

LAB1 Report

LAB1 Report

2.0 思考题

2A

2B

3. 调试和测试

3A

3B

3C

3D

4. 优化和比较

4A

4B

4C

5. AI工具使用

5A

5C

HTML 代码

2.0 思考题

2A

$$10^{-8} \times 10n^2 < 1/2 \implies n \leq 2236.$$

2B

$$10^{-8} \times 20n \log_2 n < 1/2 \implies n \leq 145746.$$

3. 调试和测试

3A

通过阅读代码, 我找到了:

- 数组可能越界, 因为 `maxn=2000`, 而 `i,j` 也可以达到2000.
- `sum` 计算可能溢出, 因为 `int` 最大值约为 21 亿, 而最大 `sum` 可达 40亿.
- 更新 `sum` 时访问的 `rowsum` 位置错误, `column` 应该减一.

通过打印 `sum` 输出, 我找到了:

- `sum` 在每次询问后未置零.

DeepSeek 找到了除`sum`计算溢出以外的所有问题, 调试能力较强.

3B

获取当前时间戳作为随机数种子.

3C

(我理解指令含义, 不需要查询文档)

```
system("rand_input.exe > rand.in");
```

执行 `rand_input.exe` 并把输出储存在 `rand.in` 中.

```
"check_input.exe < rand.in"
```

将 `rand.in` 作为输入执行 `check_input.exe`.

```
system("solution_1.exe < rand.in > 1.out");  
system("solution_2.exe < rand.in > 2.out");
```

用 `rand.in` 作为输入分别执行 `solution_1.exe` 与 `solution_2.exe`, 结果分别存在 `1.out` 与 `2.out` 中.

```
system("fc 1.out 2.out >nul")
```

检查 `1.out` 与 `2.out` 是否完全一致.

3D

最大可能出现 40 亿, 可以使用 unsigned int 或 long long 存储.

4. 优化和比较

4A

预计算以 (i, j) 为右下顶点的整个左上方矩阵的和, 查询时只需要使用至多 4 个数相加减.

4B

n	m	q	solution_1/ms	solution_2/ms	solution_3/ms
50	50	200	0	0	0
500	500	200	23	22	21
500	500	2000	46	23	21
500	500	20000	298	34	29
1999	1999	20000	7765	788	360

4C

n,m,q 都很大时, solution_3 表现出明显优势.

5. AI工具使用

5A

由于完整的 html 代码太长, 我把它放在文档末尾. 演示图片如下

矩阵子矩阵和计算算法可视化

使用前缀和矩阵优化子矩阵和查询

算法原理

该算法通过预处理构建一个前缀和矩阵，使得任何子矩阵的和都可以在O(1)时间内计算出来。

算法步骤

1 构建前缀和矩阵

对于矩阵中的每个元素(i, j)，计算从(0, 0)到(i, j)的子矩阵和：
 $prefix[i][j] = matrix[i][j] + prefix[i-1][j] + prefix[i][j-1] - prefix[i-1][j-1]$

2 处理查询

对于查询(x, y, a, b)，其中(x, y)是起始坐标，a是行数，b是列数：
 $sum = prefix[x+a-1][y+b-1] - prefix[x-1][y+b-1] - prefix[x+a-1][y-1] + prefix[x-1][y-1]$

图例说明 ☐ 原始矩阵元素

☐ 前缀和矩阵元素 ☐ 当前查询区域

☐ 结果计算区域

可视化演示

起始行 (x): 起始列 (y): 行数 (a): 列数 (b):

计算子矩阵和

原始矩阵

	1	2	3	4
1	2	5	3	6
2	8	4	9	1
3	7	2	5	8
4	3	6	1	4

前缀和矩阵

	1	2	3	4
1	2	7	10	16
2	10	19	31	38
3	17	28	45	60
4	20	37	55	74

查询: 从(2, 2)开始，2行 2列的子矩阵

子矩阵和: 20

计算过程:

$prefix[3][3] = 45$
 $- prefix[3][1] = 17$
 $- prefix[1][3] = 10$
 $+ prefix[1][1] = 2$
 $= 20$

```
// 构建前缀和矩阵
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        if (i > 0) temp += prefix[i-1][j];
        if (j > 0) temp += prefix[i][j-1];
        if (i > 0 && j > 0) temp -= prefix[i-1][j-1];
        prefix[i][j] = temp;
    }
}
```

```
// 计算子矩阵和
long long result = prefix[x+a-1][y+b-1];
if (y > 0) result -= prefix[x+a-1][y-1];
if (x > 0) result -= prefix[x-1][y+b-1];
if (x > 0 && y > 0) result += prefix[x-1][y-1];
```

