**HashMapper: Code Documentation**

This documentation provides a breakdown of each file, method, and important code section for the HashMapper project. Methods are split if very long and followed by detailed descriptions.

# app.py

Methods and classes defined in this file are shown below.

## generate\_fingerprint (Split)

def generate\_fingerprint(self, text, size, hash\_function, salt\_level, smooth\_radius):  
 """  
 Generate fingerprint using Java code  
 Returns: (raw\_image\_bytes, enhanced\_image\_bytes, stats\_dict)  
 """  
 text\_path = None  
 raw\_output = None  
 enhanced\_output = None  
 stats\_output = None  
 temp\_dir = None  
  
 try:  
 # Create a temporary directory to store all files  
 temp\_dir = tempfile.mkdtemp()  
 logger.debug(f"Created temporary directory: {temp\_dir}")

\*\*Part 1 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

# Write text to a temporary file  
 text\_path = os.path.join(temp\_dir, 'input.txt')  
 with open(text\_path, 'w', encoding='utf-8') as text\_file:  
 text\_file.write(text)  
 logger.debug(f"Wrote input text to: {text\_path} (length: {len(text)})")  
   
 # Define output paths  
 raw\_output = os.path.join(temp\_dir, "raw\_output.png")  
 enhanced\_output = os.path.join(temp\_dir, "enhanced\_output.png")  
 stats\_output = os.path.join(temp\_dir, "stats\_output.json")  
   
 logger.debug(f"Working directory: {os.getcwd()}")  
 logger.debug(f"Output files: {raw\_output}, {enhanced\_output}, {stats\_output}")

\*\*Part 2 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

# Get the path to the java directory  
 java\_dir = os.path.join(os.getcwd(), 'java')  
 logger.debug(f"Java directory: {java\_dir}")  
   
 # Build the command - USING HASHMAP EXPERIMENT RUNNER  
 cmd = [  
 "java",  
 "-Djava.awt.headless=true", # Enable headless mode for server environments  
 "-cp", f"{java\_dir}:lib/\*", # Include java directory and all JARs in lib  
 "HashMapExperimentRunner", # The main class with main method  
 "--text-file", text\_path,  
 "--size", str(size),  
 "--hash-function", hash\_function,  
 "--salt-level", str(salt\_level),  
 "--smooth-radius", str(smooth\_radius),

\*\*Part 3 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

"--raw-output", raw\_output,  
 "--enhanced-output", enhanced\_output,  
 "--stats-output", stats\_output  
 ]  
   
 # For Windows, use semicolons instead of colons in classpath  
 if os.name == 'nt':  
 cmd[3] = f"{java\_dir};lib/\*"  
   
 logger.debug(f"Executing command: {' '.join(cmd)}")  
   
 # Run the Java process  
 result = subprocess.run(  
 cmd,  
 capture\_output=True,

\*\*Part 4 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

text=True,  
 timeout=60 # Increase timeout for large inputs  
 )  
   
 logger.debug(f"Java process completed with return code: {result.returncode}")  
 logger.debug(f"stdout: {result.stdout}")  
 logger.debug(f"stderr: {result.stderr}")  
   
 if result.returncode != 0:  
 raise Exception(f"Java process failed: {result.stderr}")  
   
 # List all files in the temp directory for debugging  
 logger.debug(f"Files in temp directory: {os.listdir(temp\_dir)}")  
   
 # Check if output files exist

\*\*Part 5 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

for file\_path in [raw\_output, enhanced\_output, stats\_output]:  
 if not os.path.exists(file\_path):  
 raise FileNotFoundError(f"Output file not found: {file\_path}")  
 logger.debug(f"File exists: {file\_path}, size: {os.path.getsize(file\_path)} bytes")  
   
 # Read the output files  
 with open(raw\_output, 'rb') as f:  
 raw\_bytes = f.read()  
 logger.debug(f"Read raw\_output file: {len(raw\_bytes)} bytes")  
   
 with open(enhanced\_output, 'rb') as f:  
 enhanced\_bytes = f.read()  
 logger.debug(f"Read enhanced\_output file: {len(enhanced\_bytes)} bytes")  
   
 # Read and parse the JSON stats file with careful error handling

\*\*Part 6 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

try:  
 with open(stats\_output, 'r') as f:  
 stats\_content = f.read()  
 logger.debug(f"Stats file content: {stats\_content}")  
   
 # Validate JSON before parsing  
 stats = json.loads(stats\_content)  
 logger.debug(f"Parsed stats: {stats}")  
 except json.JSONDecodeError as e:  
 logger.error(f"JSON parsing error: {e}")  
 logger.error(f"Content that failed to parse: {stats\_content}")  
 # Provide fallback stats  
 stats = {  
 "text\_length": len(text),  
 "hash\_function": hash\_function,

\*\*Part 7 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

"salt\_level": salt\_level,  
 "smooth\_radius": smooth\_radius,  
 "error": "Failed to parse stats JSON"  
 }  
   
 return raw\_bytes, enhanced\_bytes, stats  
   
 except Exception as e:  
 logger.error(f"Error in generate\_fingerprint: {str(e)}")  
 logger.error(traceback.format\_exc())  
 raise  
   
 finally:  
 # Clean up temporary files  
 for file\_path in [text\_path, raw\_output, enhanced\_output, stats\_output]:

\*\*Part 8 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

if file\_path and os.path.exists(file\_path):  
 try:  
 os.remove(file\_path)  
 logger.debug(f"Deleted file: {file\_path}")  
 except Exception as e:  
 logger.error(f"Failed to delete {file\_path}: {str(e)}")  
   
 # Remove the temporary directory  
 if temp\_dir and os.path.exists(temp\_dir):  
 try:  
 shutil.rmtree(temp\_dir)  
 logger.debug(f"Deleted temporary directory: {temp\_dir}")  
 except Exception as e:  
 logger.error(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")

\*\*Part 9 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

## run\_experiment (Split)

def run\_experiment(self, experiment\_type):  
 """  
 Run HashMap experiment and return the visualization  
 Returns: base64 encoded image  
 """  
 output\_file = None  
 temp\_dir = None  
   
 try:  
 # Create a temporary directory  
 temp\_dir = tempfile.mkdtemp()  
 logger.debug(f"Created temporary directory: {temp\_dir}")  
   
 output\_file = os.path.join(temp\_dir, f"{experiment\_type}\_output.png")  
 logger.debug(f"Output file will be: {output\_file}")

\*\*Part 1 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

# Get the path to the java directory  
 java\_dir = os.path.join(os.getcwd(), 'java')  
 logger.debug(f"Java directory: {java\_dir}")  
   
 # Convert camelCase experiment type to snake\_case for Java  
 java\_experiment\_type = ""  
 if experiment\_type == "hashFunction":  
 java\_experiment\_type = "hash\_function"  
 elif experiment\_type == "textFingerprint":  
 java\_experiment\_type = "text\_fingerprint"  
 else:  
 # Simple conversion for other types (already snake\_case)  
 java\_experiment\_type = experiment\_type

\*\*Part 2 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

logger.debug(f"Converted experiment type from '{experiment\_type}' to '{java\_experiment\_type}' for Java")  
   
 # Use HashMapExperimentRunner for running experiments  
 experiment\_cmd = [  
 "java",  
 "-Djava.awt.headless=true",  
 "-cp", f"{java\_dir}:lib/\*",  
 "HashMapExperimentRunner", # The main class with main method  
 "--type", java\_experiment\_type,  
 "--output", output\_file  
 ]  
   
 # For Windows, use semicolons instead of colons in classpath  
 if os.name == 'nt':  
 experiment\_cmd[3] = f"{java\_dir};lib/\*"

\*\*Part 3 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

logger.debug(f"Running experiment: {' '.join(experiment\_cmd)}")  
 exp\_result = subprocess.run(  
 experiment\_cmd,  
 capture\_output=True,  
 text=True,  
 timeout=60  
 )  
   
 logger.debug(f"Experiment process completed with return code: {exp\_result.returncode}")  
 logger.debug(f"stdout: {exp\_result.stdout}")  
 logger.debug(f"stderr: {exp\_result.stderr}")  
   
 if exp\_result.returncode != 0:  
 raise Exception(f"Experiment failed: {exp\_result.stderr}")

\*\*Part 4 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

# Handle the special case of collision experiment which creates two output files  
 if experiment\_type == "collision":  
 # For collision, we get two files: \*\_string.png and \*\_integer.png  
 # For now, just use the string version  
 string\_output = output\_file.replace(".png", "\_string.png")  
 if os.path.exists(string\_output):  
 logger.debug(f"Using string collision output: {string\_output}")  
 output\_file = string\_output  
   
 # Check if output file exists  
 if not os.path.exists(output\_file):  
 # List files in directory for debugging  
 dir\_path = os.path.dirname(output\_file)  
 logger.debug(f"Files in output directory: {os.listdir(dir\_path) if os.path.exists(dir\_path) else 'Directory not found'}")

\*\*Part 5 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

raise FileNotFoundError(f"Output image not found: {output\_file}")  
   
 logger.debug(f"Output file exists: {output\_file}, size: {os.path.getsize(output\_file)} bytes")  
   
 # Read the image file  
 with open(output\_file, 'rb') as f:  
 image\_bytes = f.read()  
 logger.debug(f"Read image file: {len(image\_bytes)} bytes")  
   
 # Convert to base64  
 base64\_image = base64.b64encode(image\_bytes).decode('utf-8')  
 logger.debug(f"Converted image to base64: {len(base64\_image)} characters")  
   
 return base64\_image

\*\*Part 6 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

except Exception as e:  
 logger.error(f"Error in run\_experiment: {str(e)}")  
 logger.error(traceback.format\_exc())  
 raise  
   
 finally:  
 # Clean up  
 if output\_file and os.path.exists(output\_file):  
 try:  
 os.remove(output\_file)  
 logger.debug(f"Deleted file: {output\_file}")  
 except Exception as e:  
 logger.error(f"Failed to delete {output\_file}: {str(e)}")  
   
 # If it's a collision experiment, also try to clean up the integer output file

\*\*Part 7 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

if experiment\_type == "collision" and output\_file:  
 integer\_output = output\_file.replace("\_string.png", "\_integer.png")  
 if os.path.exists(integer\_output):  
 try:  
 os.remove(integer\_output)  
 logger.debug(f"Deleted file: {integer\_output}")  
 except Exception as e:  
 logger.error(f"Failed to delete {integer\_output}: {str(e)}")  
   
 # Remove the temporary directory  
 if temp\_dir and os.path.exists(temp\_dir):  
 try:  
 shutil.rmtree(temp\_dir)  
 logger.debug(f"Deleted temporary directory: {temp\_dir}")  
 except Exception as e:

\*\*Part 8 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

logger.error(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")  
  
# Initialize JavaBridge  
java\_bridge = JavaBridge()  
  
@app.route('/')

\*\*Part 9 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

## index (Split)

def index():  
 # Define template variables  
 template\_data = {  
 'title': 'HashMapper - Text to Image Fingerprints',  
 'version': '1.0.0',  
 'default\_text': 'It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness...',  
 'default\_map\_size': 128,  
 'default\_salt\_level': 5,  
 'default\_smooth\_radius': 2,  
 'hash\_functions': [  
 'String Length',  
 'First Character',  
 'First + Last Character',  
 'Character Sum',  
 'Random'

\*\*Part 1 of `index`:\*\*  
This section handles specific sub-tasks within `index`, such as initializing, processing loops, or output formatting.

],  
 'hash\_function\_descriptions': [  
 {'name': 'String Length', 'description': 'Uses only the length of words'},  
 {'name': 'First Character', 'description': 'Uses only the first character of words'},  
 {'name': 'First + Last Character', 'description': 'Combines first and last characters'},  
 {'name': 'Character Sum', 'description': 'Sums all character values'},  
 {'name': 'Random', 'description': 'Creates pseudo-random but deterministic patterns'}  
 ],  
 'experiments': [  
 {'type': 'collision', 'title': 'Collision Analysis'},  
 {'type': 'lookup', 'title': 'Lookup Performance'},  
 {'type': 'distribution', 'title': 'Bucket Distribution'},  
 {'type': 'hashFunction', 'title': 'Hash Function Comparison'},  
 {'type': 'comparison', 'title': 'HashMap Comparison'},  
 {'type': 'textFingerprint', 'title': 'Text Fingerprint Analysis'}

\*\*Part 2 of `index`:\*\*  
This section handles specific sub-tasks within `index`, such as initializing, processing loops, or output formatting.

]  
 }  
   
 logger.debug("Rendering index template")  
 return render\_template('index.html', \*\*template\_data)  
  
@app.route('/api/test', methods=['GET'])

\*\*Part 3 of `index`:\*\*  
This section handles specific sub-tasks within `index`, such as initializing, processing loops, or output formatting.

