4. Repetition:
  - The process repeats until no further crushes are possible, determined by the

**You said:**

i said in p5 jsscript use this "

Start Here

Flowchart


Speedrun


Templates

Go Pro Special Offer

dark mode toggle

More

 Log In

 Unlock 50% off right now — claim your discount instantly!

You are on level Student

Search...

Filter by tags

Getting Started

Overview

LeetCode Patterns: The Smart, Data-Driven Way to Prepare for Tech Interviews

How to Study

Math Basics

Runtime to Algo Cheat Sheet

Keyword to Algo Cheat Sheet

Basic Data Structures

Basic Data Structures and Algorithms

Stack Intro

Queue Intro

HashMap Intro

Python Basic Data Structures

Java Basic Data Structures

JavaScript Basic Data Structures

C++ Basic Data Structures

Basic Algorithms

Intro to Sorting

Advanced Sorting Algorithms - Merge Sort | Quick Sort

Sorting Summary

Built-in Sort with Custom Comparator

Simulation

Simulation Coding Problems: Introduction and Strategies

Binary Search

Depth First Search

Backtracking

Breadth First Search

Graph

Two Pointers

Priority Queue / Heap

Dynamic Prog.

Adv. Data Structures

Miscellaneous

OOP Design

System Design

Company OAs

LeetCode Editorials

723. Candy Crush

Medium

Array

Two Pointers

Matrix

Simulation

Leetcode Link

## Problem Description

This problem requires you to simulate the game "Candy Crush". Given a 2D grid representing a candy board of size  $m \times n$ , where each cell contains an integer that represents a different type of candy, your task is to implement an algorithm that performs a series of crushing actions according to the game's rules.

The crushing rules are as follows:

When three or more candies of the same type are adjacent either vertically or horizontally, they all get crushed at once, and their positions are set to zero, indicating an empty cell.

Once candies are crushed, gravity comes into play. Any candies above the empty cells will fall down to fill the empty spaces.

New candies do not enter the board from the top. Only existing candies drop down.

The crushing and dropping process continues until no more crushing can occur—i.e., no groups of three or more adjacent candies of the same type are left.

The goal is to keep applying these rules until the board reaches a stable state with no more possible crushes. The final stable board should be returned.

## Intuition

The solution is iterative and simulates the game's mechanics step by step. Here's how the intuition unfolds:

**Initial Check for Crushes:** Start by scanning the board to find any rows or columns of three or more candies that can be crushed. We keep a boolean flag, `run`, tracking if we performed a crushing operation during the current iteration. This flag indicates whether we need to continue processing the board.

**Marking Candies for Crushing:** When we find three or more candies in a row or column that match, we mark them by negating their value. It allows us to distinguish between candies that will be crushed and the rest of the board.

**Crushing Candies:** Once we are done checking the entire board and have marked all candies to be crushed, we perform the actual crushing by replacing the marked candies (negative numbers) with zeros.

**Implementing Gravity:** After crushing, simulate gravity by letting the uncrushed candies (positive numbers) fall down to fill any gaps. This is done by moving candies down within each column, starting from the bottom up.

**Repeating the Process:** Once we've completed a crush and gravity step, we need to check if more crushes can be made because the falling candies might create new sequences of three or more matching candies. We repeat the process as long as we keep crushing candies in each iteration. The `run` flag helps us to determine this; if set to `True`, at least one crushing occurred, so we need another pass.

**Stable Board:** Finally, when no more crushes occur in a whole pass (i.e., run is False), the board is stable, and we return it.

By iterating through these steps, the given algorithm ensures that all possible crushes are made and that the board reaches a stable state according to Candy Crush rules.

Learn more about Two Pointers patterns.

### Solution Approach

The implementation of the solution involves several important steps, each taking advantage of basic algorithms and data structures.

**Marking Candies to Crush:** We loop through the entire board, searching for sequences of three or more similar candies either horizontally or vertically. We use two nested loops for this - one for rows and one for columns. When we find such a sequence, we mark the candies by negating their values using `abs(board[i][j])` to indicate these will be crushed. The use of negation allows us to retain the candy type information, which is important to check for subsequent crushes in later iterations.

**Horizontal Crushing:** This part checks for three or more adjacent candies horizontally. The following snippet from the code performs horizontal checks and marks the candies:

```
for i in range(m):
    for j in range(n - 2):
        if (board[i][j] != 0
            and abs(board[i][j]) == abs(board[i][j + 1])
            and abs(board[i][j]) == abs(board[i][j + 2])):
            run = True
            board[i][j] = board[i][j + 1] = board[i][j + 2] = -abs(board[i][j])
```

**Vertical Crushing:** Similar to the horizontal checks, we perform vertical checks as well, identifying candies that should be crushed and marking them as negative numbers. Code snippet for the vertical check:

```
for j in range(n):
    for i in range(m - 2):
        if (board[i][j] != 0
            and abs(board[i][j]) == abs(board[i + 1][j])
            and abs(board[i][j]) == abs(board[i + 2][j])):
            run = True
            board[i][j] = board[i + 1][j] = board[i + 2][j] = -abs(board[i][j])
```

**Candies Falling (Gravity):** Once all possible candies are marked, we need to let all the unmarked (positive) candies fall down. We iterate through each column starting from the bottom

row and move unmarked candies down to the 'curr' position, which represents the next available uncrushed position from the bottom. The code for simulating gravity is as follows:

```
for j in range(n):
    curr = m - 1
    for i in range(m - 1, -1, -1):
        if board[i][j] > 0:
            board[curr][j] = board[i][j]
            curr -= 1
    while curr > -1:
        board[curr][j] = 0
        curr -= 1
```

Repeating Until Stable: The outer while run: loop allows us to repeat this process until no further candies can be crushed. The run flag is set to True whenever a crush happens, prompting another iteration. When an iteration completes with no candies crushed (run is False), the board is stable, and the loop will terminate.

Returning the Result: After the loop exits, we return the stabilized board. There are no extra data structures used outside of the input board, which is mutated in place to save space, and simple variables for iteration control.

