COMPARISON OF FIRST-FIT AND BEST-FIT MEMORY ALLOCATION STRATEGIES

1. Introduction

Efficient memory allocation is vital in operating systems as it affects both performance and resource management. This report evaluates the effectiveness of **First-Fit** and **Best-Fit** allocation methods, focusing on **execution speed** and **memory efficiency**.

2. Overview of the Algorithms

First-Fit Method:

- Scans memory sequentially and assigns the first block that meets the required size.
- Stops searching as soon as a suitable block is located.
- Quicker allocation process but may lead to fragmented memory.

Best-Fit Method:

- Checks all available blocks and selects the smallest one that can accommodate the request.
- Minimizes leftover free space but takes longer to execute.
- Helps reduce fragmentation but comes with increased processing time.

3. Performance Analysis

Algorithm	Best Case	Average Case	Worst Case
First-Fit	O(1)	O(n)	O(n)
Best-Fit	O(n)	O(n log n)	O(n log n)

- **First-Fit** is highly efficient when a free block is available early in the list, achieving a best-case complexity of **O(1)**.
- **Best-Fit** requires scanning all blocks to find an optimal match, making its worst-case complexity **O(n log n)**.
- First-Fit operates faster due to its straightforward approach, whereas Best-Fit incurs more computational overhead.

4. Memory Efficiency

Memory efficiency refers to the proportion of allocated memory relative to total memory capacity. In our study:

- **First-Fit**: Allocates memory faster but often leaves behind small unusable free spaces, reducing overall efficiency.
- **Best-Fit**: Maximizes space utilization by minimizing fragmentation but takes longer to process allocations.

Observations from the Simulation:

Algorithm	Success Rate of Allocation	Memory Usage (%)
First-Fit	Moderate	60-80%
Best-Fit	High	75-90%

- **Best-Fit** uses memory more effectively by optimizing block allocation.
- **First-Fit** may lead to early memory exhaustion due to fragmentation, even if enough total memory remains.

5. Conclusion

Factor	First-Fit	Best-Fit
Processing Speed	Faster	Slower
Memory Efficiency	Moderate	Higher
Fragmentation	More External Fragmentation	Less Fragmentation
Best Use Cases	Suitable for applications requiring quick allocation	Suitable when maximizing memory usage is critical

- **First-Fit** is ideal when speed is the priority and fragmentation is not a major concern.
- **Best-Fit** is beneficial for applications requiring optimal memory management at the cost of slightly slower processing.

CODE FOR MY PROGRAM (HTML, CSS, JAVASCRIPT)

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Memory Allocation Simulation</title>
 <style>
  body {
   font-family: 'Arial', sans-serif;
   background-color: #f4f4f9;
   color: #333;
   margin: 0;
   padding: 20px;
  }
  .container {
   max-width: 800px;
   margin: auto;
   padding: 20px;
   background-color: #ffffff;
   border-radius: 8px;
   box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);
  }
  h1 {
   text-align: center;
   color: #4a4a4a;
  }
  label {
   font-weight: bold;
```

```css
}
select, input[type="number"], button {
  padding: 10px;
  margin: 5px 0;
  border: 1px solid #ccc;
  border-radius: 4px;
  width: calc(100% - 22px);
  box-sizing: border-box;
}
button {
  background-color: #007bff;
  color: white;
  cursor: pointer;
  transition: background-color 0.3s;
}
button:hover {
  background-color: #0056b3;
}
#memory-container {
  margin-top: 20px;
  border: 1px solid #ccc;
  padding: 10px;
  border-radius: 4px;
  background-color: #f9f9f9;
}
.memory-block {
  margin: 5px 0;
  padding: 10px;
  border-radius: 4px;
```

```
      transition: transform 0.2s;
    }
    .free {
      background-color: #d4edda;
      border: 1px solid #c3e6cb;
    }
    .allocated {
      background-color: #f8d7da;
      border: 1px solid #f5c6cb;
    }
    .memory-block:hover {
      transform: scale(1.02);
    }
    #results {
      margin-top: 20px;
      padding: 10px;
      border: 1px solid #ccc;
      border-radius: 4px;
      background-color: #f9f9f9;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1>Memory Allocation Simulation</h1>

    <label for="algo">Select Allocation Algorithm:</label>
    <select id="algo">
      <option value="first-fit">First Fit</option>
```

```
  <option value="best-fit">Best Fit</option>
 </select>

 <label for="memSize">Total Memory Size:</label>
 <input type="number" id="memSize" value="100" min="1">
 <button onclick="initMemory()">Initialize Memory</button>