## test\_api

def test\_api():  
 """Simple API endpoint for testing JSON responses"""  
 logger.debug("Test API endpoint called")  
 return jsonify({'status': 'ok', 'message': 'API is working'})  
  
@app.route('/api/generate-fingerprint', methods=['POST'])

\*\*Detailed Description:\*\*   
The method `test\_api` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## generate\_fingerprint (Split)

def generate\_fingerprint():  
 """API endpoint to generate text fingerprints"""  
 logger.debug("Generate fingerprint API endpoint called")  
   
 try:  
 # Get form data  
 text = request.form.get('text', '')  
 size = int(request.form.get('size', 128))  
 hash\_function = request.form.get('hashFunction', 'String Length')  
 salt\_level = float(request.form.get('saltLevel', 0.05))  
 smooth\_radius = int(request.form.get('smoothRadius', 2))  
   
 logger.debug(f"Request parameters: text\_length={len(text)}, size={size}, hash\_function={hash\_function}, salt\_level={salt\_level}, smooth\_radius={smooth\_radius}")  
   
 if not text:

\*\*Part 1 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

logger.warning("No text provided in request")  
 return jsonify({'error': 'No text provided'}), 400  
   
 # Generate fingerprint using Java bridge  
 logger.debug("Calling java\_bridge.generate\_fingerprint()")  
 raw\_image, enhanced\_image, stats = java\_bridge.generate\_fingerprint(  
 text, size, hash\_function, salt\_level, smooth\_radius  
 )  
   
 logger.debug(f"Generate fingerprint returned: raw\_image={len(raw\_image) if raw\_image else 'None'} bytes, enhanced\_image={len(enhanced\_image) if enhanced\_image else 'None'} bytes, stats={stats}")  
   
 # Convert images to base64 for display in browser  
 raw\_base64 = base64.b64encode(raw\_image).decode('utf-8')  
 enhanced\_base64 = base64.b64encode(enhanced\_image).decode('utf-8')

\*\*Part 2 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

# Create response JSON  
 response\_data = {  
 'raw\_image': raw\_base64,  
 'enhanced\_image': enhanced\_base64,  
 'stats': stats  
 }  
   
 # Validate JSON response before returning  
 try:  
 # Test serialize to validate  
 json\_response = json.dumps(response\_data)  
 logger.debug(f"Response JSON created: {len(json\_response)} characters")  
 except Exception as e:  
 logger.error(f"JSON serialization error: {str(e)}")  
 # Create a safe response

\*\*Part 3 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

response\_data = {  
 'error': 'Error creating JSON response',  
 'message': str(e)  
 }  
   
 logger.debug("Returning successful response")  
 return jsonify(response\_data)  
   
 except Exception as e:  
 logger.error(f"Error generating fingerprint: {str(e)}")  
 logger.error(traceback.format\_exc())  
 return jsonify({'error': str(e)}), 500  
  
@app.route('/api/run-experiment', methods=['POST'])

\*\*Part 4 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

## run\_experiment (Split)

def run\_experiment():  
 """API endpoint to run HashMap experiments"""  
 logger.debug("Run experiment API endpoint called")  
   
 try:  
 # Get form data  
 experiment\_type = request.form.get('type', 'collision')  
 logger.debug(f"Request parameters: experiment\_type={experiment\_type}")  
   
 # Run experiment using Java bridge  
 logger.debug("Calling java\_bridge.run\_experiment()")  
 base64\_image = java\_bridge.run\_experiment(experiment\_type)  
 logger.debug(f"Run experiment returned: base64\_image={len(base64\_image) if base64\_image else 'None'} characters")  
   
 # Create response JSON

\*\*Part 1 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

response\_data = {  
 'image': base64\_image  
 }  
   
 # Validate JSON response before returning  
 try:  
 # Test serialize to validate  
 json\_response = json.dumps(response\_data)  
 logger.debug(f"Response JSON created: {len(json\_response)} characters")  
 except Exception as e:  
 logger.error(f"JSON serialization error: {str(e)}")  
 # Create a safe response  
 response\_data = {  
 'error': 'Error creating JSON response',  
 'message': str(e)

\*\*Part 2 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

}  
   
 logger.debug("Returning successful response")  
 return jsonify(response\_data)  
   
 except Exception as e:  
 logger.error(f"Error running experiment: {str(e)}")  
 logger.error(traceback.format\_exc())  
 return jsonify({'error': str(e)}), 500  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 logger.info("Starting Flask application")  
 app.run(debug=True)

\*\*Part 3 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

# debug\_endpoint.py

Methods and classes defined in this file are shown below.

## index

def index():  
 return "API is working"  
  
@app.route('/api/test', methods=['GET'])

\*\*Detailed Description:\*\*   
The method `index` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## test\_api

def test\_api():  
 return jsonify({'status': 'ok', 'message': 'API is working'})  
  
@app.route('/api/generate-fingerprint', methods=['POST'])

\*\*Detailed Description:\*\*   
The method `test\_api` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## generate\_fingerprint

def generate\_fingerprint():  
 return jsonify({  
 'raw\_image': 'base64\_data\_would\_go\_here',   
 'enhanced\_image': 'base64\_data\_would\_go\_here',  
 'stats': {  
 'text\_length': 100,  
 'total\_words': 20,  
 'unique\_words': 15,  
 'collisions': 5,  
 'max\_collision\_level': 2  
 }  
 })  
  
@app.route('/api/run-experiment', methods=['POST'])

\*\*Detailed Description:\*\*   
The method `generate\_fingerprint` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## run\_experiment

def run\_experiment():  
 return jsonify({  
 'image': 'base64\_data\_would\_go\_here'  
 })  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

\*\*Detailed Description:\*\*   
The method `run\_experiment` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

# java\_bridge.py

Methods and classes defined in this file are shown below.

## generate\_fingerprint (Split)

def generate\_fingerprint(text, size, hash\_function, salt\_level, smooth\_radius):  
 """  
 Generate fingerprint using Java code  
 Returns: (raw\_image\_bytes, enhanced\_image\_bytes, stats\_dict)  
 """  
 text\_path = None  
 raw\_output = None  
 enhanced\_output = None  
 stats\_output = None  
  
 try:  
 # Create a temporary directory to store all files  
 temp\_dir = tempfile.mkdtemp()  
   
 # Write text to a temporary file

\*\*Part 1 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

text\_path = os.path.join(temp\_dir, 'input.txt')  
 with open(text\_path, 'w', encoding='utf-8') as text\_file:  
 text\_file.write(text)  
   
 # Define output paths  
 raw\_output = os.path.join(temp\_dir, "raw\_output.png")  
 enhanced\_output = os.path.join(temp\_dir, "enhanced\_output.png")  
 stats\_output = os.path.join(temp\_dir, "stats\_output.json")  
   
 print(f"Working directory: {os.getcwd()}")  
 print(f"Text file: {text\_path}")  
 print(f"Output files: {raw\_output}, {enhanced\_output}, {stats\_output}")  
   
 # Get the path to the java directory  
 java\_dir = os.path.join(os.getcwd(), 'java')

\*\*Part 2 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

# Build the command  
 cmd = [  
 "java",  
 "-Djava.awt.headless=true", # Enable headless mode for server environments  
 "-cp", f"{java\_dir}:lib/\*", # Include java directory and all JARs in lib  
 "HashMapVisualizer", # The main class  
 "generateTextFingerprint", # The method to call  
 text\_path,  
 str(size),  
 hash\_function,  
 str(salt\_level),  
 str(smooth\_radius),  
 raw\_output,  
 enhanced\_output,

\*\*Part 3 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

stats\_output  
 ]  
   
 # For Windows, use semicolons instead of colons in classpath  
 if os.name == 'nt':  
 cmd[3] = f"{java\_dir};lib/\*"  
   
 print(f"Executing command: {' '.join(cmd)}")  
   
 # Run the Java process  
 result = subprocess.run(  
 cmd,  
 capture\_output=True,  
 text=True,  
 timeout=30

\*\*Part 4 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

)  
   
 print(f"Java process completed with return code: {result.returncode}")  
 print(f"stdout: {result.stdout}")  
 print(f"stderr: {result.stderr}")  
   
 if result.returncode != 0:  
 raise Exception(f"Java process failed: {result.stderr}")  
   
 # Check if output files exist  
 for file\_path in [raw\_output, enhanced\_output, stats\_output]:  
 if not os.path.exists(file\_path):  
 raise FileNotFoundError(f"Output file not found: {file\_path}")  
 print(f"File exists: {file\_path}, size: {os.path.getsize(file\_path)} bytes")

\*\*Part 5 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

# Read the output files  
 with open(raw\_output, 'rb') as f:  
 raw\_bytes = f.read()  
   
 with open(enhanced\_output, 'rb') as f:  
 enhanced\_bytes = f.read()  
   
 with open(stats\_output, 'r') as f:  
 stats = json.load(f)  
   
 return raw\_bytes, enhanced\_bytes, stats  
   
 except Exception as e:  
 print(f"Error in generate\_fingerprint: {str(e)}")  
 raise

\*\*Part 6 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

finally:  
 # Clean up temporary files  
 for file\_path in [text\_path, raw\_output, enhanced\_output, stats\_output]:  
 if file\_path and os.path.exists(file\_path):  
 try:  
 os.remove(file\_path)  
 print(f"Deleted file: {file\_path}")  
 except Exception as e:  
 print(f"Failed to delete {file\_path}: {str(e)}")  
   
 # Remove the temporary directory  
 if temp\_dir and os.path.exists(temp\_dir):  
 try:  
 import shutil

\*\*Part 7 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

shutil.rmtree(temp\_dir)  
 print(f"Deleted temporary directory: {temp\_dir}")  
 except Exception as e:  
 print(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")

\*\*Part 8 of `generate\_fingerprint`:\*\*  
This section handles specific sub-tasks within `generate\_fingerprint`, such as initializing, processing loops, or output formatting.

## run\_experiment (Split)

def run\_experiment(experiment\_type):  
 """  
 Run HashMap experiment and return the visualization  
 Returns: image bytes  
 """  
 output\_file = None  
   
 try:  
 # Create a temporary directory  
 temp\_dir = tempfile.mkdtemp()  
 output\_file = os.path.join(temp\_dir, f"{experiment\_type}\_output.png")  
   
 # Get the path to the java directory  
 java\_dir = os.path.join(os.getcwd(), 'java')

\*\*Part 1 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

# First run the experiment to generate data  
 experiment\_cmd = [  
 "java",  
 "-Djava.awt.headless=true",  
 "-cp", f"{java\_dir}:lib/\*",  
 "HashMapExperiment",  
 f"run{experiment\_type.capitalize()}Experiment"  
 ]  
   
 # For Windows, use semicolons instead of colons in classpath  
 if os.name == 'nt':  
 experiment\_cmd[3] = f"{java\_dir};lib/\*"  
   
 print(f"Running experiment: {' '.join(experiment\_cmd)}")  
 exp\_result = subprocess.run(

\*\*Part 2 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

experiment\_cmd,  
 capture\_output=True,  
 text=True,  
 timeout=30  
 )  
   
 if exp\_result.returncode != 0:  
 raise Exception(f"Experiment failed: {exp\_result.stderr}")  
   
 print(f"Experiment completed: {exp\_result.stdout}")  
   
 # Then run the visualization  
 csv\_file = f"{experiment\_type.lower()}\_data.csv"  
 if not os.path.exists(csv\_file):  
 # Use the appropriate CSV file based on experiment type

\*\*Part 3 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

if experiment\_type == "hashFunction":  
 csv\_file = "hash\_function\_comparison.csv"  
 elif experiment\_type == "collision":  
 csv\_file = "string\_collisions.csv"  
 elif experiment\_type == "lookup":  
 csv\_file = "lookup\_performance.csv"  
 elif experiment\_type == "distribution":  
 csv\_file = "bucket\_distribution.csv"  
 elif experiment\_type == "comparison":  
 csv\_file = "hashmap\_comparison.csv"  
 elif experiment\_type == "textFingerprint":  
 csv\_file = "text\_fingerprint\_analysis.csv"  
   
 # Check if CSV file exists  
 if not os.path.exists(csv\_file):

\*\*Part 4 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

raise FileNotFoundError(f"CSV file not found: {csv\_file}")  
   
 print(f"Using CSV file: {csv\_file}")  
   
 # Run visualization  
 viz\_cmd = [  
 "java",  
 "-Djava.awt.headless=true",  
 "-cp", f"{java\_dir}:lib/\*",  
 "HashMapVisualizer",  
 f"visualize{experiment\_type.capitalize()}",  
 csv\_file,  
 "Experiment Results",  
 output\_file  
 ]

\*\*Part 5 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

# For Windows, use semicolons instead of colons in classpath  
 if os.name == 'nt':  
 viz\_cmd[3] = f"{java\_dir};lib/\*"  
   
 print(f"Running visualization: {' '.join(viz\_cmd)}")  
 viz\_result = subprocess.run(  
 viz\_cmd,  
 capture\_output=True,  
 text=True,  
 timeout=30  
 )  
   
 if viz\_result.returncode != 0:  
 raise Exception(f"Visualization failed: {viz\_result.stderr}")

\*\*Part 6 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

print(f"Visualization completed: {viz\_result.stdout}")  
   
 # Check if output file exists  
 if not os.path.exists(output\_file):  
 raise FileNotFoundError(f"Output image not found: {output\_file}")  
   
 print(f"Output file exists: {output\_file}, size: {os.path.getsize(output\_file)} bytes")  
   
 # Read the image file  
 with open(output\_file, 'rb') as f:  
 image\_bytes = f.read()  
   
 # Convert to base64  
 base64\_image = base64.b64encode(image\_bytes).decode('utf-8')

\*\*Part 7 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

return base64\_image  
   
 except Exception as e:  
 print(f"Error in run\_experiment: {str(e)}")  
 raise  
   
 finally:  
 # Clean up  
 if output\_file and os.path.exists(output\_file):  
 try:  
 os.remove(output\_file)  
 print(f"Deleted file: {output\_file}")  
 except Exception as e:  
 print(f"Failed to delete {output\_file}: {str(e)}")

\*\*Part 8 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

# Remove the temporary directory  
 if temp\_dir and os.path.exists(temp\_dir):  
 try:  
 import shutil  
 shutil.rmtree(temp\_dir)  
 print(f"Deleted temporary directory: {temp\_dir}")  
 except Exception as e:  
 print(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")

\*\*Part 9 of `run\_experiment`:\*\*  
This section handles specific sub-tasks within `run\_experiment`, such as initializing, processing loops, or output formatting.