By iterating over the marked candies and applying gravity as long as there are candies to crush, the algorithm ensures an accurate portrayal of the Candy Crush mechanism.

Ready to land your dream job?

Unlock your dream job with a 2-minute evaluator for a personalized learning plan!

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using a simple 3x3 grid to demonstrate the Candy Crush algorithm.

Suppose our initial 3x3 grid is as follows:

```
1 1 1
2 2 3
3 3 2
```

Following the solution approach:

Marking Candies to Crush: First, we look for horizontal and vertical sequences of three or more similar candies to mark for crushing. We see that the top row has three 1s horizontally.

After marking, our grid looks like this:

```
-1 -1 -1
2 2 3
```

3 3 2

There are no vertical sequences to mark, so we continue to the next step.

Horizontal and Vertical Crushing: Since we've already marked the candies in step 1, we now set those positions to zero, simulating the crush.

Our grid now looks like this:

0 0 0

2 2 3

3 3 2

Candies Falling (Gravity): We apply gravity, allowing the candies to fall down to fill the empty spaces. Starting from the bottom of each column, we move positive numbers down.

After gravity, our grid becomes:

0 0 0

0 0 0

2 2 3

Now we have uncrushed candies at the bottom with zeros representing the empty cells above them.

Repeating Until Stable: We need to check again if any new sequences of three or more similar candies have formed due to gravity. Since no such sequences exist, no further action is required, and the run flag remains False.

Returning the Result: With no further crushes possible, and the board in a stable state, this is the final board state:

0 0 0

0 0 0

2 2 3

And this is the state we return as our answer, illustrating a simple run-through of the Candy Crush algorithm.

Solution Implementation

Python

Java

C++

TypeScript

from typing import List

class Solution:

def candyCrush(self, board: List[List[int]]) -> List[List[int]]:

```

# Dimensions of the board
num_rows, num_cols = len(board), len(board[0])

# Flag to indicate if we should continue crushing candies
should_crush = True

# Keep crushing candies until no more moves can be made
while should_crush:
    should_crush = False # Reset the flag for each iteration

    # Mark the candies to be crushed horizontally
    for row in range(num_rows):
        for col in range(num_cols - 2):
            candy_value = abs(board[row][col])
            # Check if three consecutive candies have the same value
            if candy_value != 0 and candy_value == abs(board[row][col + 1]) ==
abs(board[row][col + 2]):
                should_crush = True # We will need another pass after crushing
                # Mark the candies for crushing by negating their value
                board[row][col] = board[row][col + 1] = board[row][col + 2] = -candy_value

    # Mark the candies to be crushed vertically
    for col in range(num_cols):
        for row in range(num_rows - 2):
            candy_value = abs(board[row][col])
            # Check if three consecutive candies vertically have the same value
            if candy_value != 0 and candy_value == abs(board[row + 1][col]) == abs(board[row
+ 2][col]):
                should_crush = True # We will need another pass after crushing
                # Mark the candies for crushing
                board[row][col] = board[row + 1][col] = board[row + 2][col] = -candy_value

# Drop the candies to fill the empty spaces caused by crushing
if should_crush:
    for col in range(num_cols):
        # Pointer to where the next non-crushed candy will fall
        write_row = num_rows - 1
        for row in range(num_rows - 1, -1, -1):
            # If the candy is not marked for crushing, bring it down
            if board[row][col] > 0:
                board[write_row][col] = board[row][col]
                write_row -= 1
        # Fill the remaining spaces at the top with zeros
        while write_row >= 0:

```

```
board[write_row][col] = 0
write_row -= 1
```

```
# Return the modified board after all possible crushes are completed
return board
```

#### Time and Space Complexity

##### Time Complexity

The time complexity of the given code is primarily determined by the nested loops and the operations performed within them. Let's analyze it step by step.

The outer while loop that repeats until no more candies can be crushed ( $run = False$ ) will run until the board stabilizes, which relates to the dimensions of the board. In the worst case, this might be  $O(m * n)$  because each candy might need to fall all the way down the board in the most extreme case.

Inside the while loop, there are two sets of nested for loops: one for checking rows and one for checking columns for crushable candies. Both of these sets involve iterating over all elements on the board once, giving us a  $O(m * n)$  for each full scan.

Crushing candies and making them negative take  $O(1)$  time each, but since it's inside the nested loops, it doesn't add more than a constant factor to the overall complexity of scanning the board.

The final part within the while loop is another nested loop that handles gravity, making candies fall down. This, again, requires going over each column and, in the worst case, moving each candy down to the bottom, which takes  $O(m)$  for each column. Multiplying by  $n$  columns gives us  $O(m * n)$ .

Combining these, in the worst case, the while loop may run  $m * n$  times due to the possibility of each iteration only crushing one candy and making others fall one place. This is a pessimistic estimate, but it serves as an upper boundary. Thus, the time complexity of the nested loops, taken together, results in  $O((m * n) * (m * n))$  or  $O((m * n)^2)$ .

##### Space Complexity

The space complexity of the code is:

$O(1)$  for the variables  $m$ ,  $n$ ,  $run$ ,  $curr$ , and  $i$ ,  $j$  inside loops, as they do not depend on the size of the board and use a constant amount of space.

The input board is modified in-place and no additional data structures of significant size are created or used in the process, meaning we do not use additional space proportional to the size of the input.

Hence, the space complexity of the algorithm is  $O(1)$ .



Note: While the actual number of iterations needed to stabilize the board might be much less than  $m * n$ , we are considering the worst-case time complexity here for completeness.

Learn more about how to find time and space complexity quickly using problem constraints.