5C

大模型的回答:

支持二维矩阵点更新和范围查询的数据结构

以下是一些常用的数据结构，用于高效支持点更新（修改某个数字）和矩形区域和查询。假设矩阵大小为 $N \times M$ ，操作次数较多。

1. 朴素二维数组

- 描述：直接使用二维数组存储矩阵元素。
- 更新时间复杂度： $O(1)$ ，直接通过索引修改元素。
- 查询时间复杂度： $O(k)$ ，其中 k 是查询矩形的大小，最坏情况下为 $O(N \times M)$ 。
- 优缺点：实现简单，但查询效率低，适用于更新极频繁、查询极少的场景。

2. 前缀和数组

- 描述：预计算一个二维前缀和数组，其中每个元素存储从左上角到该位置的累加和。查询时通过差分计算矩形和。
- 更新时间复杂度： $O(N \times M)$ ，因为更新一个元素需要重新计算整个前缀和数组。
- 查询时间复杂度： $O(1)$ 。
- 优缺点：查询极快，但更新成本高，适用于静态数据或更新极少的场景。

3. 二维线段树

- 描述：将线段树扩展到二维，每个节点代表一个矩形子区域。支持点更新和范围查询。
- 更新时间复杂度： $O(\log N \times \log M)$ 。
- 查询时间复杂度： $O(\log N \times \log M)$ 。
- 优缺点：平衡了更新和查询性能，但实现较复杂，常数因子较大。

4. 二维树状数组（Fenwick Tree）

- 描述：一维树状数组的二维扩展，通过维护多个低维树状数组来支持点更新和前缀和查询，矩形和可通过二维前缀和差分计算。
- 更新时间复杂度： $O(\log N \times \log M)$ 。
- 查询时间复杂度： $O(\log N \times \log M)$ 。
- 优缺点：与二维线段树类似，但实现更简单，常数因子更小，是常见选择。

5. 其他数据结构

- 四叉树（Quad Tree）：将区域递归四分，适用于稀疏矩阵，但和查询效率可能不如树状数组，平均复杂度不确定。

- 稀疏矩阵表示：如字典存储非零元素，但仅适用于矩阵非常稀疏的情况，查询效率取决于数据分布。

总结

- 如果操作以查询为主且更新很少，前缀和数组是最优选择。
- 如果更新和查询都频繁，二维树状数组或二维线段树是高效选择，时间复杂度均为 $O(\log N \times \log M)$ 。
- 选择时还需考虑实现难度和常数因子，二维树状数组通常更实用。

我的疑问：

二维线段树, 二维树状数组等具体如何实现？

HTML 代码

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>矩阵子矩阵和计算算法可视化</title>
  <style>
    * {
      box-sizing: border-box;
      margin: 0;
      padding: 0;
      font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    }

    body {
      background-color: #f5f7fa;
      color: #333;
      line-height: 1.6;
      padding: 20px;
      max-width: 1200px;
      margin: 0 auto;
    }

    header {
      text-align: center;
      margin-bottom: 30px;
      padding: 20px;
      background: linear-gradient(135deg, #6a11cb 0%, #2575fc 100%);
```

```
    color: white;
    border-radius: 10px;
    box-shadow: 0 4px 15px rgba(0, 0, 0, 0.1);
}

h1 {
    font-size: 2.5rem;
    margin-bottom: 10px;
}

h2 {
    font-size: 1.8rem;
    margin: 25px 0 15px;
    color: #2c3e50;
    border-bottom: 2px solid #3498db;
    padding-bottom: 8px;
}

h3 {
    font-size: 1.4rem;
    margin: 20px 0 10px;
    color: #2980b9;
}

.container {
    display: flex;
    flex-wrap: wrap;
    gap: 30px;
    margin-bottom: 30px;
}

.explanation {
    flex: 1;
    min-width: 300px;
    background: white;
    padding: 25px;
    border-radius: 10px;
    box-shadow: 0 4px 10px rgba(0, 0, 0, 0.08);
}

.visualization {
    flex: 2;
    min-width: 500px;
    background: white;
```

```
padding: 25px;
border-radius: 10px;
box-shadow: 0 4px 10px rgba(0, 0, 0, 0.08);
}