# HashMapExperiment.java

Methods and classes defined in this file are shown below.

## {

public class HashMapExperiment {  
  
 /\*\*  
 \* Generate a dataset of random strings  
 \*/

\*\*Detailed Description:\*\*   
The method `{` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## generateStringDataset (Split)

private static List<String> generateStringDataset(int count, int minLength, int maxLength) {  
 List<String> dataset = new ArrayList<>();  
 Random random = new Random();  
 String chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";  
  
 for (int i = 0; i < count; i++) {  
 int length = random.nextInt(maxLength - minLength + 1) + minLength;  
 StringBuilder sb = new StringBuilder(length);  
  
 for (int j = 0; j < length; j++) {  
 int index = random.nextInt(chars.length());  
 sb.append(chars.charAt(index));  
 }  
  
 dataset.add(sb.toString());

\*\*Part 1 of `generateStringDataset`:\*\*  
This section handles specific sub-tasks within `generateStringDataset`, such as initializing, processing loops, or output formatting.

}  
  
 return dataset;  
 }  
  
 /\*\*  
 \* Generate a dataset of random integers  
 \*/

\*\*Part 2 of `generateStringDataset`:\*\*  
This section handles specific sub-tasks within `generateStringDataset`, such as initializing, processing loops, or output formatting.

## generateIntegerDataset

private static List<Integer> generateIntegerDataset(int count, int max) {  
 List<Integer> dataset = new ArrayList<>();  
 Random random = new Random();  
  
 for (int i = 0; i < count; i++) {  
 dataset.add(random.nextInt(max));  
 }  
  
 return dataset;  
 }  
  
 /\*\*  
 \* Run experiments with different hash functions  
 \*/

\*\*Detailed Description:\*\*   
The method `generateIntegerDataset` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## runHashFunctionExperiment (Split)

public static void runHashFunctionExperiment() throws IOException {  
 String[] hashFunctions = {  
 "String Length", "First Character", "First + Last Character",  
 "Character Sum", "Random"  
 };  
  
 int dataSize = 10000;  
 int mapSize = 128;  
  
 FileWriter writer = new FileWriter("hash\_function\_comparison.csv");  
 writer.write("HashFunction,Collisions,MaxBucketSize,EmptyBuckets\n");  
  
 // Generate dataset  
 List<String> dataset = generateStringDataset(dataSize, 5, 15);

\*\*Part 1 of `runHashFunctionExperiment`:\*\*  
This section handles specific sub-tasks within `runHashFunctionExperiment`, such as initializing, processing loops, or output formatting.

for (String hashFunction : hashFunctions) {  
 // Set hash function  
 SimpleHashMap.setHashFunctionType(hashFunction);  
  
 // Create HashMap  
 SimpleHashMap<String, Boolean> map = new SimpleHashMap<>(mapSize);  
  
 // Insert all data  
 for (String item : dataset) {  
 map.put(item, true);  
 }  
  
 // Get metrics  
 int collisions = map.getCollisionCount();  
 int[] distribution = map.getBucketDistribution();

\*\*Part 2 of `runHashFunctionExperiment`:\*\*  
This section handles specific sub-tasks within `runHashFunctionExperiment`, such as initializing, processing loops, or output formatting.

// Find max bucket size  
 int maxBucketSize = 0;  
 int emptyBuckets = 0;  
 for (int size : distribution) {  
 maxBucketSize = Math.max(maxBucketSize, size);  
 if (size == 0) emptyBuckets++;  
 }  
  
 // Write results  
 writer.write(String.format("%s,%d,%d,%d\n",  
 hashFunction, collisions, maxBucketSize, emptyBuckets));  
 }  
  
 writer.close();

\*\*Part 3 of `runHashFunctionExperiment`:\*\*  
This section handles specific sub-tasks within `runHashFunctionExperiment`, such as initializing, processing loops, or output formatting.

System.out.println("Hash function experiment completed.");  
 }  
  
 /\*\*  
 \* Run experiment to measure collisions with different hash map sizes  
 \*/

\*\*Part 4 of `runHashFunctionExperiment`:\*\*  
This section handles specific sub-tasks within `runHashFunctionExperiment`, such as initializing, processing loops, or output formatting.

## runCollisionExperiment (Split)

public static void runCollisionExperiment() throws IOException {  
 int[] dataSizes = {1000, 5000, 10000, 20000};  
 int[] mapSizes = {16, 32, 64, 128, 256, 512, 1024};  
  
 // Create CSV file for string data  
 FileWriter stringWriter = new FileWriter("string\_collisions.csv");  
 stringWriter.write("DataSize,MapSize,Collisions,LoadFactor\n");  
  
 // Run experiment with string data  
 for (int dataSize : dataSizes) {  
 List<String> dataset = generateStringDataset(dataSize, 5, 15);  
  
 for (int mapSize : mapSizes) {  
 SimpleHashMap<String, Boolean> map = new SimpleHashMap<>(mapSize);

\*\*Part 1 of `runCollisionExperiment`:\*\*  
This section handles specific sub-tasks within `runCollisionExperiment`, such as initializing, processing loops, or output formatting.

// Insert all data  
 for (String item : dataset) {  
 map.put(item, true);  
 }  
  
 // Record results  
 stringWriter.write(String.format("%d,%d,%d,%.4f\n",  
 dataSize, mapSize, map.getCollisionCount(), map.getLoadFactor()));  
 }  
 }  
 stringWriter.close();  
  
 // Create CSV file for integer data  
 FileWriter intWriter = new FileWriter("integer\_collisions.csv");  
 intWriter.write("DataSize,MapSize,Collisions,LoadFactor\n");

\*\*Part 2 of `runCollisionExperiment`:\*\*  
This section handles specific sub-tasks within `runCollisionExperiment`, such as initializing, processing loops, or output formatting.

// Run experiment with integer data  
 for (int dataSize : dataSizes) {  
 List<Integer> dataset = generateIntegerDataset(dataSize, 100000);  
  
 for (int mapSize : mapSizes) {  
 SimpleHashMap<Integer, Boolean> map = new SimpleHashMap<>(mapSize);  
  
 // Insert all data  
 for (Integer item : dataset) {  
 map.put(item, true);  
 }  
  
 // Record results  
 intWriter.write(String.format("%d,%d,%d,%.4f\n",

\*\*Part 3 of `runCollisionExperiment`:\*\*  
This section handles specific sub-tasks within `runCollisionExperiment`, such as initializing, processing loops, or output formatting.

dataSize, mapSize, map.getCollisionCount(), map.getLoadFactor()));  
 }  
 }  
 intWriter.close();  
  
 System.out.println("Collision experiment completed.");  
 }  
  
 /\*\*  
 \* Run experiment to measure lookup performance  
 \*/

\*\*Part 4 of `runCollisionExperiment`:\*\*  
This section handles specific sub-tasks within `runCollisionExperiment`, such as initializing, processing loops, or output formatting.

## runLookupExperiment (Split)

public static void runLookupExperiment() throws IOException {  
 int[] dataSizes = {10000, 50000, 100000};  
 int[] mapSizes = {16, 64, 256, 1024, 4096};  
 int lookupCount = 10000;  
  
 FileWriter writer = new FileWriter("lookup\_performance.csv");  
 writer.write("DataSize,MapSize,LoadFactor,LookupTimeMs\n");  
  
 for (int dataSize : dataSizes) {  
 List<String> dataset = generateStringDataset(dataSize, 5, 15);  
 List<String> lookupKeys = dataset.subList(0, Math.min(lookupCount, dataSize));  
  
 for (int mapSize : mapSizes) {  
 SimpleHashMap<String, Boolean> map = new SimpleHashMap<>(mapSize);

\*\*Part 1 of `runLookupExperiment`:\*\*  
This section handles specific sub-tasks within `runLookupExperiment`, such as initializing, processing loops, or output formatting.

// Insert all data  
 for (String item : dataset) {  
 map.put(item, true);  
 }  
  
 // Measure lookup time  
 long startTime = System.nanoTime();  
 for (String key : lookupKeys) {  
 map.get(key);  
 }  
 long endTime = System.nanoTime();  
  
 double elapsedMs = (endTime - startTime) / 1\_000\_000.0;  
  
 // Record results

\*\*Part 2 of `runLookupExperiment`:\*\*  
This section handles specific sub-tasks within `runLookupExperiment`, such as initializing, processing loops, or output formatting.

writer.write(String.format("%d,%d,%.4f,%.4f\n",  
 dataSize, mapSize, map.getLoadFactor(), elapsedMs));  
 }  
 }  
 writer.close();  
  
 System.out.println("Lookup experiment completed.");  
 }  
  
 /\*\*  
 \* Run experiment to analyze bucket distribution  
 \*/

\*\*Part 3 of `runLookupExperiment`:\*\*  
This section handles specific sub-tasks within `runLookupExperiment`, such as initializing, processing loops, or output formatting.

## runDistributionExperiment (Split)

public static void runDistributionExperiment() throws IOException {  
 int dataSize = 10000;  
 int mapSize = 128;  
  
 List<String> dataset = generateStringDataset(dataSize, 5, 15);  
 SimpleHashMap<String, Boolean> map = new SimpleHashMap<>(mapSize);  
  
 // Insert all data  
 for (String item : dataset) {  
 map.put(item, true);  
 }  
  
 // Get bucket distribution  
 int[] distribution = map.getBucketDistribution();

\*\*Part 1 of `runDistributionExperiment`:\*\*  
This section handles specific sub-tasks within `runDistributionExperiment`, such as initializing, processing loops, or output formatting.

// Write distribution to CSV  
 FileWriter writer = new FileWriter("bucket\_distribution.csv");  
 writer.write("BucketIndex,ItemCount\n");  
  
 for (int i = 0; i < distribution.length; i++) {  
 writer.write(String.format("%d,%d\n", i, distribution[i]));  
 }  
 writer.close();  
  
 System.out.println("Distribution experiment completed.");  
 }  
  
 /\*\*  
 \* Compare with Java's HashMap  
 \*/

\*\*Part 2 of `runDistributionExperiment`:\*\*  
This section handles specific sub-tasks within `runDistributionExperiment`, such as initializing, processing loops, or output formatting.

## compareWithJavaHashMap (Split)

public static void compareWithJavaHashMap() throws IOException {  
 int[] dataSizes = {10000, 50000, 100000};  
 int lookupCount = 10000;  
  
 FileWriter writer = new FileWriter("hashmap\_comparison.csv");  
 writer.write("DataSize,SimpleHashMapTimeMs,JavaHashMapTimeMs\n");  
  
 for (int dataSize : dataSizes) {  
 List<String> dataset = generateStringDataset(dataSize, 5, 15);  
 List<String> lookupKeys = dataset.subList(0, Math.min(lookupCount, dataSize));  
  
 // Test with SimpleHashMap  
 SimpleHashMap<String, Boolean> simpleMap = new SimpleHashMap<>(1024);  
 for (String item : dataset) {  
 simpleMap.put(item, true);

\*\*Part 1 of `compareWithJavaHashMap`:\*\*  
This section handles specific sub-tasks within `compareWithJavaHashMap`, such as initializing, processing loops, or output formatting.

