

专业课程实验报告

课程名称： 算法分析与设计

开课学期： 2020 至 2021 学年 第 一 学期

专业： 软件工程 年级班级： 1班

学生姓名： 宋行健 学号： 222018321062006

实验教师： 曹严元

计算机与信息科学学院 软件学院

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验项目名称 | | 递归与分治策略—分治策略实现棋盘覆盖问题、斯特拉森矩阵乘法 | | | |
| 实验时间 | | 2020年9月29日 | 实验类型 | | □验证性 设计性 □综合性 |
| 一、实验目的   1. 了解并掌握递归的概念，递归算法的基本思想； 2. 掌握分治法的基本思想方法； 3. 了解适用于用分治法求解的问题类型，并能用递归或非递归的方式设计相应的分治法算法； 4. 掌握分治法复杂性分析方法，比较同一个问题的递归算法与循环迭代算法的效率。   二、实验要求   1. 预习实验指导书及教材的有关内容，掌握分治法的基本思想； 2. 严格按照实验内容进行实验，培养良好的算法设计和编程的习惯； 3. 认真听讲，服从安排，独立思考并完成实验。 | | | | | |
| 三、实验内容与设计（主要内容，操作步骤、算法描述或程序代码） 用分治策略实现棋盘覆盖问题  1. 数据结构：   在本次实验中，选用二维数组的数据结构进行问题的表示。二维数组中的每一个元素就代表着棋盘中的一个格子。因此这种数据结构的表达方式十分直观清晰，但是也有一些不足，例如二维数组所占的内存空间较大，浪费空间资源。   1. 棋盘覆盖问题的伪码算法：          1. 编制C++或JAVA等高级语言程序实现伪码算法： 2. #include <iostream> 4. **int** tile = 1;        // 骨牌序号 5. **int** board[128][128]; // 二维数组模拟棋盘 7. // (tr,tc)表示棋盘的左上角坐标(即确定棋盘位置), (dr,dc)表示特殊方块的位置, size=2^k确定棋盘大小 8. **void** chessBoard(**int** tr, **int** tc, **int** dr, **int** dc, **int** size) 9. { 10. // 递归出口 11. **if** (size == 1) 12. **return**; 14. **int** s = size / 2; //分割棋盘 15. **int** t = tile++;   //t记录本层骨牌序号 16. // 判断特殊方格在不在左上棋盘 17. **if** (dr < tr + s && dc < tc + s) 18. { 19. chessBoard(tr, tc, dr, dc, s); //特殊方格在左上棋盘的递归处理方法 20. } 21. **else** 22. { 23. board[tr + s - 1][tc + s - 1] = t;             // 用t号的L型骨牌覆盖右下角 24. chessBoard(tr, tc, tr + s - 1, tc + s - 1, s); // 递归覆盖其余方格 25. } 27. // 判断特殊方格在不在右上棋盘 28. **if** (dr < tr + s && dc >= tc + s) 29. { 30. chessBoard(tr, tc + s, dr, dc, s); 31. } 32. **else** 33. { 34. board[tr + s - 1][tc + s] = t; 35. chessBoard(tr, tc + s, tr + s - 1, tc + s, s); 36. } 38. // 判断特殊方格在不在左下棋盘 39. **if** (dr >= tr + s && dc < tc + s) 40. { 41. chessBoard(tr + s, tc, dr, dc, s); 42. } 43. **else** 44. { 45. board[tr + s][tc + s - 1] = t; 46. chessBoard(tr + s, tc, tr + s, tc + s - 1, s); 47. } 49. // 判断特殊方格在不在右下棋盘 50. **if** (dr >= tr + s && dc >= tc + s) 51. { 52. chessBoard(tr + s, tc + s, dr, dc, s); 53. } 54. **else** 55. { 56. board[tr + s][tc + s] = t; 57. chessBoard(tr + s, tc + s, tr + s, tc + s, s); 58. } 59. } 61. **int** main() 62. { 63. **int** boardSize = 4;                 // 棋盘边长 64. chessBoard(0, 0, 3, 3, boardSize); // (0, 0)为顶点，大小为boardSize的棋盘； 特殊方块位于(3, 3) 66. // 打印棋盘 67. **int** i, j; 68. printf("\n\n\n"); 69. **for** (i = 0; i < boardSize; i++) 70. { 71. **for** (j = 0; j < boardSize; j++) 72. { 73. printf("%d\t", board[i][j]); 74. } 75. printf("\n\n\n"); 76. } 77. **return** 0; 78. } 79. 分析时间复杂度   设是该算法棋盘覆盖一个棋盘所需时间，从算法的划分策略可知，满足如下递推关系：  进行求解：  所以。 | | | | | |