Discover Your Strengths and Weaknesses: Take Our 2-Minute Quiz to Tailor Your Study Plan:  
Question 1 out of 10

Which of the following problems can be solved with backtracking (select multiple)

Generating subsets

Generating random numbers

Generating permutations

Sorting integers

Recommended Readings

Tech Interview Pattern | Two Pointers Introduction

Tech Interview Pattern Two Pointers Introduction If you prefer videos here's a super quick introduction to Two Pointers 

iframe src https www youtube com embed xZ4AfXHQ1VQ title YouTube video player frameborder 0 allow accelerometer autoplay clipboard write encrypted media gyroscope picture in picture allowfullscreen

iframe div Two pointers is a common interview

LeetCode Patterns: The Smart, Data-Driven Way to Prepare for Tech Interviews

LeetCode Patterns Your Personal Dijkstra's Algorithm to Landing Your Dream Job The goal of AlgoMonster is to help you get a job in the shortest amount of time possible in a data driven way We compiled datasets of tech interview problems and broke them down by patterns This way we

Recursion Review

Recursion Recursion is one of the most important concepts in computer science Simply speaking recursion is the process of a function calling itself Using a real life analogy imagine a scenario where you invite your friends to lunch <https://algomonster.s3.us-east-2.amazonaws.com/recursion.jpg> You first

Want a Structured Path to Master System Design Too? Don't Miss This!

Coding Interview Strategies

Dive into our free, detailed pattern charts and company guides to understand what each company focuses on.

"

**ChatGPT said:**

ChatGPT

Here's the implementation of a Candy Crush simulation in p5.js, using the provided structure and explanation as a guide.

## Setup

We'll use the p5.js library to display the grid and dynamically simulate the Candy Crush algorithm. Each step will involve marking candies for crushing, performing gravity, and rendering the changes visually.

javascript

Copy code

```
let board; let rows = 9; let cols = 9; let cellSize = 50; let crushing = false;

function setup() { createCanvas(cols * cellSize, rows * cellSize); board =
createRandomBoard(rows, cols); } function draw() { background(220); drawBoard();
if (!crushing) { crushing = true; setTimeout(() => { if (!simulateCandyCrush())
noLoop(); // Stop when stable crushing = false; }, 500); } } function
createRandomBoard(rows, cols) { let candies = [1, 2, 3, 4, 5]; // Candy types let
board = []; for (let i = 0; i < rows; i++) { board[i] = []; for (let j = 0; j <
cols; j++) { board[i][j] = random(candies); } } return board; } function
drawBoard() { for (let i = 0; i < rows; i++) { for (let j = 0; j < cols; j++) { if
(board[i][j] === 0) { fill(200); // Empty cells } else { fill(board[i][j] * 50,
100, 200); // Color candy } rect(j * cellSize, i * cellSize, cellSize, cellSize);
} } } function simulateCandyCrush() { let marked = markCandies(); if (!marked)
return false; applyGravity(); return true; } function markCandies() { let marked =
false; // Horizontal check for (let i = 0; i < rows; i++) { for (let j = 0; j <
cols - 2; j++) { let candy = abs(board[i][j]); if (candy !== 0 && candy ===
abs(board[i][j + 1]) && candy === abs(board[i][j + 2])) { board[i][j] = board[i][j
+ 1] = board[i][j + 2] = -candy; marked = true; } } } // Vertical check for (let j
```

```

= 0; j < cols; j++) { for (let i = 0; i < rows - 2; i++) { let candy =
abs(board[i][j]); if (candy !== 0 && candy === abs(board[i + 1][j]) && candy ===
abs(board[i + 2][j])) { board[i][j] = board[i + 1][j] = board[i + 2][j] = -candy;
marked = true; } } } // Crush candies for (let i = 0; i < rows; i++) { for (let j
= 0; j < cols; j++) { if (board[i][j] < 0) { board[i][j] = 0; } } } return marked;
} function applyGravity() { for (let j = 0; j < cols; j++) { let emptyRow = rows -
1; for (let i = rows - 1; i >= 0; i--) { if (board[i][j] > 0) { board[emptyRow][j]
= board[i][j]; if (emptyRow !== i) board[i][j] = 0; emptyRow--; } } } }

```

## Key Features:

1. Grid Rendering:
  - The candies are represented as rectangles in a 2D grid.
  - Each candy type is color-coded for visual clarity.
2. Candy Crush Algorithm:
  - Marking: Horizontal and vertical checks identify candies to crush and mark them as 0.
  - Gravity: Candies above empty cells fall down, filling gaps.
3. Visualization:
  - Each step pauses briefly using `setTimeout` for animation.
  - The board stabilizes when no more candies can be crushed, halting the loop.

## How to Run:

1. Copy the code into the p5.js web editor: [p5.js Web Editor](#).
2. Run the sketch to see the simulation in action.

This will visually simulate the Candy Crush game mechanics in a grid. Let me know if you need additional features, such as user interaction or a scoring system!

**You said:**

its not moving grid find all similar candy to ply display scre

**ChatGPT said:**

ChatGPT

Here's an updated implementation of the Candy Crush game in p5.js that includes these features:

1. Dynamic Movement of the Grid: The grid updates dynamically when candies are crushed and fall due to gravity.
2. Score Display: A score counter increases as candies are crushed.
3. Highlight Matching Candies: A function identifies and highlights clusters of three or more matching candies.