}  
  
 long simpleStartTime = System.nanoTime();  
 for (String key : lookupKeys) {  
 simpleMap.get(key);  
 }  
 long simpleEndTime = System.nanoTime();  
 double simpleElapsedMs = (simpleEndTime - simpleStartTime) / 1\_000\_000.0;  
  
 // Test with Java HashMap  
 java.util.HashMap<String, Boolean> javaMap = new java.util.HashMap<>(1024);  
 for (String item : dataset) {  
 javaMap.put(item, true);  
 }

\*\*Part 2 of `compareWithJavaHashMap`:\*\*  
This section handles specific sub-tasks within `compareWithJavaHashMap`, such as initializing, processing loops, or output formatting.

long javaStartTime = System.nanoTime();  
 for (String key : lookupKeys) {  
 javaMap.get(key);  
 }  
 long javaEndTime = System.nanoTime();  
 double javaElapsedMs = (javaEndTime - javaStartTime) / 1\_000\_000.0;  
  
 // Record results  
 writer.write(String.format("%d,%.4f,%.4f\n", dataSize, simpleElapsedMs, javaElapsedMs));  
 }  
 writer.close();  
  
 System.out.println("HashMap comparison completed.");  
 }

\*\*Part 3 of `compareWithJavaHashMap`:\*\*  
This section handles specific sub-tasks within `compareWithJavaHashMap`, such as initializing, processing loops, or output formatting.

/\*\*  
 \* Run experiment to analyze text fingerprint collision patterns  
 \*/

\*\*Part 4 of `compareWithJavaHashMap`:\*\*  
This section handles specific sub-tasks within `compareWithJavaHashMap`, such as initializing, processing loops, or output formatting.

## runTextFingerprintExperiment (Split)

public static void runTextFingerprintExperiment() throws IOException {  
 // Sample texts with different characteristics  
 String[] texts = {  
 // Literature sample  
 "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, " +  
 "it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness...",  
  
 // Technical sample  
 "A hash function is any function that can be used to map data of arbitrary size to fixed-size values. " +  
 "The values returned by a hash function are called hash values, hash codes, digests, or simply hashes.",  
  
 // Poetry sample  
 "Two roads diverged in a yellow wood, And sorry I could not travel both And be one traveler, long I stood " +  
 "And looked down one as far as I could To where it bent in the undergrowth",

\*\*Part 1 of `runTextFingerprintExperiment`:\*\*  
This section handles specific sub-tasks within `runTextFingerprintExperiment`, such as initializing, processing loops, or output formatting.

// Code sample  
 "public static void main(String[] args) { System.out.println(\"Hello, World!\"); " +  
 "for(int i=0; i<10; i++) { if(i % 2 == 0) { System.out.println(i); } } }"  
 };  
  
 String[] textTypes = {"Literature", "Technical", "Poetry", "Code"};  
 int mapSize = 64;  
  
 FileWriter writer = new FileWriter("text\_fingerprint\_analysis.csv");  
 writer.write("TextType,Collisions,MaxCollisionLevel,UniqueWords,TotalWords\n");  
  
 for (int i = 0; i < texts.length; i++) {  
 // Create HashMap for this text  
 SimpleHashMap<String, Integer> map = new SimpleHashMap<>(mapSize);

\*\*Part 2 of `runTextFingerprintExperiment`:\*\*  
This section handles specific sub-tasks within `runTextFingerprintExperiment`, such as initializing, processing loops, or output formatting.

// Process words  
 String[] words = texts[i].split("\\s+");  
 int uniqueWords = 0;  
 java.util.HashSet<String> uniqueWordSet = new java.util.HashSet<>();  
  
 for (String word : words) {  
 word = word.toLowerCase().replaceAll("[^a-z]", "");  
 if (!word.isEmpty()) {  
 if (!uniqueWordSet.contains(word)) {  
 uniqueWordSet.add(word);  
 uniqueWords++;  
 }  
 map.put(word, 1);  
 }  
 }

\*\*Part 3 of `runTextFingerprintExperiment`:\*\*  
This section handles specific sub-tasks within `runTextFingerprintExperiment`, such as initializing, processing loops, or output formatting.

// Get metrics  
 int collisions = map.getCollisionCount();  
  
 // Find max collision level (by analyzing bucket sizes)  
 int[] distribution = map.getBucketDistribution();  
 int maxBucketSize = 0;  
 for (int size : distribution) {  
 maxBucketSize = Math.max(maxBucketSize, size);  
 }  
  
 // Write results  
 writer.write(String.format("%s,%d,%d,%d,%d\n",  
 textTypes[i], collisions, maxBucketSize, uniqueWords, words.length));  
 }

\*\*Part 4 of `runTextFingerprintExperiment`:\*\*  
This section handles specific sub-tasks within `runTextFingerprintExperiment`, such as initializing, processing loops, or output formatting.

writer.close();  
 System.out.println("Text fingerprint experiment completed.");  
 }

\*\*Part 5 of `runTextFingerprintExperiment`:\*\*  
This section handles specific sub-tasks within `runTextFingerprintExperiment`, such as initializing, processing loops, or output formatting.

## main

public static void main(String[] args) {  
 try {  
 System.out.println("Starting HashMap experiments...");  
  
 runHashFunctionExperiment();  
 runCollisionExperiment();  
 runLookupExperiment();  
 runDistributionExperiment();  
 compareWithJavaHashMap();  
 runTextFingerprintExperiment();  
  
 System.out.println("All experiments completed successfully.");  
 } catch (IOException e) {  
 System.err.println("Error writing experiment results: " + e.getMessage());  
 }  
 }  
}

\*\*Detailed Description:\*\*   
The method `main` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

# HashMapExperimentRunner.java

Methods and classes defined in this file are shown below.

## {

public class HashMapExperimentRunner {

\*\*Detailed Description:\*\*   
The method `{` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## main (Split)

public static void main(String[] args) {  
 System.out.println("Starting HashMap Experiment...");  
  
 try {  
 // Parse command-line arguments  
 String textFile = null;  
 int size = 128;  
 String hashFunction = "String Length";  
 double saltLevel = 0.05;  
 int smoothRadius = 2;  
 String rawOutput = null;  
 String enhancedOutput = null;  
 String statsOutput = null;  
 String experimentType = null;  
 String output = null;

\*\*Part 1 of `main`:\*\*  
This section handles specific sub-tasks within `main`, such as initializing, processing loops, or output formatting.

for (int i = 0; i < args.length; i++) {  
 switch (args[i]) {  
 case "--text-file":  
 textFile = args[++i];  
 break;  
 case "--size":  
 size = Integer.parseInt(args[++i]);  
 break;  
 case "--hash-function":  
 hashFunction = args[++i];  
 break;  
 case "--salt-level":  
 saltLevel = Double.parseDouble(args[++i]);  
 break;

\*\*Part 2 of `main`:\*\*  
This section handles specific sub-tasks within `main`, such as initializing, processing loops, or output formatting.

case "--smooth-radius":  
 smoothRadius = Integer.parseInt(args[++i]);  
 break;  
 case "--raw-output":  
 rawOutput = args[++i];  
 break;  
 case "--enhanced-output":  
 enhancedOutput = args[++i];  
 break;  
 case "--stats-output":  
 statsOutput = args[++i];  
 break;  
 case "--type":  
 experimentType = args[++i];  
 break;

\*\*Part 3 of `main`:\*\*  
This section handles specific sub-tasks within `main`, such as initializing, processing loops, or output formatting.

case "--output":  
 output = args[++i];  
 break;  
 }  
 }  
  
 if (textFile != null && rawOutput != null && enhancedOutput != null && statsOutput != null) {  
 // Generate text fingerprint  
 System.out.println("Generating text fingerprint...");  
 HashMapVisualizer.generateTextFingerprint(  
 textFile, size, hashFunction, saltLevel, smoothRadius,  
 rawOutput, enhancedOutput, statsOutput  
 );  
 System.out.println("Text fingerprint generation completed.");  
 } else if (experimentType != null && output != null) {

\*\*Part 4 of `main`:\*\*  
This section handles specific sub-tasks within `main`, such as initializing, processing loops, or output formatting.

// Run experiments  
 System.out.println("Running experiments...");  
 HashMapExperiment.main(args);  
  
 // Generate visualizations  
 System.out.println("Generating visualizations...");  
 switch (experimentType) {  
 case "hash\_function":  
 HashMapVisualizer.visualizeHashFunctionComparison("hash\_function\_comparison.csv", output);  
 break;  
 case "collision":  
 HashMapVisualizer.visualizeCollisions("string\_collisions.csv", "String Key Collisions", output.replace(".png", "\_string.png"));  
 HashMapVisualizer.visualizeCollisions("integer\_collisions.csv", "Integer Key Collisions", output.replace(".png", "\_integer.png"));  
 break;  
 case "lookup":

\*\*Part 5 of `main`:\*\*  
This section handles specific sub-tasks within `main`, such as initializing, processing loops, or output formatting.

HashMapVisualizer.visualizeLookupPerformance("lookup\_performance.csv", output);  
 break;  
 case "distribution":  
 HashMapVisualizer.visualizeBucketDistribution("bucket\_distribution.csv", output);  
 break;  
 case "comparison":  
 HashMapVisualizer.visualizeHashMapComparison("hashmap\_comparison.csv", output);  
 break;  
 case "text\_fingerprint":  
 HashMapVisualizer.visualizeTextFingerprintAnalysis("text\_fingerprint\_analysis.csv", output);  
 break;  
 default:  
 System.err.println("Unknown experiment type: " + experimentType);  
 System.exit(1);  
 }

\*\*Part 6 of `main`:\*\*  
This section handles specific sub-tasks within `main`, such as initializing, processing loops, or output formatting.

System.out.println("Visualizations completed.");  
 } else {  
 System.err.println("Invalid arguments");  
 System.exit(1);  
 }  
  
 System.out.println("Experiment completed successfully!");  
 } catch (Exception e) {  
 System.err.println("Error during experiment: " + e.getMessage());  
 e.printStackTrace();  
 System.exit(1);  
 }  
 }  
}

\*\*Part 7 of `main`:\*\*  
This section handles specific sub-tasks within `main`, such as initializing, processing loops, or output formatting.

# HashMapVisualizer.java

Methods and classes defined in this file are shown below.

## {

public class HashMapVisualizer {  
  
 /\*\*  
 \* Create a visualization of collision data and save to a file  
 \*/

\*\*Detailed Description:\*\*   
The method `{` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## visualizeCollisions (Split)

public static void visualizeCollisions(String csvFile, String title, String outputFile) {  
 // Read data from CSV file  
 java.util.Map<Integer, java.util.Map<Integer, Integer>> collisionData = new java.util.HashMap<>();  
 java.util.List<Integer> mapSizes = new java.util.ArrayList<>();  
 java.util.List<Integer> dataSizes = new java.util.ArrayList<>();  
  
 try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {  
 String line = br.readLine(); // Skip header  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(",");  
 int dataSize = Integer.parseInt(values[0]);  
 int mapSize = Integer.parseInt(values[1]);  
 int collisions = Integer.parseInt(values[2]);  
  
 if (!dataSizes.contains(dataSize)) {

\*\*Part 1 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

dataSizes.add(dataSize);  
 }  
 if (!mapSizes.contains(mapSize)) {  
 mapSizes.add(mapSize);  
 }  
  
 collisionData.computeIfAbsent(dataSize, k -> new java.util.HashMap<>()).put(mapSize, collisions);  
 }  
 } catch (IOException e) {  
 System.err.println("Error reading CSV file: " + e.getMessage());  
 System.exit(1);  
 }  
  
 // Create BufferedImage  
 int width = 800;

\*\*Part 2 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

int height = 600;  
 BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);  
 Graphics2D g2d = image.createGraphics();  
 g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);  
 g2d.setColor(Color.WHITE);  
 g2d.fillRect(0, 0, width, height);  
  
 int padding = 50;  
  
 // Draw axes  
 g2d.setColor(Color.BLACK);  
 g2d.drawLine(padding, height - padding, width - padding, height - padding); // x-axis  
 g2d.drawLine(padding, height - padding, padding, padding); // y-axis  
  
 // Draw x-axis labels

\*\*Part 3 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

FontMetrics fm = g2d.getFontMetrics();  
 for (int i = 0; i < mapSizes.size(); i++) {  
 int mapSize = mapSizes.get(i);  
 String label = String.valueOf(mapSize);  
 int labelWidth = fm.stringWidth(label);  
 float x = padding + i \* (width - 2 \* padding) / (mapSizes.size() - 1);  
 g2d.drawString(label, x - labelWidth / 2, height - padding / 2);  
 }  
  