.matrix-container {
margin: 20px 0;
overflow-x: auto;
}

.matrix {
display: inline-block;
border-collapse: collapse;
margin: 15px 0;
}

.matrix-cell {
width: 60px;
height: 60px;
border: 2px solid #3498db;
text-align: center;
font-weight: bold;
font-size: 1.2rem;
background-color: #e8f4fc;
transition: all 0.3s ease;
}

.matrix-header {
background-color: #2c3e50;
color: white;
font-weight: bold;
}

.prefix-cell {
background-color: #d4edda;
}

.query-cell {
background-color: #fff3cd;
animation: pulse 1.5s infinite;
}

.result-cell {
background-color: #f8d7da;
```



```

}

.controls {
  display: flex;
  flex-wrap: wrap;
  gap: 15px;
  margin: 20px 0;
  align-items: center;
}

button {
  padding: 12px 20px;
  background: #3498db;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  font-weight: bold;
  transition: background 0.3s;
}

button:hover {
  background: #2980b9;
}

input {
  padding: 10px;
  border: 2px solid #ddd;
  border-radius: 5px;
  width: 80px;
  text-align: center;
}

.query-result {
  margin-top: 20px;
  padding: 15px;
  background-color: #e8f4fc;
  border-left: 5px solid #3498db;
  border-radius: 5px;
}

.code-snippet {
  background-color: #2d2d2d;
  color: #f8f8f2;

```

```

padding: 15px;
border-radius: 5px;
overflow-x: auto;
margin: 20px 0;
font-family: 'Consolas', 'Monaco', monospace;
}

.algorithm-steps {
margin: 20px 0;
}

.step {
margin: 15px 0;
padding: 15px;
background-color: #f8f9fa;
border-radius: 5px;
border-left: 5px solid #6c757d;
}

.step-number {
display: inline-block;
width: 30px;
height: 30px;
background: #6c757d;
color: white;
text-align: center;
line-height: 30px;
border-radius: 50%;
margin-right: 10px;
}

@keyframes pulse {
0% { transform: scale(1); }
50% { transform: scale(1.05); }
100% { transform: scale(1); }
}

.legend {
display: flex;
flex-wrap: wrap;
gap: 15px;
margin: 20px 0;
}

```

```

    .legend-item {
        display: flex;
        align-items: center;
        margin-right: 20px;
    }

    .legend-color {
        width: 20px;
        height: 20px;
        margin-right: 8px;
        border: 1px solid #999;
    }

    footer {
        text-align: center;
        margin-top: 40px;
        padding: 20px;
        color: #7f8c8d;
        font-size: 0.9rem;
    }

    @media (max-width: 768px) {
        .container {
            flex-direction: column;
        }

        .visualization, .explanation {
            min-width: 100%;
        }
    }
</style>
</head>
<body>
    <header>
        <h1>矩阵子矩阵和计算算法可视化</h1>
        <p>使用前缀和矩阵优化子矩阵和查询</p>
    </header>

    <div class="container">
        <div class="explanation">
            <h2>算法原理</h2>
            <p>该算法通过预处理构建一个前缀和矩阵，使得任何子矩阵的和都可以在 $O(1)$ 时间内计算出来。</p>

```

```

<div class="algorithm-steps">
  <h3>算法步骤</h3>
  <div class="step">
    <span class="step-number">1</span>
    <strong>构建前缀和矩阵</strong>
    <p>对于矩阵中的每个元素(i, j)，计算从(0, 0)到(i, j)的子矩阵和：
  </p>
    <p>prefix[i][j] = matrix[i][j] + prefix[i-1][j] + prefix[i][j-1] -
prefix[i-1][j-1]</p>
  </div>

  <div class="step">
    <span class="step-number">2</span>
    <strong>处理查询</strong>
    <p>对于查询(x, y, a, b)，其中(x, y)是起始坐标，a是行数，b是列数：
  </p>
    <p>sum = prefix[x+a-1][y+b-1] - prefix[x-1][y+b-1] - prefix[x+a-1][y-
1] + prefix[x-1][y-1]</p>
  </div>
</div>

<div class="legend">
  <h3>图例说明</h3>
  <div class="legend-item">
    <div class="legend-color" style="background-color: #e8f4fc;"></div>
    <span>原始矩阵元素</span>
  </div>
  <div class="legend-item">
    <div class="legend-color" style="background-color: #d4edda;"></div>
    <span>前缀和矩阵元素</span>
  </div>
  <div class="legend-item">
    <div class="legend-color" style="background-color: #fff3cd;"></div>
    <span>当前查询区域</span>
  </div>
  <div class="legend-item">
    <div class="legend-color" style="background-color: #f8d7da;"></div>
    <span>结果计算区域</span>
  </div>
</div>