javascript

Copy code

```
let board; let rows = 9; let cols = 9; let cellSize = 50; let score = 0; let
crushing = false; function setup() { createCanvas(cols * cellSize, rows * cellSize
+ 50); // Extra space for score board = createRandomBoard(rows, cols); } function
draw() { background(220); drawBoard(); displayScore(); if (!crushing) { crushing =
true; setTimeout(() => { if (!simulateCandyCrush()) noLoop(); // Stop when stable
crushing = false; }, 500); } } function createRandomBoard(rows, cols) { let
candies = [1, 2, 3, 4, 5]; // Candy types let board = []; for (let i = 0; i <
rows; i++) { board[i] = []; for (let j = 0; j < cols; j++) { board[i][j] =
random(candies); } } return board; } function drawBoard() { for (let i = 0; i <
rows; i++) { for (let j = 0; j < cols; j++) { if (board[i][j] === 0) { fill(200);
// Empty cells } else { fill(board[i][j] * 50, 100, 200); // Color candy } rect(j
* cellSize, i * cellSize, cellSize, cellSize); } } } function displayScore() {
fill(0); textSize(20); textAlign(CENTER, CENTER); text(`Score: ${score}`, width /
2, rows * cellSize + 25); } function simulateCandyCrush() { let marked =
markCandies(); if (!marked) return false; applyGravity(); return true; } function
markCandies() { let marked = false; // Horizontal check for (let i = 0; i < rows;
i++) { for (let j = 0; j < cols - 2; j++) { let candy = abs(board[i][j]); if
(candy !== 0 && candy === abs(board[i][j + 1]) && candy === abs(board[i][j + 2]))
{ board[i][j] = board[i][j + 1] = board[i][j + 2] = -candy; marked = true; } } }
// Vertical check for (let j = 0; j < cols; j++) { for (let i = 0; i < rows - 2;
i++) { let candy = abs(board[i][j]); if (candy !== 0 && candy === abs(board[i +
1][j]) && candy === abs(board[i + 2][j])) { board[i][j] = board[i + 1][j] =
board[i + 2][j] = -candy; marked = true; } } } // Crush candies and update score
for (let i = 0; i < rows; i++) { for (let j = 0; j < cols; j++) { if (board[i][j]
```

```

< 0) { board[i][j] = 0; score += 10; // Add 10 points per crushed candy } } }
return marked; } function applyGravity() { for (let j = 0; j < cols; j++) { let
emptyRow = rows - 1; for (let i = rows - 1; i >= 0; i--) { if (board[i][j] > 0) {
board[emptyRow][j] = board[i][j]; if (emptyRow !== i) board[i][j] = 0; emptyRow--;
} } } }

```

## Key Features:

1. Dynamic Grid Updates:
  - Candies are crushed and replaced with zeros.
  - Remaining candies fall to fill empty spaces using gravity.
2. Score Tracking:
  - Each crushed candy adds 10 points to the score.
  - The score is displayed at the bottom of the canvas.
3. Stop Simulation When Stable:
  - The game stops when no more matches are possible.
  - Uses `noLoop()` to halt the animation.
4. Candy Color Coding:
  - Each candy type is assigned a unique color for better visual distinction.

---

## How to Run:

1. Open the p5.js Web Editor: [p5.js Web Editor](#).
2. Copy and paste the code into the editor.
3. Press the Play button to see the simulation in action.

Let me know if you'd like additional features such as interactive candy swapping or bonus points for longer matches!

### You said:

show movement in canvas

### ChatGPT said:

ChatGPT

To visually show the movement of candies falling in the canvas, we'll need to animate the process of candies "falling" into place. Here's the updated implementation that animates the movement:

## Key Updates

1. Animation for Falling Candies:
  - Use an array to track the current position of each candy during the animation.
  - Smoothly update their positions frame by frame.
2. Gravity Simulation with Animation:
  - Gradually move candies to their new positions using interpolation.
  - Update the board once all candies have settled.

## Full Code

javascript

Copy code

```
let board; let rows = 9; let cols = 9; let cellSize = 50; let score = 0; let
animations = []; // To track falling candies let animationSpeed = 5; // Pixels per
frame function setup() { createCanvas(cols * cellSize, rows * cellSize + 50); //
Extra space for score board = createRandomBoard(rows, cols); } function draw() {
background(220); drawBoard(); displayScore(); if (animations.length > 0) {
animateCandies(); } else if (!simulateCandyCrush()) { noLoop(); // Stop the game
when stable } } function createRandomBoard(rows, cols) { let candies = [1, 2, 3,
4, 5]; // Candy types let board = []; for (let i = 0; i < rows; i++) { board[i] =
[]; for (let j = 0; j < cols; j++) { board[i][j] = random(candies); } } return
board; } function drawBoard() { for (let i = 0; i < rows; i++) { for (let j = 0; j
< cols; j++) { if (board[i][j] === 0) { fill(200); // Empty cells } else {
fill(board[i][j] * 50, 100, 200); // Color candy } rect(j * cellSize, i *
cellSize, cellSize, cellSize); } } // Draw animations for (let anim of animations)
{ fill(anim.type * 50, 100, 200); rect(anim.x * cellSize, anim.y, cellSize,
cellSize); } } function displayScore() { fill(0); textSize(20); textAlign(CENTER,
CENTER); text(`Score: ${score}`, width / 2, rows * cellSize + 25); } function
simulateCandyCrush() { let marked = markCandies(); if (!marked) return false;
startCandyAnimations(); return true; } function markCandies() { let marked =
false; // Horizontal check for (let i = 0; i < rows; i++) { for (let j = 0; j <
cols - 2; j++) { let candy = abs(board[i][j]); if (candy !== 0 && candy ===
```

```

abs(board[i][j + 1]) && candy === abs(board[i][j + 2])) { board[i][j] = board[i][j
+ 1] = board[i][j + 2] = -candy; marked = true; } } // Vertical check for (let j
= 0; j < cols; j++) { for (let i = 0; i < rows - 2; i++) { let candy =
abs(board[i][j]); if (candy !== 0 && candy === abs(board[i + 1][j]) && candy ===
abs(board[i + 2][j])) { board[i][j] = board[i + 1][j] = board[i + 2][j] = -candy;
marked = true; } } } // Crush candies and update score for (let i = 0; i < rows;
i++) { for (let j = 0; j < cols; j++) { if (board[i][j] < 0) { board[i][j] = 0;
score += 10; // Add 10 points per crushed candy } } } return marked; } function
startCandyAnimations() { animations = []; for (let j = 0; j < cols; j++) { let
emptyRow = rows - 1; for (let i = rows - 1; i >= 0; i--) { if (board[i][j] > 0) {
if (emptyRow !== i) { animations.push({ type: board[i][j], // Candy type x: j, y:
i * cellSize, targetY: emptyRow * cellSize, }); board[emptyRow][j] = board[i][j];
board[i][j] = 0; } emptyRow--; } } } } function animateCandies() { let
stillAnimating = false; for (let anim of animations) { if (anim.y < anim.targetY)
{ anim.y += animationSpeed; stillAnimating = true; } else { anim.y = anim.targetY;
// Snap to target to avoid overshooting } } // If no more candies are animating,
clear the animation array if (!stillAnimating) animations = []; }