 // Find max collision value for scaling  
 int maxCollisions = 0;  
 for (java.util.Map<Integer, Integer> dataMap : collisionData.values()) {  
 for (int collisions : dataMap.values()) {  
 maxCollisions = Math.max(maxCollisions, collisions);  
 }

\*\*Part 4 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

}  
  
 // Draw lines for each data size  
 Color[] colors = {Color.RED, Color.BLUE, Color.GREEN, Color.ORANGE, Color.MAGENTA};  
 for (int i = 0; i < dataSizes.size(); i++) {  
 int dataSize = dataSizes.get(i);  
 g2d.setColor(colors[i % colors.length]);  
  
 int prevX = 0;  
 int prevY = 0;  
 boolean first = true;  
  
 for (int j = 0; j < mapSizes.size(); j++) {  
 int mapSize = mapSizes.get(j);  
 int collisions = collisionData.get(dataSize).get(mapSize);

\*\*Part 5 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

float x = padding + j \* (width - 2 \* padding) / (mapSizes.size() - 1);  
 float y = height - padding - (collisions \* (height - 2 \* padding) / maxCollisions);  
  
 g2d.fillOval((int) x - 3, (int) y - 3, 6, 6);  
 if (!first) {  
 g2d.drawLine(prevX, prevY, (int) x, (int) y);  
 }  
  
 prevX = (int) x;  
 prevY = (int) y;  
 first = false;  
 }  
 }

\*\*Part 6 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

// Draw legend  
 int legendX = width - 200;  
 int legendY = 50;  
 for (int i = 0; i < dataSizes.size(); i++) {  
 g2d.setColor(colors[i % colors.length]);  
 g2d.fillRect(legendX, legendY + i \* 20, 10, 10);  
 g2d.setColor(Color.BLACK);  
 g2d.drawString("Data Size: " + dataSizes.get(i), legendX + 20, legendY + i \* 20 + 10);  
 }  
  
 // Draw titles  
 g2d.setFont(new Font("Arial", Font.BOLD, 16));  
 String xAxisTitle = "HashMap Size";  
 String yAxisTitle = "Number of Collisions";

\*\*Part 7 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

g2d.drawString(xAxisTitle, width / 2 - fm.stringWidth(xAxisTitle) / 2, height - 10);  
 g2d.translate(15, height / 2 + fm.stringWidth(yAxisTitle) / 2);  
 g2d.rotate(-Math.PI / 2);  
 g2d.drawString(yAxisTitle, 0, 0);  
 g2d.rotate(Math.PI / 2);  
 g2d.translate(-15, -(height / 2 + fm.stringWidth(yAxisTitle) / 2));  
  
 g2d.dispose();  
  
 // Save image  
 try {  
 ImageIO.write(image, "png", new File(outputFile));  
 } catch (IOException e) {  
 System.err.println("Error saving image: " + e.getMessage());  
 System.exit(1);

\*\*Part 8 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

}  
 }  
  
 /\*\*  
 \* Create a visualization of lookup performance and save to a file  
 \*/

\*\*Part 9 of `visualizeCollisions`:\*\*  
This section handles specific sub-tasks within `visualizeCollisions`, such as initializing, processing loops, or output formatting.

## visualizeLookupPerformance (Split)

public static void visualizeLookupPerformance(String csvFile, String outputFile) {  
 // Read data from CSV file  
 java.util.Map<Integer, java.util.Map<Integer, Double>> lookupData = new java.util.HashMap<>();  
 java.util.List<Integer> mapSizes = new java.util.ArrayList<>();  
 java.util.List<Integer> dataSizes = new java.util.ArrayList<>();  
  
 try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {  
 String line = br.readLine();  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(",");  
 int dataSize = Integer.parseInt(values[0]);  
 int mapSize = Integer.parseInt(values[1]);  
 double lookupTime = Double.parseDouble(values[3]);  
  
 if (!dataSizes.contains(dataSize)) {

\*\*Part 1 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

dataSizes.add(dataSize);  
 }  
 if (!mapSizes.contains(mapSize)) {  
 mapSizes.add(mapSize);  
 }  
  
 lookupData.computeIfAbsent(dataSize, k -> new java.util.HashMap<>()).put(mapSize, lookupTime);  
 }  
 } catch (IOException e) {  
 System.err.println("Error reading CSV file: " + e.getMessage());  
 System.exit(1);  
 }  
  
 // Create BufferedImage  
 int width = 800;

\*\*Part 2 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

int height = 600;  
 BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);  
 Graphics2D g2d = image.createGraphics();  
 g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);  
 g2d.setColor(Color.WHITE);  
 g2d.fillRect(0, 0, width, height);  
  
 int padding = 50;  
  
 // Draw axes  
 g2d.setColor(Color.BLACK);  
 g2d.drawLine(padding, height - padding, width - padding, height - padding);  
 g2d.drawLine(padding, height - padding, padding, padding);  
  
 // Draw x-axis labels

\*\*Part 3 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

FontMetrics fm = g2d.getFontMetrics();  
 for (int i = 0; i < mapSizes.size(); i++) {  
 int mapSize = mapSizes.get(i);  
 String label = String.valueOf(mapSize);  
 int labelWidth = fm.stringWidth(label);  
 float x = padding + i \* (width - 2 \* padding) / (mapSizes.size() - 1);  
 g2d.drawString(label, x - labelWidth / 2, height - padding / 2);  
 }  
  
 // Find max lookup time for scaling  
 double maxLookupTime = 0;  
 for (java.util.Map<Integer, Double> dataMap : lookupData.values()) {  
 for (double lookupTime : dataMap.values()) {  
 maxLookupTime = Math.max(maxLookupTime, lookupTime);  
 }

\*\*Part 4 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

}  
  
 // Draw lines for each data size  
 Color[] colors = {Color.RED, Color.BLUE, Color.GREEN, Color.ORANGE, Color.MAGENTA};  
 for (int i = 0; i < dataSizes.size(); i++) {  
 int dataSize = dataSizes.get(i);  
 g2d.setColor(colors[i % colors.length]);  
  
 int prevX = 0;  
 int prevY = 0;  
 boolean first = true;  
  
 for (int j = 0; j < mapSizes.size(); j++) {  
 int mapSize = mapSizes.get(j);  
 double lookupTime = lookupData.get(dataSize).get(mapSize);

\*\*Part 5 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

float x = padding + j \* (width - 2 \* padding) / (mapSizes.size() - 1);  
 float y = height - padding - (float) (lookupTime \* (height - 2 \* padding) / maxLookupTime);  
  
 g2d.fillOval((int) x - 3, (int) y - 3, 6, 6);  
 if (!first) {  
 g2d.drawLine(prevX, prevY, (int) x, (int) y);  
 }  
  
 prevX = (int) x;  
 prevY = (int) y;  
 first = false;  
 }  
 }

\*\*Part 6 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

// Draw legend  
 int legendX = width - 200;  
 int legendY = 50;  
 for (int i = 0; i < dataSizes.size(); i++) {  
 g2d.setColor(colors[i % colors.length]);  
 g2d.fillRect(legendX, legendY + i \* 20, 10, 10);  
 g2d.setColor(Color.BLACK);  
 g2d.drawString("Data Size: " + dataSizes.get(i), legendX + 20, legendY + i \* 20 + 10);  
 }  
  
 // Draw titles  
 g2d.setFont(new Font("Arial", Font.BOLD, 16));  
 String xAxisTitle = "HashMap Size";  
 String yAxisTitle = "Lookup Time (ms)";

\*\*Part 7 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

g2d.drawString(xAxisTitle, width / 2 - fm.stringWidth(xAxisTitle) / 2, height - 10);  
 g2d.translate(15, height / 2 + fm.stringWidth(yAxisTitle) / 2);  
 g2d.rotate(-Math.PI / 2);  
 g2d.drawString(yAxisTitle, 0, 0);  
 g2d.rotate(Math.PI / 2);  
 g2d.translate(-15, -(height / 2 + fm.stringWidth(yAxisTitle) / 2));  
  
 g2d.dispose();  
  
 // Save image  
 try {  
 ImageIO.write(image, "png", new File(outputFile));  
 } catch (IOException e) {  
 System.err.println("Error saving image: " + e.getMessage());  
 System.exit(1);

\*\*Part 8 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

}  
 }  
  
 /\*\*  
 \* Create a visualization of bucket distribution and save to a file  
 \*/

\*\*Part 9 of `visualizeLookupPerformance`:\*\*  
This section handles specific sub-tasks within `visualizeLookupPerformance`, such as initializing, processing loops, or output formatting.

## visualizeBucketDistribution (Split)

public static void visualizeBucketDistribution(String csvFile, String outputFile) {  
 // Read data from CSV file  
 java.util.List<Integer> bucketCounts = new java.util.ArrayList<>();  
 try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {  
 String line = br.readLine();  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(",");  
 int count = Integer.parseInt(values[1]);  
 bucketCounts.add(count);  
 }  
 } catch (IOException e) {  
 System.err.println("Error reading CSV file: " + e.getMessage());  
 System.exit(1);  
 }

\*\*Part 1 of `visualizeBucketDistribution`:\*\*  
This section handles specific sub-tasks within `visualizeBucketDistribution`, such as initializing, processing loops, or output formatting.

// Create BufferedImage  
 int width = 800;  
 int height = 600;  
 BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);  
 Graphics2D g2d = image.createGraphics();  
 g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);  
 g2d.setColor(Color.WHITE);  
 g2d.fillRect(0, 0, width, height);  
  
 int padding = 50;  
  
 // Draw axes  
 g2d.setColor(Color.BLACK);  
 g2d.drawLine(padding, height - padding, width - padding, height - padding);  
 g2d.drawLine(padding, height - padding, padding, padding);

\*\*Part 2 of `visualizeBucketDistribution`:\*\*  
This section handles specific sub-tasks within `visualizeBucketDistribution`, such as initializing, processing loops, or output formatting.

// Find max bucket count for scaling  
 int maxCount = Math.max(Collections.max(bucketCounts), 0);  
  
 // Calculate bar width  
 int barWidth = (width - 2 \* padding) / bucketCounts.size();  
  
 // Draw bars  
 g2d.setColor(Color.BLUE);  
 for (int i = 0; i < bucketCounts.size(); i++) {  
 int count = bucketCounts.get(i);  
 int barHeight = (int) ((double) count / maxCount \* (height - 2 \* padding));  
 int x = padding + i \* barWidth;  
 g2d.fillRect(x, height - padding - barHeight, barWidth - 2, barHeight);  
 }

\*\*Part 3 of `visualizeBucketDistribution`:\*\*  
This section handles specific sub-tasks within `visualizeBucketDistribution`, such as initializing, processing loops, or output formatting.

// Draw axes labels  
 for (int i = 0; i < bucketCounts.size(); i += bucketCounts.size() / 10) {  
 int x = padding + i \* barWidth;  
 g2d.drawString(String.valueOf(i), x, height - padding + 15);  
 }  
  
 // Y-axis scale  
 for (int i = 0; i <= 10; i++) {  
 int y = height - padding - i \* (height - 2 \* padding) / 10;  
 int value = i \* maxCount / 10;  
 g2d.drawString(String.valueOf(value), padding - 30, y + 5);  
 g2d.drawLine(padding - 5, y, padding, y);  
 }

\*\*Part 4 of `visualizeBucketDistribution`:\*\*  
This section handles specific sub-tasks within `visualizeBucketDistribution`, such as initializing, processing loops, or output formatting.

// Draw titles  
 g2d.setFont(new Font("Arial", Font.BOLD, 16));  
 String xAxisTitle = "Bucket Index";  
 String yAxisTitle = "Number of Items";  
  
 g2d.drawString(xAxisTitle, width / 2 - g2d.getFontMetrics().stringWidth(xAxisTitle) / 2, height - 10);  
 g2d.translate(15, height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2);  
 g2d.rotate(-Math.PI / 2);  
 g2d.drawString(yAxisTitle, 0, 0);  
 g2d.rotate(Math.PI / 2);  
 g2d.translate(-15, -(height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2));  
  
 g2d.dispose();  
  
 // Save image

\*\*Part 5 of `visualizeBucketDistribution`:\*\*  
This section handles specific sub-tasks within `visualizeBucketDistribution`, such as initializing, processing loops, or output formatting.

try {  
 ImageIO.write(image, "png", new File(outputFile));  
 } catch (IOException e) {  
 System.err.println("Error saving image: " + e.getMessage());  
 System.exit(1);  
 }  
 }  
  
 /\*\*  
 \* Create a visualization comparing our HashMap with Java's HashMap and save to a file  
 \*/

\*\*Part 6 of `visualizeBucketDistribution`:\*\*  
This section handles specific sub-tasks within `visualizeBucketDistribution`, such as initializing, processing loops, or output formatting.