<div class="visualization">
  <h2>可视化演示</h2>

```

```

<div class="controls">
  <div>
    <label for="queryX">起始行 (x): </label>
    <input type="number" id="queryX" min="1" max="4" value="1">
  </div>
  <div>
    <label for="queryY">起始列 (y): </label>
    <input type="number" id="queryY" min="1" max="4" value="1">
  </div>
  <div>
    <label for="queryA">行数 (a): </label>
    <input type="number" id="queryA" min="1" max="4" value="2">
  </div>
  <div>
    <label for="queryB">列数 (b): </label>
    <input type="number" id="queryB" min="1" max="4" value="2">
  </div>
  <button id="calculateBtn">计算子矩阵和</button>
</div>

<h3>原始矩阵</h3>
<div class="matrix-container">
  <table class="matrix" id="originalMatrix">
    <!-- JavaScript会填充这里 -->
  </table>
</div>

<h3>前缀和矩阵</h3>
<div class="matrix-container">
  <table class="matrix" id="prefixMatrix">
    <!-- JavaScript会填充这里 -->
  </table>
</div>

<div class="query-result" id="queryResult">
  <p>请输入查询参数并点击"计算子矩阵和"按钮查看结果</p>
</div>

<div class="code-snippet">
  <pre>
// 构建前缀和矩阵
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {

```

```

        if (i > 0) temp += prefix[i-1][j];
        if (j > 0) temp += prefix[i][j-1];
        if (i > 0 && j > 0) temp -= prefix[i-1][j-1];
        prefix[i][j] = temp;
    }
}

// 计算子矩阵和
long long result = prefix[x+a-1][y+b-1];
if (y > 0) result -= prefix[x+a-1][y-1];
if (x > 0) result -= prefix[x-1][y+b-1];
if (x > 0 && y > 0) result += prefix[x-1][y-1];

```

</pre>

</div>

</div>

</div>

<footer>

<p>© 2023 矩阵算法可视化 | 基于C++程序solution_3.cpp实现</p>

</footer>

<script>

```

document.addEventListener('DOMContentLoaded', function() {
    // 示例矩阵数据
    const originalMatrix = [
        [2, 5, 3, 6],
        [8, 4, 9, 1],
        [7, 2, 5, 8],
        [3, 6, 1, 4]
    ];

    // 计算前缀和矩阵
    const prefixMatrix = [];
    for (let i = 0; i < originalMatrix.length; i++) {
        prefixMatrix[i] = [];
        for (let j = 0; j < originalMatrix[i].length; j++) {
            let temp = originalMatrix[i][j];
            if (i > 0) temp += prefixMatrix[i-1][j];
            if (j > 0) temp += prefixMatrix[i][j-1];
            if (i > 0 && j > 0) temp -= prefixMatrix[i-1][j-1];
            prefixMatrix[i][j] = temp;
        }
    }
}

```

```

// 渲染原始矩阵
const originalTable = document.getElementById('originalMatrix');
renderMatrix(originalTable, originalMatrix, false);

// 渲染前缀和矩阵
const prefixTable = document.getElementById('prefixMatrix');
renderMatrix(prefixTable, prefixMatrix, true);

// 计算按钮点击事件
document.getElementById('calculateBtn').addEventListener('click', function()
{
    const x = parseInt(document.getElementById('queryX').value) - 1;
    const y = parseInt(document.getElementById('queryY').value) - 1;
    const a = parseInt(document.getElementById('queryA').value);
    const b = parseInt(document.getElementById('queryB').value);

    // 验证输入
    if (x < 0 || y < 0 || x + a > originalMatrix.length || y + b >
originalMatrix[0].length) {
        document.getElementById('queryResult').innerHTML =
            '<p style="color: red;">错误：查询参数超出矩阵范围</p>';
        return;
    }

    // 计算子矩阵和
    let result = prefixMatrix[x + a - 1][y + b - 1];
    if (y > 0) result -= prefixMatrix[x + a - 1][y - 1];
    if (x > 0) result -= prefixMatrix[x - 1][y + b - 1];
    if (x > 0 && y > 0) result += prefixMatrix[x - 1][y - 1];