```

---

## Key Changes

1. Candy Animation Logic:
    - Animations are tracked using the `animations` array.
    - Each falling candy is stored with its current position and target position.
    - `animateCandies()` updates the position of falling candies smoothly.
  2. Dynamic Drawing:
    - During the animation, candies in motion are drawn separately.
    - Once a candy reaches its target, it snaps into place.
  3. Game Stabilization:
    - The game continues crushing and animating until the grid reaches a stable state.
-

## How It Works

1. Mark Candies:
    - Identify candies to crush and set their positions to 0.
  2. Apply Gravity with Animation:
    - Calculate which candies need to fall and initialize animations for them.
    - Gradually move candies to their target positions.
  3. Repeat Until Stable:
    - After each animation, check for new matches and repeat the process.
- 

## How to Run

1. Copy and paste this code into the p5.js Web Editor.
2. Press Play.
3. Watch the candies animate as they fall and new matches are created.

Let me know if you'd like further enhancements!

**You've hit the Free plan limit for GPT-4o.**

Responses will use another model until your limit resets after 7:08 PM.

[Get Plus](#)

ChatGPT can make mistakes. Check important info.

?



## Chat history

[ChatGPT](#)

[ChatGPT](#)

[Explore GPTs](#)

### Today

[Game Over Detection](#)

[Candy Crush Clone p5.js](#)

### Yesterday

[Vijaykanth vs Terrorists Action](#)

### Previous 7 Days

[Dystopian Video Game Battle](#)

[Dhanush Nayanthara Army Scene](#)

Upgrade plan More access to the best models

ChatGPT

Share



**You said:**

```
// Grid and candy variables
```

```
let grid;
```

```
let cols = 8; // Number of columns
```

```
let rows = 8; // Number of rows
```

```
let tileSize = 60; // Size of each tile
```

```
let candyColors = []; // Gradient colors for different candies
```

```
let sparkleParticles = [];
```

```
let sparkleFrames = 10; // Sparkle animation frames
```

```
function setup() {
```

```
  createCanvas(cols * tileSize, rows * tileSize);
```

```
  // Generate gradient colors for candies
```

```
  for (let i = 0; i < 5; i++) {
```

```
    let color1 = color(random(50, 200), random(50, 200), random(50, 200));
```

```

    let color2 = color(random(200, 255), random(200, 255), random(200, 255));
    candyColors.push([color1, color2]);
  }

  // Initialize the grid
  grid = createGrid(cols, rows);
}

function draw() {
  background(30);

  // Draw the grid
  drawGrid();

  // Handle sparkles
  for (let i = sparkleParticles.length - 1; i >= 0; i--) {
    let sparkle = sparkleParticles[i];
    sparkle.display();
    sparkle.update();

    // Remove particle when it's finished
    if (sparkle.finished) {
      sparkleParticles.splice(i, 1);
    }
  }
}

// Create a grid of random candies
function createGrid(cols, rows) {
  let grid = [];
  for (let y = 0; y < rows; y++) {
    let row = [];
    for (let x = 0; x < cols; x++) {
      row.push(floor(random(candyColors.length))); // Random candy type
    }
    grid.push(row);
  }
  return grid;
}

// Draw the grid
function drawGrid() {
  for (let y = 0; y < rows; y++) {
    for (let x = 0; x < cols; x++) {

```

```

    let candyType = grid[y][x];
    if (candyType !== null) {
      drawCandy(x * tileSize, y * tileSize, candyType);
    }
  }
}

```

// Draw a single candy with gradient

```

function drawCandy(x, y, type) {
  let gradient = candyColors[type];
  let color1 = gradient[0];
  let color2 = gradient[1];

  // Create a gradient effect
  for (let i = 0; i < tileSize - 20; i++) {
    let inter = map(i, 0, tileSize - 20, 0, 1);
    let col = lerpColor(color1, color2, inter);
    stroke(col);
    line(x + 10, y + 10 + i, x + tileSize - 10, y + 10 + i);
  }
}

```

// Add a highlight

```

noStroke();
fill(255, 255, 255, 50);
ellipse(x + tileSize / 2, y + tileSize / 3, tileSize / 3, tileSize / 4);
}

```

// Check for matches and mark candies for removal

```

function checkMatches() {
  let matches = [];

  // Check rows for matches
  for (let y = 0; y < rows; y++) {
    for (let x = 0; x < cols - 2; x++) {
      let candyType = Math.abs(grid[y][x]);
      if (
        candyType !== null &&
        candyType === Math.abs(grid[y][x + 1]) &&
        candyType === Math.abs(grid[y][x + 2])
      ) {
        matches.push({ x, y });
        matches.push({ x: x + 1, y });
        matches.push({ x: x + 2, y });
      }
    }
  }
}

```

```

        grid[y][x] = grid[y][x + 1] = grid[y][x + 2] = -candyType;
    }
}

// Check columns for matches
for (let x = 0; x < cols; x++) {
    for (let y = 0; y < rows - 2; y++) {
        let candyType = Math.abs(grid[y][x]);
        if (
            candyType !== null &&
            candyType === Math.abs(grid[y + 1][x]) &&
            candyType === Math.abs(grid[y + 2][x])
        ) {
            matches.push({ x, y });
            matches.push({ x, y: y + 1 });
            matches.push({ x, y: y + 2 });
            grid[y][x] = grid[y + 1][x] = grid[y + 2][x] = -candyType;
        }
    }
}

return matches;
}

// Collapse the grid after matches
function collapseGrid() {
    for (let x = 0; x < cols; x++) {
        let writeRow = rows - 1;
        for (let y = rows - 1; y >= 0; y--) {
            if (grid[y][x] > 0) {
                grid[writeRow][x] = grid[y][x];
                writeRow--;
            }
        }
        while (writeRow >= 0) {
            grid[writeRow][x] = floor(random(candyColors.length));
            writeRow--;
        }
    }
}

// Trigger sparkle animation
function triggerSparkle(matches) {