## visualizeHashMapComparison (Split)

public static void visualizeHashMapComparison(String csvFile, String outputFile) {  
 // Read data from CSV file  
 java.util.List<Integer> dataSizes = new java.util.ArrayList<>();  
 java.util.List<Double> simpleHashMapTimes = new java.util.ArrayList<>();  
 java.util.List<Double> javaHashMapTimes = new java.util.ArrayList<>();  
  
 try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {  
 String line = br.readLine();  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(",");  
 int dataSize = Integer.parseInt(values[0]);  
 double simpleTime = Double.parseDouble(values[1]);  
 double javaTime = Double.parseDouble(values[2]);  
  
 dataSizes.add(dataSize);

\*\*Part 1 of `visualizeHashMapComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashMapComparison`, such as initializing, processing loops, or output formatting.

simpleHashMapTimes.add(simpleTime);  
 javaHashMapTimes.add(javaTime);  
 }  
 } catch (IOException e) {  
 System.err.println("Error reading CSV file: " + e.getMessage());  
 System.exit(1);  
 }  
  
 // Create BufferedImage  
 int width = 800;  
 int height = 600;  
 BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);  
 Graphics2D g2d = image.createGraphics();  
 g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);  
 g2d.setColor(Color.WHITE);

\*\*Part 2 of `visualizeHashMapComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashMapComparison`, such as initializing, processing loops, or output formatting.

g2d.fillRect(0, 0, width, height);  
  
 int padding = 50;  
  
 // Draw axes  
 g2d.setColor(Color.BLACK);  
 g2d.drawLine(padding, height - padding, width - padding, height - padding);  
 g2d.drawLine(padding, height - padding, padding, padding);  
  
 // Find max lookup time value for scaling  
 double maxTime = Math.max(  
 simpleHashMapTimes.stream().mapToDouble(Double::doubleValue).max().orElse(0),  
 javaHashMapTimes.stream().mapToDouble(Double::doubleValue).max().orElse(0)  
 );

\*\*Part 3 of `visualizeHashMapComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashMapComparison`, such as initializing, processing loops, or output formatting.

// Draw data points and lines  
 g2d.setColor(Color.RED);  
 drawLine(g2d, dataSizes, simpleHashMapTimes, maxTime, width, height, padding);  
  
 g2d.setColor(Color.BLUE);  
 drawLine(g2d, dataSizes, javaHashMapTimes, maxTime, width, height, padding);  
  
 // Draw x-axis labels  
 for (int i = 0; i < dataSizes.size(); i++) {  
 int dataSize = dataSizes.get(i);  
 String label = String.valueOf(dataSize);  
 int labelWidth = g2d.getFontMetrics().stringWidth(label);  
 float x = padding + i \* (width - 2 \* padding) / (dataSizes.size() - 1);  
 g2d.setColor(Color.BLACK);  
 g2d.drawString(label, x - labelWidth / 2, height - padding + 15);

\*\*Part 4 of `visualizeHashMapComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashMapComparison`, such as initializing, processing loops, or output formatting.

}  
  
 // Draw legend  
 g2d.setColor(Color.RED);  
 g2d.fillRect(width - 200, 50, 10, 10);  
 g2d.setColor(Color.BLACK);  
 g2d.drawString("SimpleHashMap", width - 180, 60);  
  
 g2d.setColor(Color.BLUE);  
 g2d.fillRect(width - 200, 70, 10, 10);  
 g2d.setColor(Color.BLACK);  
 g2d.drawString("Java HashMap", width - 180, 80);  
  
 // Draw titles  
 g2d.setFont(new Font("Arial", Font.BOLD, 16));

\*\*Part 5 of `visualizeHashMapComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashMapComparison`, such as initializing, processing loops, or output formatting.

String xAxisTitle = "Data Size";  
 String yAxisTitle = "Lookup Time (ms)";  
  
 g2d.drawString(xAxisTitle, width / 2 - g2d.getFontMetrics().stringWidth(xAxisTitle) / 2, height - 10);  
 g2d.translate(15, height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2);  
 g2d.rotate(-Math.PI / 2);  
 g2d.drawString(yAxisTitle, 0, 0);  
 g2d.rotate(Math.PI / 2);  
 g2d.translate(-15, -(height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2));  
  
 g2d.dispose();  
  
 // Save image  
 try {  
 ImageIO.write(image, "png", new File(outputFile));

\*\*Part 6 of `visualizeHashMapComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashMapComparison`, such as initializing, processing loops, or output formatting.

} catch (IOException e) {  
 System.err.println("Error saving image: " + e.getMessage());  
 System.exit(1);  
 }  
 }

\*\*Part 7 of `visualizeHashMapComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashMapComparison`, such as initializing, processing loops, or output formatting.

## drawLine (Split)

private static void drawLine(Graphics2D g2d, java.util.List<Integer> xValues, java.util.List<Double> yValues,  
 double maxY, int width, int height, int padding) {  
 int prevX = 0;  
 int prevY = 0;  
 boolean first = true;  
  
 for (int i = 0; i < xValues.size(); i++) {  
 float x = padding + i \* (width - 2 \* padding) / (xValues.size() - 1);  
 float y = height - padding - (float) (yValues.get(i) \* (height - 2 \* padding) / maxY);  
  
 g2d.fillOval((int) x - 3, (int) y - 3, 6, 6);  
 if (!first) {  
 g2d.drawLine(prevX, prevY, (int) x, (int) y);  
 }

\*\*Part 1 of `drawLine`:\*\*  
This section handles specific sub-tasks within `drawLine`, such as initializing, processing loops, or output formatting.

prevX = (int) x;  
 prevY = (int) y;  
 first = false;  
 }  
 }  
  
 /\*\*  
 \* Visualize hash function comparison data and save to a file  
 \*/

\*\*Part 2 of `drawLine`:\*\*  
This section handles specific sub-tasks within `drawLine`, such as initializing, processing loops, or output formatting.

## visualizeHashFunctionComparison (Split)

public static void visualizeHashFunctionComparison(String csvFile, String outputFile) {  
 // Read data from CSV file  
 java.util.List<String> hashFunctions = new java.util.ArrayList<>();  
 java.util.List<Integer> collisions = new java.util.ArrayList<>();  
 java.util.List<Integer> maxBucketSizes = new java.util.ArrayList<>();  
 java.util.List<Integer> emptyBuckets = new java.util.ArrayList<>();  
  
 try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {  
 String line = br.readLine();  
 while ((line = br.readLine()) != null) {  
 String[] values = line.split(",");  
 hashFunctions.add(values[0]);  
 collisions.add(Integer.parseInt(values[1]));  
 maxBucketSizes.add(Integer.parseInt(values[2]));  
 emptyBuckets.add(Integer.parseInt(values[3]));

\*\*Part 1 of `visualizeHashFunctionComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashFunctionComparison`, such as initializing, processing loops, or output formatting.

}  
 } catch (IOException e) {  
 System.err.println("Error reading CSV file: " + e.getMessage());  
 System.exit(1);  
 }  
  
 // Create BufferedImage  
 int width = 800;  
 int height = 600;  
 BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);  
 Graphics2D

\*\*Part 2 of `visualizeHashFunctionComparison`:\*\*  
This section handles specific sub-tasks within `visualizeHashFunctionComparison`, such as initializing, processing loops, or output formatting.

# HashMapper.java

Methods and classes defined in this file are shown below.

## {

public class HashMapper {  
  
 // Different hash function types

\*\*Detailed Description:\*\*   
The method `{` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## Length";

private static String selectedHashFunction = "String Length";  
  
 // Set the hash function type

\*\*Detailed Description:\*\*   
The method `Length";` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## setHashFunction

public static void setHashFunction(String hashFunctionType) {  
 selectedHashFunction = hashFunctionType;  
 }  
  
 // Inner class for our dumb hash map implementation  
 static class DumbHashMap<K, V> {

\*\*Detailed Description:\*\*   
The method `setHashFunction` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## {

private static class Entry<K, V> {  
 K key;  
 V value;  
 int collisionLevel; // Track how many collisions occurred for this entry  
  
 Entry(K key, V value, int collisionLevel) {  
 this.key = key;  
 this.value = value;  
 this.collisionLevel = collisionLevel;  
 }  
 }

\*\*Detailed Description:\*\*   
The method `{` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## buckets;

private ArrayList<Entry<K, V>>[] buckets;

\*\*Detailed Description:\*\*   
The method `buckets;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## size;

private final int size;

\*\*Detailed Description:\*\*   
The method `size;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## collisions;

private int collisions;

\*\*Detailed Description:\*\*   
The method `collisions;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## collisionDistribution;

private Map<Integer, Integer> collisionDistribution;

\*\*Detailed Description:\*\*   
The method `collisionDistribution;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## 0;

private int maxCollisionLevel = 0;  
  
 @SuppressWarnings("unchecked")

\*\*Detailed Description:\*\*   
The method `0;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## DumbHashMap

public DumbHashMap(int size) {  
 this.size = size;  
 this.buckets = new ArrayList[size];  
 this.collisions = 0;  
 this.collisionDistribution = new HashMap<>();  
  
 for (int i = 0; i < size; i++) {  
 buckets[i] = new ArrayList<>();  
 }  
 }  
  
 // Our intentionally poor hash function

\*\*Detailed Description:\*\*   
The method `DumbHashMap` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## dumbHash (Split)

private int dumbHash(K key) {  
 if (key == null) {  
 return 0;  
 }  
  
 // For strings  
 if (key instanceof String) {  
 String str = (String) key;  
 if (str.isEmpty()) {  
 return 0;  
 }  
  
 // Use different hash functions based on selection  
 switch (selectedHashFunction) {  
 case "String Length":

\*\*Part 1 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

return str.length() % size;  
  
 case "First Character":  
 return str.charAt(0) % size;  
  
 case "First + Last Character":  
 if (str.length() > 1) {  
 return (str.charAt(0) + str.charAt(str.length() - 1)) % size;  
 } else {  
 return str.charAt(0) % size;  
 }  
  
 case "Character Sum":  
 int sum = 0;  
 for (char c : str.toCharArray()) {

\*\*Part 2 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

sum += c;  
 }  
 return sum % size;  
  
 case "Random":  
 // Pseudo-random but deterministic based on first and last chars  
 if (str.length() > 1) {  
 return ((str.charAt(0) \* 31) ^ str.charAt(str.length() - 1)) % size;  
 } else {  
 return str.charAt(0) % size;  
 }  
  
 default:  
 return str.length() % size;  
 }

\*\*Part 3 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

}  
  
 // For integers, just mod with size (very poor distribution)  
 if (key instanceof Integer) {  
 Integer num = (Integer) key;  
 return Math.abs(num) % size;  
 }  
  
 // For other types, use a very basic approach  
 String keyString = key.toString();  
 if (keyString.isEmpty()) {  
 return 0;  
 }  
  
 // Just use first and last character

\*\*Part 4 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

int hash = keyString.charAt(0);  
 if (keyString.length() > 1) {  
 hash += keyString.charAt(keyString.length() - 1);  
 }  
  
 return Math.abs(hash) % size;  
 }

\*\*Part 5 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

## put (Split)

public void put(K key, V value) {  
 int index = dumbHash(key);  
 ArrayList<Entry<K, V>> bucket = buckets[index];  
  
 // Check if key already exists  
 for (int i = 0; i < bucket.size(); i++) {  
 if (Objects.equals(bucket.get(i).key, key)) {  
 bucket.get(i).value = value;  
 return;  
 }  
 }  
  
 // New entry - check for collision and track collision level  
 int collisionLevel = 0;  
 if (!bucket.isEmpty()) {

\*\*Part 1 of `put`:\*\*  
This section handles specific sub-tasks within `put`, such as initializing, processing loops, or output formatting.

collisions++;  
 collisionLevel = bucket.size();  
  
 // Update collision distribution  
 collisionDistribution.put(collisionLevel,  
 collisionDistribution.getOrDefault(collisionLevel, 0) + 1);  
  
 // Track maximum collision level  
 maxCollisionLevel = Math.max(maxCollisionLevel, collisionLevel);  
 }  
  
 bucket.add(new Entry<>(key, value, collisionLevel));  
 }

\*\*Part 2 of `put`:\*\*  
This section handles specific sub-tasks within `put`, such as initializing, processing loops, or output formatting.