 <label for="allocSize">Allocation Size:</label>
 <input type="number" id="allocSize" value="10" min="1">
 <button onclick="allocateMemory()">Allocate Memory</button>

 <label for="deallocIndex">Deallocate Block Index:</label>
 <input type="number" id="deallocIndex" value="0" min="0">
 <button onclick="deallocateMemory()">Deallocate Memory</button>

 <div id="memory-container"></div>
 <div id="results"></div>
</div>

<script>
 let memory = [];
 let totalMemorySize = 100;
 let firstFitStats = { attempts: 0, successful: 0 };
 let bestFitStats = { attempts: 0, successful: 0 };

 function initMemory() {
  totalMemorySize = parseInt(document.getElementById("memSize").value, 10);
  memory = [{
   start: 0,
```

```
      size: totalMemorySize,
      allocated: false
    }];
    renderMemory();
    resetStats();
  }

  function resetStats() {
    firstFitStats = { attempts: 0, successful: 0 };
    bestFitStats = { attempts: 0, successful: 0 };
    updateResults();
  }

  function renderMemory() {
    const container = document.getElementById("memory-container");
    container.innerHTML = "";
    memory.forEach((block, index) => {
      const div = document.createElement("div");
      div.className = "memory-block " + (block.allocated ? "allocated" : "free");
      div.textContent = "Block " + index + ": " + (block.allocated ? "Allocated" : "Free") + " | Size: " +
block.size + " | Start: " + block.start;
      container.appendChild(div);
    });
  }

  function allocateMemory() {
    const size = parseInt(document.getElementById("allocSize").value, 10);
    const algo = document.getElementById("algo").value;
    let chosenIndex = -1;
```

```
let chosenBlock = null;

if (algo === "first-fit") {
  firstFitStats.attempts++;
  for (let i = 0; i < memory.length; i++) {
    if (!memory[i].allocated && memory[i].size >= size) {
      chosenIndex = i;
      chosenBlock = memory[i];
      break;
    }
  }
} else if (algo === "best-fit") {
  bestFitStats.attempts++;
  let bestIndex = -1;
  let bestSize = Infinity;
  for (let i = 0; i < memory.length; i++) {
    if (!memory[i].allocated && memory[i].size >= size && memory[i].size < bestSize) {
      bestSize = memory[i].size;
      bestIndex = i;
    }
  }
  if (bestIndex !== -1) {
    chosenIndex = bestIndex;
    chosenBlock = memory[bestIndex];
  }
}

if (chosenBlock === null) {
  alert("Not enough memory available to allocate " + size + " units.");
```

```
  return;
}

const allocatedBlock = {
  start: chosenBlock.start,
  size: size,
  allocated: true
};

const remainingSize = chosenBlock.size - size;
if (remainingSize > 0) {
  const freeBlock = {
    start: chosenBlock.start + size,
    size: remainingSize,
    allocated: false
  };
  memory.splice(chosenIndex, 1, allocatedBlock, freeBlock);
} else {
  memory.splice(chosenIndex, 1, allocatedBlock);
}

if (algo === "first-fit") {
  firstFitStats.successful++;
} else {
  bestFitStats.successful++;
}

renderMemory();
updateResults();
```

```
}

function deallocateMemory() {
  const index = parseInt(document.getElementById("deallocIndex").value, 10);
  if (index < 0 || index >= memory.length || memory[index].allocated === false) {
    alert("Invalid index or block is already free.");
    return;
  }

  // Mark the block as free
  memory[index].allocated = false;

  // Merge adjacent free blocks
  mergeFreeBlocks();

  // Re-render the memory blocks
  renderMemory();
  updateResults();
}

function mergeFreeBlocks() {
  for (let i = 0; i < memory.length - 1; i++) {
    if (!memory[i].allocated && !memory[i + 1].allocated) {
      memory[i].size += memory[i + 1].size;
      memory.splice(i + 1, 1);
      i--;
    }
  }
}
```

```
   function updateResults() {
    const resultsDiv = document.getElementById("results");
    const totalAllocated = memory.reduce((acc, block) => acc + (block.allocated ? block.size : 0), 0);
    const utilization = ((totalAllocated / totalMemorySize) * 100).toFixed(2);

    resultsDiv.innerHTML = `
     <h3>Results</h3>
     <p><strong>First Fit:</strong> Attempts: ${firstFitStats.attempts}, Successful:
${firstFitStats.successful}</ p>
     <p><strong>Best Fit:</strong> Attempts: ${bestFitStats.attempts}, Successful:
${bestFitStats.successful}</p>
     <p><strong>Memory Utilization:</strong> ${utilization}%</p>
    `;
   }

   window.onload = initMemory;
  </script>
</body>
</html>
```