```

```

matches.forEach(({ x, y }) => {
  for (let i = 0; i < 20; i++) {
    sparkleParticles.push(
      new Sparkle(x * tileSize + tileSize / 2, y * tileSize + tileSize / 2)
    );
  }
});
}

```

// Create sparkle animation

```

class Sparkle {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.life = sparkleFrames;
    this.finished = false;
  }

```

```

  update() {
    this.life--;
    if (this.life <= 0) {
      this.finished = true;
    }
  }
}

```

```

  display() {
    noStroke();
    fill(255, random(150, 255), random(150, 255), this.life * 25);
    ellipse(
      this.x + random(-10, 10),
      this.y + random(-10, 10),
      random(5, 10)
    );
  }
}

```

// Handle mouse click

```

function mousePressed() {
  let matches = checkMatches();
  if (matches.length > 0) {
    triggerSparkle(matches);
  }
}

```

// Remove matched candies

```

matches.forEach(({ x, y }) => {

```

```

        grid[y][x] = null;
    });

    collapseGrid();
}
} if game is over tell its over or time completed

```

ChatGPT said:

ChatGPT

To detect when the game is over, you can add logic to check for possible matches or a time-based condition. Here's how you can incorporate the functionality:

## Add a Timer

If the game is time-based, you can set a `timeRemaining` variable and decrement it in the `draw()` function. When it reaches zero, display a "Game Over" message.

## Check for Possible Moves

If the game ends when no more matches are possible, you can add a function to scan the grid for potential moves.

Here's how you can integrate these features:

## Updated Code

javascript

Copy code

```

let grid; let cols = 8; // Number of columns let rows = 8; // Number of rows let
tileSize = 60; // Size of each tile let candyColors = []; // Gradient colors for
different candies let sparkleParticles = []; let sparkleFrames = 10; // Sparkle
animation frames let timeRemaining = 60; // Game time in seconds let gameOver =
false; function setup() { createCanvas(cols * tileSize, rows * tileSize); //
Generate gradient colors for candies for (let i = 0; i < 5; i++) { let color1 =
color(random(50, 200), random(50, 200), random(50, 200)); let color2 =
color(random(200, 255), random(200, 255), random(200, 255));
candyColors.push([color1, color2]); } // Initialize the grid grid =

```

```

createGrid(cols, rows); } function draw() { if (gameOver) { displayGameOver();
return; } background(30); // Update and display the timer updateTimer(); // Draw
the grid drawGrid(); // Handle sparkles for (let i = sparkleParticles.length - 1;
i >= 0; i--) { let sparkle = sparkleParticles[i]; sparkle.display();
sparkle.update(); // Remove particle when it's finished if (sparkle.finished) {
sparkleParticles.splice(i, 1); } } // Check if there are possible moves if
(!hasPossibleMoves()) { gameOver = true; } } // Create a grid of random candies
function createGrid(cols, rows) { let grid = []; for (let y = 0; y < rows; y++) {
let row = []; for (let x = 0; x < cols; x++) {
row.push(floor(random(candyColors.length))); // Random candy type }
grid.push(row); } return grid; } // Update and display the timer function
updateTimer() { fill(255); textSize(20); textAlign(CENTER); text(`Time:
${timeRemaining}`, width / 2, 30); if (frameCount % 60 === 0 && timeRemaining > 0)
{ timeRemaining--; } if (timeRemaining <= 0) { gameOver = true; } } // Display the
"Game Over" screen function displayGameOver() { background(0, 0, 0, 200);
fill(255); textSize(40); textAlign(CENTER, CENTER); text("Game Over", width / 2,
height / 2); textSize(20); text("Refresh to play again!", width / 2, height / 2 +
40); } // Check if there are possible moves function hasPossibleMoves() { //
Horizontal check for (let y = 0; y < rows; y++) { for (let x = 0; x < cols - 1;
x++) { // Swap candies and check for matches swapCandies(x, y, x + 1, y); if
(checkMatches().length > 0) { swapCandies(x, y, x + 1, y); // Undo swap return
true; } swapCandies(x, y, x + 1, y); // Undo swap } } // Vertical check for (let x
= 0; x < cols; x++) { for (let y = 0; y < rows - 1; y++) { swapCandies(x, y, x, y
+ 1); if (checkMatches().length > 0) { swapCandies(x, y, x, y + 1); // Undo swap
return true; } swapCandies(x, y, x, y + 1); // Undo swap } } return false; } //
Swap two candies in the grid function swapCandies(x1, y1, x2, y2) { let temp =
grid[y1][x1]; grid[y1][x1] = grid[y2][x2]; grid[y2][x2] = temp; } // Handle mouse
click function mousePressed() { if (gameOver) return; let matches =
checkMatches(); if (matches.length > 0) { triggerSparkle(matches); // Remove

```

```
matched candies matches.forEach(({ x, y }) => { grid[y][x] = null; });  
collapseGrid(); } }
```

## How It Works:

1. Timer (`timeRemaining`): Counts down and ends the game when it reaches zero.
2. Game Over (`gameOver`): Stops the game and displays a "Game Over" screen.
3. Possible Moves Check (`hasPossibleMoves`): Swaps adjacent candies temporarily to check if a match can be made. If no moves are possible, the game ends.

You can adjust the game logic to suit your needs, such as adding a scoring system or difficulty levels.

### You said:

723. Candy Crush

#### 1. Question

This question is about implementing a basic elimination algorithm for Candy Crush.

Given a 2D integer array `board` representing the grid of candy, different positive integers `board[i][j]` represent different types of candies. A value of `board[i][j] = 0` represents that the cell at position  $(i, j)$  is empty. The given board represents the state of the game following the player's move. Now, you need to restore the board to a stable state by crushing candies according to the following rules:

If three or more candies of the same type are adjacent vertically or horizontally, "crush" them all at the same time - these positions become empty.