## get

public V get(K key) {  
 int index = dumbHash(key);  
 ArrayList<Entry<K, V>> bucket = buckets[index];  
  
 for (Entry<K, V> entry : bucket) {  
 if (Objects.equals(entry.key, key)) {  
 return entry.value;  
 }  
 }  
  
 return null; // Key not found  
 }

\*\*Detailed Description:\*\*   
The method `get` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getCollisionCount

public int getCollisionCount() {  
 return collisions;  
 }

\*\*Detailed Description:\*\*   
The method `getCollisionCount` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getCollisionDistribution

public Map<Integer, Integer> getCollisionDistribution() {  
 return collisionDistribution;  
 }

\*\*Detailed Description:\*\*   
The method `getCollisionDistribution` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getBucketDistribution

public int[] getBucketDistribution() {  
 int[] distribution = new int[size];  
  
 for (int i = 0; i < size; i++) {  
 distribution[i] = buckets[i].size();  
 }  
  
 return distribution;  
 }

\*\*Detailed Description:\*\*   
The method `getBucketDistribution` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getMaxCollisionLevel

public int getMaxCollisionLevel() {  
 return maxCollisionLevel;  
 }

\*\*Detailed Description:\*\*   
The method `getMaxCollisionLevel` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getBucketSize

public int getBucketSize(int index) {  
 if (index < 0 || index >= size) {  
 return 0;  
 }  
 return buckets[index].size();  
 }  
  
 // Get bucket entries for advanced analysis

\*\*Detailed Description:\*\*   
The method `getBucketSize` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getBucketEntries

public java.util.List<Entry<K, V>> getBucketEntries(int index) {  
 if (index < 0 || index >= size) {  
 return new ArrayList<>();  
 }  
 return new ArrayList<>(buckets[index]);  
 }  
  
 // Get all entries for analysis

\*\*Detailed Description:\*\*   
The method `getBucketEntries` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getAllEntries

public java.util.List<Entry<K, V>> getAllEntries() {  
 java.util.List<Entry<K, V>> allEntries = new ArrayList<>();  
 for (ArrayList<Entry<K, V>> bucket : buckets) {  
 allEntries.addAll(bucket);  
 }  
 return allEntries;  
 }  
 }  
  
 /\*\*  
 \* Class for creating a visual fingerprint from text  
 \*/  
 static class TextVisualizer {  
  
 // Process a text and generate a visual fingerprint

\*\*Detailed Description:\*\*   
The method `getAllEntries` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## createVisualFingerprint (Split)

public static BufferedImage createVisualFingerprint(String text, int size) {  
 DumbHashMap<String, Integer> wordMap = new DumbHashMap<>(size);  
 DumbHashMap<Character, Integer> charMap = new DumbHashMap<>(size);  
  
 // Process words  
 String[] words = text.split("\\s+");  
 for (String word : words) {  
 word = word.toLowerCase().replaceAll("[^a-z]", "");  
 if (!word.isEmpty()) {  
 wordMap.put(word, 1);  
 }  
 }  
  
 // Process characters  
 for (char c : text.toCharArray()) {

\*\*Part 1 of `createVisualFingerprint`:\*\*  
This section handles specific sub-tasks within `createVisualFingerprint`, such as initializing, processing loops, or output formatting.

if (Character.isLetterOrDigit(c)) {  
 charMap.put(c, 1);  
 }  
 }  
  
 // Create image  
 BufferedImage image = new BufferedImage(size, size, BufferedImage.TYPE\_INT\_RGB);  
 Graphics2D g = image.createGraphics();  
  
 // Fill background  
 g.setColor(Color.BLACK);  
 g.fillRect(0, 0, size, size);  
  
 // Map bucket distribution to colors  
 int[] wordDist = wordMap.getBucketDistribution();

\*\*Part 2 of `createVisualFingerprint`:\*\*  
This section handles specific sub-tasks within `createVisualFingerprint`, such as initializing, processing loops, or output formatting.

int[] charDist = charMap.getBucketDistribution();  
  
 // Find max values for scaling  
 int maxWord = 0;  
 int maxChar = 0;  
 for (int i = 0; i < size; i++) {  
 maxWord = Math.max(maxWord, wordDist[i]);  
 maxChar = Math.max(maxChar, charDist[i]);  
 }  
  
 // Draw fingerprint  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 // Calculate color intensity based on bucket sizes  
 int wordIntensity = (int) (255.0 \* wordDist[i] / (maxWord > 0 ? maxWord : 1));

\*\*Part 3 of `createVisualFingerprint`:\*\*  
This section handles specific sub-tasks within `createVisualFingerprint`, such as initializing, processing loops, or output formatting.

int charIntensity = (int) (255.0 \* charDist[j] / (maxChar > 0 ? maxChar : 1));  
  
 // Create color: red channel from word distribution, blue from char distribution  
 Color color = new Color(  
 wordIntensity,  
 (wordIntensity + charIntensity) / 4,  
 charIntensity  
 );  
  
 image.setRGB(i, j, color.getRGB());  
 }  
 }  
  
 g.dispose();  
 return image;

\*\*Part 4 of `createVisualFingerprint`:\*\*  
This section handles specific sub-tasks within `createVisualFingerprint`, such as initializing, processing loops, or output formatting.

}  
  
 // Apply salt and smooth algorithms to the fingerprint

\*\*Part 5 of `createVisualFingerprint`:\*\*  
This section handles specific sub-tasks within `createVisualFingerprint`, such as initializing, processing loops, or output formatting.

## saltAndSmooth (Split)

public static BufferedImage saltAndSmooth(BufferedImage original, double saltLevel, int smoothRadius) {  
 int width = original.getWidth();  
 int height = original.getHeight();  
  
 // Create a copy of the original image  
 BufferedImage result = new BufferedImage(width, height, original.getType());  
  
 // Apply salt  
 Random random = new Random();  
 for (int x = 0; x < width; x++) {  
 for (int y = 0; y < height; y++) {  
 if (random.nextDouble() < saltLevel) {  
 // Apply salt (random noise)  
 result.setRGB(x, y, new Color(  
 random.nextInt(256),

\*\*Part 1 of `saltAndSmooth`:\*\*  
This section handles specific sub-tasks within `saltAndSmooth`, such as initializing, processing loops, or output formatting.

random.nextInt(256),  
 random.nextInt(256)  
 ).getRGB());  
 } else {  
 // Copy original pixel  
 result.setRGB(x, y, original.getRGB(x, y));  
 }  
 }  
 }  
  
 // Smooth using a Gaussian-like blur  
 BufferedImage smoothed = new BufferedImage(width, height, original.getType());  
 for (int x = 0; x < width; x++) {  
 for (int y = 0; y < height; y++) {  
 // Get average of nearby pixels

\*\*Part 2 of `saltAndSmooth`:\*\*  
This section handles specific sub-tasks within `saltAndSmooth`, such as initializing, processing loops, or output formatting.

int totalR = 0, totalG = 0, totalB = 0;  
 int count = 0;  
  
 for (int dx = -smoothRadius; dx <= smoothRadius; dx++) {  
 for (int dy = -smoothRadius; dy <= smoothRadius; dy++) {  
 int nx = x + dx;  
 int ny = y + dy;  
  
 if (nx >= 0 && nx < width && ny >= 0 && ny < height) {  
 Color pixel = new Color(result.getRGB(nx, ny));  
 totalR += pixel.getRed();  
 totalG += pixel.getGreen();  
 totalB += pixel.getBlue();  
 count++;  
 }

\*\*Part 3 of `saltAndSmooth`:\*\*  
This section handles specific sub-tasks within `saltAndSmooth`, such as initializing, processing loops, or output formatting.

}  
 }  
  
 // Set pixel to average color  
 Color avgColor = new Color(  
 totalR / count,  
 totalG / count,  
 totalB / count  
 );  
 smoothed.setRGB(x, y, avgColor.getRGB());  
 }  
 }  
  
 return smoothed;  
 }

\*\*Part 4 of `saltAndSmooth`:\*\*  
This section handles specific sub-tasks within `saltAndSmooth`, such as initializing, processing loops, or output formatting.

}  
  
 /\*\*  
 \* Class for analyzing text patterns using collision data  
 \*/  
 static class TextAnalyzer {  
  
 // Analyze a text and return statistics

\*\*Part 5 of `saltAndSmooth`:\*\*  
This section handles specific sub-tasks within `saltAndSmooth`, such as initializing, processing loops, or output formatting.

## analyzeText (Split)

public static Map<String, Object> analyzeText(String text, int mapSize) {  
 DumbHashMap<String, Integer> wordFreq = new DumbHashMap<>(mapSize);  
  
 // Process words and count frequency  
 String[] words = text.split("\\s+");  
 Map<String, Integer> actualFreq = new HashMap<>();  
  
 for (String word : words) {  
 word = word.toLowerCase().replaceAll("[^a-z]", "");  
 if (!word.isEmpty()) {  
 actualFreq.put(word, actualFreq.getOrDefault(word, 0) + 1);  
 wordFreq.put(word, actualFreq.get(word));  
 }  
 }

\*\*Part 1 of `analyzeText`:\*\*  
This section handles specific sub-tasks within `analyzeText`, such as initializing, processing loops, or output formatting.

// Calculate statistics  
 Map<String, Object> stats = new HashMap<>();  
 stats.put("totalWords", words.length);  
 stats.put("uniqueWords", actualFreq.size());  
 stats.put("collisions", wordFreq.getCollisionCount());  
 stats.put("collisionDistribution", wordFreq.getCollisionDistribution());  
 stats.put("bucketDistribution", wordFreq.getBucketDistribution());  
 stats.put("maxCollisionLevel", wordFreq.getMaxCollisionLevel());  
  
 return stats;  
 }  
  
 // Generate a collision spectrum chart and save to a file

\*\*Part 2 of `analyzeText`:\*\*  
This section handles specific sub-tasks within `analyzeText`, such as initializing, processing loops, or output formatting.

## generateCollisionSpectrum (Split)

public static void generateCollisionSpectrum(Map<String, Object> stats, String outputFile) {  
 // Extract collision distribution  
 @SuppressWarnings("unchecked")  
 Map<Integer, Integer> collisionDist = (Map<Integer, Integer>) stats.get("collisionDistribution");  
  
 // Create dataset  
 XYSeries series = new XYSeries("Collision Spectrum");  
 for (Map.Entry<Integer, Integer> entry : collisionDist.entrySet()) {  
 series.add(entry.getKey(), entry.getValue());  
 }  
  
 XYSeriesCollection dataset = new XYSeriesCollection();  
 dataset.addSeries(series);  
  
 // Create chart

\*\*Part 1 of `generateCollisionSpectrum`:\*\*  
This section handles specific sub-tasks within `generateCollisionSpectrum`, such as initializing, processing loops, or output formatting.

JFreeChart chart = ChartFactory.createXYLineChart(  
 "Text Collision Spectrum",  
 "Collision Level",  
 "Frequency",  
 dataset,  
 PlotOrientation.VERTICAL,  
 true,  
 true,  
 false  
 );  
  
 // Customize appearance  
 XYPlot plot = chart.getXYPlot();  
 XYLineAndShapeRenderer renderer = new XYLineAndShapeRenderer();  
 renderer.setSeriesPaint(0, Color.RED);

\*\*Part 2 of `generateCollisionSpectrum`:\*\*  
This section handles specific sub-tasks within `generateCollisionSpectrum`, such as initializing, processing loops, or output formatting.

renderer.setSeriesStroke(0, new BasicStroke(2.0f));  
 plot.setRenderer(renderer);  
  
 // Save chart to file  
 try {  
 ChartUtils.saveChartAsPNG(new File(outputFile), chart, 800, 600);  
 } catch (IOException e) {  
 System.err.println("Error saving chart: " + e.getMessage());  
 System.exit(1);  
 }  
 }  
  
 // Compare two texts and calculate similarity based on collision patterns

\*\*Part 3 of `generateCollisionSpectrum`:\*\*  
This section handles specific sub-tasks within `generateCollisionSpectrum`, such as initializing, processing loops, or output formatting.

## calculateSimilarity (Split)

public static double calculateSimilarity(String text1, String text2, int mapSize) {  
 Map<String, Object> stats1 = analyzeText(text1, mapSize);  
 Map<String, Object> stats2 = analyzeText(text2, mapSize);  
  
 int[] bucketDist1 = (int[]) stats1.get("bucketDistribution");  
 int[] bucketDist2 = (int[]) stats2.get("bucketDistribution");  
  
 // Calculate Euclidean distance between bucket distributions  
 double sumSquaredDiff = 0;  
 for (int i = 0; i < mapSize; i++) {  
 sumSquaredDiff += Math.pow(bucketDist1[i] - bucketDist2[i], 2);  
 }  
  
 double distance = Math.sqrt(sumSquaredDiff);

\*\*Part 1 of `calculateSimilarity`:\*\*  
This section handles specific sub-tasks within `calculateSimilarity`, such as initializing, processing loops, or output formatting.