    // 显示结果
    document.getElementById('queryResult').innerHTML = `
        <p><strong>查询:</strong> 从(${x+1}, ${y+1})开始, ${a}行 ${b}列的子
矩阵</p>

        <p><strong>子矩阵和:</strong> ${result}</p>
        <p><strong>计算过程:</strong><br>
        prefix[${x+a-1+1}][${y+b-1+1}] = ${prefixMatrix[x+a-1][y+b-1]}<br>
        ${y > 0 ? ` - prefix[${x+a-1+1}][${y}] = ${prefixMatrix[x+a-1][y-1]}
<br>` : ''}

        ${x > 0 ? ` - prefix[${x}][${y+b-1+1}] = ${prefixMatrix[x-1][y+b-1]}
<br>` : ''}

        ${x > 0 && y > 0 ? ` + prefix[${x}][${y}] = ${prefixMatrix[x-1][y-
1]}<br>` : ''}

        = ${result}
    `;
}

```

```

        </p>
    `;

    // 高亮显示相关区域
    highlightMatrices(x, y, a, b);
});

// 渲染矩阵函数
function renderMatrix(table, matrix, isPrefix) {
    table.innerHTML = '';

    // 添加表头行
    let headerRow = document.createElement('tr');
    let cornerCell = document.createElement('td');
    cornerCell.className = 'matrix-cell matrix-header';
    cornerCell.textContent = '';
    headerRow.appendChild(cornerCell);

    for (let j = 0; j < matrix[0].length; j++) {
        let headerCell = document.createElement('td');
        headerCell.className = 'matrix-cell matrix-header';
        headerCell.textContent = j + 1;
        headerRow.appendChild(headerCell);
    }
    table.appendChild(headerRow);

    // 添加数据行
    for (let i = 0; i < matrix.length; i++) {
        let row = document.createElement('tr');

        // 行号单元格
        let rowHeader = document.createElement('td');
        rowHeader.className = 'matrix-cell matrix-header';
        rowHeader.textContent = i + 1;
        row.appendChild(rowHeader);

        for (let j = 0; j < matrix[i].length; j++) {
            let cell = document.createElement('td');
            cell.className = 'matrix-cell';
            if (isPrefix) cell.classList.add('prefix-cell');
            cell.textContent = matrix[i][j];
            cell.dataset.row = i;
            cell.dataset.col = j;
            row.appendChild(cell);
        }
    }
}

```



```

    }
    table.appendChild(row);
  }
}

// 高亮显示矩阵区域
function highlightMatrices(x, y, a, b) {
  // 重置所有单元格样式
  const allCells = document.querySelectorAll('.matrix-cell');
  allCells.forEach(cell => {
    cell.classList.remove('query-cell', 'result-cell');
    if (!cell.classList.contains('matrix-header') &&
!cell.classList.contains('prefix-cell')) {
      cell.style.backgroundColor = '#e8f4fc';
    } else if (cell.classList.contains('prefix-cell')) {
      cell.style.backgroundColor = '#d4edda';
    }
  });

  // 高亮原始矩阵中的查询区域
  for (let i = x; i < x + a; i++) {
    for (let j = y; j < y + b; j++) {
      const cell = document.querySelector(`#originalMatrix .matrix-
cell[data-row="${i}"][data-col="${j}"]`);
      if (cell) cell.classList.add('query-cell');
    }
  }

  // 高亮前缀和矩阵中的计算区域
  const resultCells = [
    {row: x+a-1, col: y+b-1, type: 'positive'},
    {row: x+a-1, col: y-1, type: 'negative'},
    {row: x-1, col: y+b-1, type: 'negative'},
    {row: x-1, col: y-1, type: 'positive'}
  ];

  resultCells.forEach(cellInfo => {
    if (cellInfo.row >= 0 && cellInfo.col >= 0 &&
      cellInfo.row < prefixMatrix.length && cellInfo.col <
prefixMatrix[0].length) {
      const cell = document.querySelector(`#prefixMatrix .matrix-
cell[data-row="${cellInfo.row}"][data-col="${cellInfo.col}"]`);
      if (cell) {
        cell.classList.add('result-cell');
      }
    }
  });
}

```

```
        cell.style.backgroundColor = cellInfo.type === 'positive' ?  
        '#d4edda' : '#f8d7da';  
    }  
    }  
});  
}  
});  
</script>  
</body>  
</html>
```