After crushing all candies simultaneously, if an empty space on the board has candies on top of itself, then these candies will drop until they hit a candy or bottom at the same time. (No new candies will drop outside the top boundary.)

After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.

If there does not exist more candies that can be crushed (ie. the board is stable), then return the current board.

You need to perform the above rules until the board becomes stable, then return the current board.

Example 1:



Copy

Input:

board =

```
[[110,5,112,113,114],[210,211,5,213,214],[310,311,3,313,314],[410,411,412,5,414],[5,1,512,3,3],[610,4,1,613,614],[710,1,2,713,714],[810,1,2,1,1],[1,1,2,2,2],[4,1,4,4,1014]]
```

Output:

```
[[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[110,0,0,0,114],[210,0,0,0,214],[310,0,0,113,314],[410,0,0,213,414],[610,211,112,313,614],[710,311,412,613,714],[810,411,512,713,1014]]
```

Explanation:

Note:

The length of board will be in the range [3, 50].

The length of board[i] will be in the range [3, 50].

Each board[i][j] will initially start as an integer in the range [1, 2000].

## 2. Implementation

### (1) Two Pointers

思路: Two Pointers的思想指体现在Candy Drop的时候,对每一列,将非零的数往下drop,而将0往上移。

Copy

```
class Solution {
    public int[][] candyCrush(int[][] board) {
        if (board == null || board.length == 0) {
            return new int[0][0];
        }
        int m = board.length, n = board[0].length;

        while (true) {
            Set<Coordinate> set = new HashSet<>();

            // Find index where candy can be crushed
            for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                    if (board[i][j] == 0) continue;

                    int top = i, bottom = i, left = j, right = j;
```

```

        while (top >= 0 && top > i - 3 && board[top][j] == board[i][j]) --top;
        while (bottom < m && bottom < i + 3 && board[bottom][j] == board[i][j]) ++bottom;
        while (left >= 0 && left > j - 3 && board[i][left] == board[i][j]) --left;
        while (right < n && right < j + 3 && board[i][right] == board[i][j]) ++right;

        if (right - left > 3 || bottom - top > 3) {
            set.add(new Coordinate(i, j));
        }
    }
}

// Board is in stable state
if (set.isEmpty()) break;

// Mark index as 0
for (Coordinate c : set) {
    board[c.x][c.y] = 0;
}

// Use Two Pointers to move zero to top, and keep number in order in column
for (int col = 0; col < n; col++) {
    int index = m - 1;
    for (int row = m - 1; row >= 0; row--) {
        if (board[row][col] != 0) {
            swap(board, index, row, col);
            --index;
        }
    }
}
return board;
}

public void swap(int[][] board, int row1, int row2, int col) {
    if (row1 == row2) return;
    int temp = board[row1][col];
    board[row1][col] = board[row2][col];
    board[row2][col] = temp;
}

class Coordinate {
    int x;
    int y;
}

```

```

    public Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

} follow this but give correctly in p5js

ChatGPT said:

ChatGPT

Here's how you can translate the given Candy Crush elimination algorithm into p5.js. This implementation identifies and "crushes" candies in a grid based on the same principles as the Java solution:

## Code Implementation in p5.js

javascript

Copy code

```

let board; // The grid representing the candy board let rows = 10; // Number of
rows let cols = 10; // Number of columns let tileSize = 50; // Size of each tile
function setup() { createCanvas(cols * tileSize, rows * tileSize); board =
createBoard(rows, cols); // Initialize the grid noLoop(); // Run the simulation
step by step } function draw() { background(30); drawBoard(); // Draw the current
state of the board if (!candyCrush(board)) { console.log("Stable board reached!");
noLoop(); // Stop the simulation when the board is stable } } // Create a random
board function createBoard(rows, cols) { let grid = []; for (let i = 0; i < rows;
i++) { let row = []; for (let j = 0; j < cols; j++) { row.push(floor(random(1,
6))); // Random candy types (1-5) } grid.push(row); } return grid; } // Draw the
board function drawBoard() { for (let i = 0; i < rows; i++) { for (let j = 0; j <
cols; j++) { let value = board[i][j]; fill(value === 0 ? 50 : value * 40, 150,
200); // Color candies rect(j * tileSize, i * tileSize, tileSize, tileSize); //
Display candy type for debugging fill(255); textSize(16); textAlign(CENTER,
CENTER); if (value > 0) text(value, j * tileSize + tileSize / 2, i * tileSize +
tileSize / 2); } } } // Perform candy crushing function candyCrush(board) { let
rows = board.length; let cols = board[0].length; let toCrush = new Set(); //
Identify candies to crush for (let i = 0; i < rows; i++) { for (let j = 0; j <

```

```

cols; j++) { let value = board[i][j]; if (value === 0) continue; // Check
horizontal match if (j <= cols - 3 && value === board[i][j + 1] && value ===
board[i][j + 2]) { toCrush.add(`${i},${j}`); toCrush.add(`${i},${j + 1}`);
toCrush.add(`${i},${j + 2}`); } // Check vertical match if (i <= rows - 3 && value
=== board[i + 1][j] && value === board[i + 2][j]) { toCrush.add(`${i},${j}`);
toCrush.add(`${i + 1},${j}`); toCrush.add(`${i + 2},${j}`); } } } // If no candies
to crush, board is stable if (toCrush.size === 0) return false; // Mark candies as
crushed (set to 0) for (let cell of toCrush) { let [x, y] =
cell.split(",").map(Number); board[x][y] = 0; } // Apply gravity to the board
applyGravity(board); return true; // Board is not stable yet } // Apply gravity:
candies fall down into empty spaces function applyGravity(board) { let rows =
board.length; let cols = board[0].length; for (let j = 0; j < cols; j++) { let
index = rows - 1; // Start from the bottom of the column for (let i = rows - 1; i
>= 0; i--) { if (board[i][j] > 0) { // Move candy to the bottom-most empty space
board[index][j] = board[i][j]; if (index !== i) board[i][j] = 0; index--; } } } }

```

---

## Explanation of the Code:

1. Board Initialization:
    - A grid of random numbers (1–5) is created to represent different candy types.
  2. Candy Crushing:
    - The `candyCrush` function scans the board to find horizontal or vertical matches of three or more candies and marks them as crushed by setting their values to 0.
  3. Gravity Application:
    - The `applyGravity` function simulates the effect of gravity. Non-zero candies drop to the bottom of each column, and empty spaces (zeros) move to the top.
  4. Board Stability:
    - The simulation continues until no more matches can be found, at which point the board is considered stable.
  5. Visualization:
    - The `drawBoard` function renders the board using rectangles, with unique colors representing candy types.
-

## How to Use:

1. Run the sketch in a p5.js environment.
2. The program will simulate Candy Crush and print "Stable board reached!" in the console when no more matches are possible.

You can adjust the board size, candy types, or visualization as needed.

**You said:**

show the animation

**ChatGPT said:**

ChatGPT

To add animation to the Candy Crush simulation, we need to visualize the crushing of candies and their subsequent falling over several frames. Here's an updated version of the code with animations integrated into the simulation:

## Animated Candy Crush in p5.js

javascript

Copy code