// Convert distance to similarity (inverse and normalized)  
 Object uniqueWords1 = stats1.get("uniqueWords");  
 Object uniqueWords2 = stats2.get("uniqueWords");  
 double maxUniqueWords = Math.max(  
 (uniqueWords1 instanceof Number) ? ((Number) uniqueWords1).doubleValue() : 0,  
 (uniqueWords2 instanceof Number) ? ((Number) uniqueWords2).doubleValue() : 0  
 );  
  
 double maxPossibleDistance = Math.sqrt(mapSize \* Math.pow(maxUniqueWords, 2));  
  
 // Similarity between 0 and 1  
 return 1.0 - (distance / maxPossibleDistance);  
 }  
 }

\*\*Part 2 of `calculateSimilarity`:\*\*  
This section handles specific sub-tasks within `calculateSimilarity`, such as initializing, processing loops, or output formatting.

/\*\*  
 \* Plotter class that creates advanced visualizations using Octave-like styling  
 \*/  
 static class Plotter {  
  
 // Create a 3D visualization of text fingerprint

\*\*Part 3 of `calculateSimilarity`:\*\*  
This section handles specific sub-tasks within `calculateSimilarity`, such as initializing, processing loops, or output formatting.

## create3DPlot (Split)

public static BufferedImage create3DPlot(int[] bucketDistribution, int size) {  
 // Create a 3D representation of the bucket distribution  
 double[][] data = new double[size][size];  
  
 // Map the 1D bucket distribution to a 2D grid  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 // Create an interesting pattern using the bucket data  
 double value1 = bucketDistribution[i];  
 double value2 = bucketDistribution[j];  
  
 // Create a surface by combining values, avoid NaN  
 double product = value1 \* value2;  
 data[i][j] = (product > 0) ? Math.sqrt(product) \* Math.sin(i \* j / (double) (size \* size) \* Math.PI) : 0;  
 }

\*\*Part 1 of `create3DPlot`:\*\*  
This section handles specific sub-tasks within `create3DPlot`, such as initializing, processing loops, or output formatting.

}  
  
 // Normalize data for better visualization  
 double maxVal = Double.MIN\_VALUE;  
 double minVal = Double.MAX\_VALUE;  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 maxVal = Math.max(maxVal, data[i][j]);  
 minVal = Math.min(minVal, data[i][j]);  
 }  
 }  
  
 // Handle case where maxVal equals minVal to avoid division by zero  
 if (maxVal == minVal) {  
 maxVal = minVal + 1.0;

\*\*Part 2 of `create3DPlot`:\*\*  
This section handles specific sub-tasks within `create3DPlot`, such as initializing, processing loops, or output formatting.

}  
  
 // Create a BufferedImage to visualize the 3D surface with color mapping  
 BufferedImage image = new BufferedImage(size, size, BufferedImage.TYPE\_INT\_RGB);  
 Graphics2D g2d = image.createGraphics();  
  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 // Normalize value between 0 and 1  
 double normValue = (data[i][j] - minVal) / (maxVal - minVal);  
  
 // Use a color gradient (blue to red, through green)  
 Color color;  
 if (normValue < 0.5) {  
 // Blue to green

\*\*Part 3 of `create3DPlot`:\*\*  
This section handles specific sub-tasks within `create3DPlot`, such as initializing, processing loops, or output formatting.

int green = (int) (normValue \* 2 \* 255);  
 color = new Color(0, green, 255 - green);  
 } else {  
 // Green to red  
 int red = (int) ((normValue - 0.5) \* 2 \* 255);  
 color = new Color(red, 255 - red, 0);  
 }  
  
 image.setRGB(i, j, color.getRGB());  
 }  
 }  
  
 // Add a grid to make it look like an octave plot  
 g2d.setColor(new Color(255, 255, 255, 50)); // Translucent white  
 int gridSize = size / 20; // Scale grid to image size

\*\*Part 4 of `create3DPlot`:\*\*  
This section handles specific sub-tasks within `create3DPlot`, such as initializing, processing loops, or output formatting.

if (gridSize > 0) {  
 for (int i = 0; i <= size; i += gridSize) {  
 g2d.drawLine(i, 0, i, size);  
 g2d.drawLine(0, i, size, i);  
 }  
 }  
  
 g2d.dispose();  
 return image;  
 }

\*\*Part 5 of `create3DPlot`:\*\*  
This section handles specific sub-tasks within `create3DPlot`, such as initializing, processing loops, or output formatting.

# SimpleHashMap.java

Methods and classes defined in this file are shown below.

## {

public class SimpleHashMap<K, V> {  
  
 // Inner class for storing key-value pairs

\*\*Detailed Description:\*\*   
The method `{` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## {

private static class Entry<K, V> {  
 K key;  
 V value;  
  
 Entry(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
  
 @Override

\*\*Detailed Description:\*\*   
The method `{` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## equals

public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Entry<?, ?> entry = (Entry<?, ?>) o;  
 return Objects.equals(key, entry.key);  
 }  
  
 @Override

\*\*Detailed Description:\*\*   
The method `equals` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## hashCode

public int hashCode() {  
 return Objects.hashCode(key);  
 }  
 }

\*\*Detailed Description:\*\*   
The method `hashCode` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## buckets;

private ArrayList<Entry<K, V>>[] buckets;

\*\*Detailed Description:\*\*   
The method `buckets;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## size;

private final int size;

\*\*Detailed Description:\*\*   
The method `size;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## collisions;

private int collisions;

\*\*Detailed Description:\*\*   
The method `collisions;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## itemCount;

private int itemCount;

\*\*Detailed Description:\*\*   
The method `itemCount;` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## Length";

private static String hashFunctionType = "String Length";  
  
 /\*\*  
 \* Constructor with default size of 16  
 \*/  
 @SuppressWarnings("unchecked")

\*\*Detailed Description:\*\*   
The method `Length";` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## SimpleHashMap

public SimpleHashMap() {  
 this(16);  
 }  
  
 /\*\*  
 \* Constructor with specified size  
 \*/  
 @SuppressWarnings("unchecked")

\*\*Detailed Description:\*\*   
The method `SimpleHashMap` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## SimpleHashMap

public SimpleHashMap(int size) {  
 this.size = size;  
 this.buckets = new ArrayList[size];  
 this.collisions = 0;  
 this.itemCount = 0;  
  
 // Initialize buckets  
 for (int i = 0; i < size; i++) {  
 buckets[i] = new ArrayList<>();  
 }  
 }  
  
 /\*\*  
 \* Set the hash function type  
 \*/

\*\*Detailed Description:\*\*   
The method `SimpleHashMap` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## setHashFunctionType

public static void setHashFunctionType(String type) {  
 hashFunctionType = type;  
 }  
  
 /\*\*  
 \* Dumb hash function - intentionally simplified and inefficient  
 \* This function is designed to demonstrate collision behaviors  
 \*/

\*\*Detailed Description:\*\*   
The method `setHashFunctionType` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## dumbHash (Split)

private int dumbHash(K key) {  
 if (key == null) {  
 return 0;  
 }  
  
 // For strings  
 if (key instanceof String) {  
 String str = (String) key;  
 if (str.isEmpty()) {  
 return 0;  
 }  
  
 // Use different hash functions based on selection  
 switch (hashFunctionType) {  
 case "String Length":

\*\*Part 1 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

return str.length() % size;  
  
 case "First Character":  
 return str.charAt(0) % size;  
  
 case "First + Last Character":  
 if (str.length() > 1) {  
 return (str.charAt(0) + str.charAt(str.length() - 1)) % size;  
 } else {  
 return str.charAt(0) % size;  
 }  
  
 case "Character Sum":  
 int sum = 0;  
 for (char c : str.toCharArray()) {

\*\*Part 2 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

sum += c;  
 }  
 return sum % size;  
  
 case "Random":  
 // Pseudo-random but deterministic based on first and last chars  
 if (str.length() > 1) {  
 return ((str.charAt(0) \* 31) ^ str.charAt(str.length() - 1)) % size;  
 } else {  
 return str.charAt(0) % size;  
 }  
  
 default:  
 return str.length() % size;  
 }

\*\*Part 3 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

}  
  
 // For integers  
 if (key instanceof Integer) {  
 Integer num = (Integer) key;  
 return Math.abs(num) % size;  
 }  
  
 // For other types, use a very basic approach  
 String keyString = key.toString();  
 if (keyString.isEmpty()) {  
 return 0;  
 }  
  
 // Just use first and last character

\*\*Part 4 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

int hash = keyString.charAt(0);  
 if (keyString.length() > 1) {  
 hash += keyString.charAt(keyString.length() - 1);  
 }  
  
 return Math.abs(hash) % size;  
 }  
  
 /\*\*  
 \* Hash function wrapper - uses dumbHash for this implementation  
 \*/

\*\*Part 5 of `dumbHash`:\*\*  
This section handles specific sub-tasks within `dumbHash`, such as initializing, processing loops, or output formatting.

## hash

private int hash(K key) {  
 return dumbHash(key);  
 }  
  
 /\*\*  
 \* Insert or update a key-value pair  
 \*/

\*\*Detailed Description:\*\*   
The method `hash` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## put (Split)

public void put(K key, V value) {  
 int index = hash(key);  
 ArrayList<Entry<K, V>> bucket = buckets[index];  
  
 // Check if key already exists  
 for (int i = 0; i < bucket.size(); i++) {  
 if (Objects.equals(bucket.get(i).key, key)) {  
 bucket.get(i).value = value; // Update existing value  
 return;  
 }  
 }  
  
 // New entry - check for collision  
 if (!bucket.isEmpty()) {  
 collisions++;

\*\*Part 1 of `put`:\*\*  
This section handles specific sub-tasks within `put`, such as initializing, processing loops, or output formatting.

}  
  
 // Add new entry  
 bucket.add(new Entry<>(key, value));  
 itemCount++;  
 }  
  
 /\*\*  
 \* Get a value by key  
 \*/

\*\*Part 2 of `put`:\*\*  
This section handles specific sub-tasks within `put`, such as initializing, processing loops, or output formatting.

## get

public V get(K key) {  
 int index = hash(key);  
 ArrayList<Entry<K, V>> bucket = buckets[index];  
  
 // Search for key in bucket  
 for (Entry<K, V> entry : bucket) {  
 if (Objects.equals(entry.key, key)) {  
 return entry.value;  
 }  
 }  
  
 return null; // Key not found  
 }  
  
 /\*\*  
 \* Remove a key-value pair  
 \*/

\*\*Detailed Description:\*\*   
The method `get` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## remove

public boolean remove(K key) {  
 int index = hash(key);  
 ArrayList<Entry<K, V>> bucket = buckets[index];  
  
 // Find and remove entry  
 for (int i = 0; i < bucket.size(); i++) {  
 if (Objects.equals(bucket.get(i).key, key)) {  
 bucket.remove(i);  
 itemCount--;  
 return true;  
 }  
 }  
  
 return false; // Key not found  
 }  
  
 /\*\*  
 \* Check if key exists  
 \*/

\*\*Detailed Description:\*\*   
The method `remove` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## containsKey

public boolean containsKey(K key) {  
 int index = hash(key);  
 ArrayList<Entry<K, V>> bucket = buckets[index];  
  
 for (Entry<K, V> entry : bucket) {  
 if (Objects.equals(entry.key, key)) {  
 return true;  
 }  
 }  
  
 return false;  
 }  
  
 /\*\*  
 \* Get all keys in the HashMap  
 \*/

\*\*Detailed Description:\*\*   
The method `containsKey` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## keys

public List<K> keys() {  
 List<K> allKeys = new ArrayList<>();  
  
 for (ArrayList<Entry<K, V>> bucket : buckets) {  
 for (Entry<K, V> entry : bucket) {  
 allKeys.add(entry.key);  
 }  
 }  
  
 return allKeys;  
 }  
  
 /\*\*  
 \* Get number of items in the HashMap  
 \*/

\*\*Detailed Description:\*\*   
The method `keys` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## size

public int size() {  
 return itemCount;  
 }  
  
 /\*\*  
 \* Get number of collisions that occurred  
 \*/

\*\*Detailed Description:\*\*   
The method `size` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getCollisionCount

public int getCollisionCount() {  
 return collisions;  
 }  
  
 /\*\*  
 \* Get current load factor  
 \*/

\*\*Detailed Description:\*\*   
The method `getCollisionCount` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getLoadFactor

public double getLoadFactor() {  
 return (double) itemCount / size;  
 }  
  
 /\*\*  
 \* Get distribution of items across buckets  
 \*/

\*\*Detailed Description:\*\*   
The method `getLoadFactor` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.

## getBucketDistribution

public int[] getBucketDistribution() {  
 int[] distribution = new int[size];  
  
 for (int i = 0; i < size; i++) {  
 distribution[i] = buckets[i].size();  
 }  
  
 return distribution;  
 }  
}

\*\*Detailed Description:\*\*   
The method `getBucketDistribution` is essential for this class. It processes user input, manages hashmaps, controls text analysis, or generates output depending on its context.  
It usually involves steps like condition checks, iterations, object manipulations, and output processing.