```
let board; // The grid representing the candy board let rows = 10; // Number of
rows let cols = 10; // Number of columns let tileSize = 50; // Size of each tile
let animating = false; // Flag to indicate if an animation is in progress let
animationSteps = []; // Holds intermediate steps for the animation function setup()
{ createCanvas(cols * tileSize, rows * tileSize); board = createBoard(rows, cols);
// Initialize the grid frameRate(10); // Set frame rate for smooth animations }
function draw() { background(30); drawBoard(); // Draw the current state of the
board if (!animating) { if (!candyCrush(board)) { console.log("Stable board
reached!"); noLoop(); // Stop the simulation when the board is stable } else { //
Start animation steps animating = true; animationSteps = generateAnimationSteps();
} } else { if (animationSteps.length > 0) { let step = animationSteps.shift();
applyAnimationStep(step); // Apply one animation step } else { animating = false;
// Animation complete } } } // Create a random board function createBoard(rows,
cols) { let grid = []; for (let i = 0; i < rows; i++) { let row = []; for (let j =
0; j < cols; j++) { row.push(floor(random(1, 6))); // Random candy types (1-5) }
```

```

grid.push(row); } return grid; } // Draw the board function drawBoard() { for (let
i = 0; i < rows; i++) { for (let j = 0; j < cols; j++) { let value = board[i][j];
fill(value === 0 ? 50 : value * 40, 150, 200); // Color candies rect(j * tileSize,
i * tileSize, tileSize, tileSize); // Display candy type for debugging fill(255);
textSize(16); textAlign(CENTER, CENTER); if (value > 0) text(value, j * tileSize +
tileSize / 2, i * tileSize + tileSize / 2); } } } // Perform candy crushing
function candyCrush(board) { let rows = board.length; let cols = board[0].length;
let toCrush = new Set(); // Identify candies to crush for (let i = 0; i < rows;
i++) { for (let j = 0; j < cols; j++) { let value = board[i][j]; if (value === 0)
continue; // Check horizontal match if (j <= cols - 3 && value === board[i][j + 1]
&& value === board[i][j + 2]) { toCrush.add(`${i},${j}`); toCrush.add(`${i},${j +
1}`); toCrush.add(`${i},${j + 2}`); } // Check vertical match if (i <= rows - 3 &&
value === board[i + 1][j] && value === board[i + 2][j]) {
toCrush.add(`${i},${j}`); toCrush.add(`${i + 1},${j}`); toCrush.add(`${i +
2},${j}`); } } } // If no candies to crush, board is stable if (toCrush.size === 0)
return false; // Mark candies as crushed (set to 0) for (let cell of toCrush) {
let [x, y] = cell.split(",").map(Number); board[x][y] = 0; } // Apply gravity to
the board applyGravity(board); return true; // Board is not stable yet } // Apply
gravity: candies fall down into empty spaces function applyGravity(board) { let
rows = board.length; let cols = board[0].length; for (let j = 0; j < cols; j++) {
let index = rows - 1; // Start from the bottom of the column for (let i = rows -
1; i >= 0; i--) { if (board[i][j] > 0) { // Move candy to the bottom-most empty
space board[index][j] = board[i][j]; if (index !== i) board[i][j] = 0; index--; }
} } } // Generate animation steps for candy drops function generateAnimationSteps()
{ let steps = []; let tempBoard = board.map(row => row.slice()); for (let j = 0; j
< cols; j++) { let colSteps = []; let candies = []; // Extract candies for this
column for (let i = 0; i < rows; i++) { if (tempBoard[i][j] > 0) {
candies.push(tempBoard[i][j]); tempBoard[i][j] = 0; // Set to 0 for visualization
} } // Add candies falling step by step for (let i = rows - 1, candyIdx =
candies.length - 1; i >= 0 && candyIdx >= 0; i--, candyIdx--) { colSteps.push({

```

```
row: i, col: j, value: candies[candyIdx] }); } steps =
steps.concat(colSteps.reverse()); } return steps; } // Apply a single animation
step function applyAnimationStep(step) { let { row, col, value } = step;
board[row][col] = value; // Drop the candy into the position }
```

---

## Key Features

1. Animation Steps:
    - The `generateAnimationSteps` function creates intermediate steps for candies falling down column by column.
  2. Candy Drops:
    - The `applyAnimationStep` function processes each animation step, dropping candies to their new positions over multiple frames.
  3. Smooth Animations:
    - `frameRate(10)` slows the simulation down to visualize the process clearly.
  4. Stable State:
    - Once the board reaches a stable state, the animation stops, and the final state is displayed.
- 

## How It Works

- Run the code in a p5.js editor.
- The board will crush candies, apply gravity, and animate candy drops.
- It will stop when no more matches can be found, leaving the board in a stable state.

Let me know if you'd like further adjustments or explanations!

**You've hit the Free plan limit for GPT-4o.**

Responses will use another model until your limit resets after 7:08 PM.

Get Plus

ChatGPT can make mistakes. Check important info.

?