| 用分治策略实现斯特拉森矩阵乘法  1. 数据结构：   在本次实验中，选用二维数组的数据结构进行问题的表示。二维数组自然表示矩阵。在函数调用时，将二维数组以指针的形式进行传递，对于矩阵的运算直接通过地址对元素进行取值和运算。  特拉森矩阵乘法在数据结构上与棋盘覆盖类似，都是在二维的平面上进行运算，两者可以互为参考和补充。   1. 斯特拉森矩阵乘法的伪码算法：          1. 编制C++或JAVA等高级语言程序实现伪码算法： 2. #include <iostream> 3. #include <string.h> 5. **using** **namespace** std; 7. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 8. \* 函数描述： 析取矩阵元素 9. \* 函数参数： pM——矩阵指针 10. nCol——使用矩阵列大小（指针取值使用） 11. i——索引横坐标 12. j——索引纵坐标 13. \* 函数返回： 矩阵对应位置的元素值 14. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 15. **int**& GetArrayVal(**int**\* pM, **int** nCol, **int** i, **int** j) 16. { 17. **return** \*(pM + i \* nCol + j); 18. }  21. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 22. \* 函数描述： 创建矩阵 23. 矩阵元素为 [0,5] 范围内的随机整数 24. \* 函数参数： pM——矩阵指针 25. nRow——创建矩阵的行规模 26. nCol——创建矩阵的列规模 27. \* 函数返回： void 28. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 29. **void** CreateMatrix(**int**\*\* pM, **int** nRow, **int** nCol) 30. { 31. \*pM = **new** **int**[nRow \* nCol]; 32. memset(\*pM, 0, **sizeof**(**int**\*) \* nRow \* nCol); 34. **for** (**int** i = 0; i < nRow; ++i) 35. { 36. **for** (**int** j = 0; j < nCol; ++j) 37. { 38. GetArrayVal(\*pM, nCol, i, j) = rand() % 6; 39. } 40. } 41. }  44. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 45. \* 函数描述： 销毁矩阵(内存管理) 46. \* 函数参数： pM——矩阵指针 47. \* 函数返回：void 48. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 49. **void** DeleteMatrix(**int**\*\* pM) 50. { 51. **if** (NULL != \*pM) 52. { 53. **delete**\* pM; 54. \*pM = NULL; 55. } 56. }  59. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 60. \* 函数描述： 矩阵加减法（n阶方阵） 61. \* 函数参数： pM1——矩阵1 62. nLeftIndex1，nTopIndex1——矩阵1左上角索引点（相对于源矩阵pMl） 63. nTotalCol1——矩阵1实际使用的列数 64. pM2——矩阵2 65. nLeftIndex2, nTopIndex2——矩阵2左上角索引点（相对于源矩阵pM2） 66. nTotalCol2——矩阵2实际使用的列数 67. nCount——方阵阶数n 68. pResult——运算结果矩阵 69. bAdd——加减标记 70. \* 函数返回：void 71. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 72. **void** MatrixAddOrSub(**int**\* pM1, **int** nLeftIndex1, **int** nTopIndex1, **int** nTotalCol1, 73. **int**\* pM2, **int** nLeftIndex2, **int** nTopIndex2, **int** nTotalCol2, 74. **int** nCount, **int**\*\* pResult, **bool** bAdd) 75. { 76. \*pResult = **new** **int**[nCount \* nCount]; 77. **for** (**int** i = 0; i < nCount; ++i) 78. { 79. **for** (**int** j = 0; j < nCount; ++j) 80. { 81. **if** (bAdd)   // 加法 82. { 83. GetArrayVal(\*pResult, nCount, i, j) = GetArrayVal(pM1, nTotalCol1, nLeftIndex1 + i, nTopIndex1 + j) 84. + GetArrayVal(pM2, nTotalCol2, nLeftIndex2 + i, nTopIndex2 + j); 85. } 86. **else**        // 减法 87. { 88. GetArrayVal(\*pResult, nCount, i, j) = GetArrayVal(pM1, nTotalCol1, nLeftIndex1 + i, nTopIndex1 + j) 89. - GetArrayVal(pM2, nTotalCol2, nLeftIndex2 + i, nTopIndex2 + j); 90. } 92. } 93. } 94. }  97. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 98. \* 函数描述： 矩阵乘法（n阶方阵） 99. \* 函数参数： pM1——矩阵1 100. nLeftIndex1，nTopIndex1——矩阵1左上角索引点（相对于源矩阵pMl） 101. nTotalCol1——矩阵1实际使用的列数 102. pM2——矩阵2 103. nLeftIndex2, nTopIndex2——矩阵2左上角索引点（相对于源矩阵pM2） 104. nTotalCol2——矩阵2实际使用的列数 105. nCount——方阵阶数n 106. pResult——运算结果矩阵 107. \* 函数返回：void 108. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 109. **void** MatrixMulti(**int**\* pM1, **int** nLeftIndex1, **int** nTopIndex1, **int** nTotalCol1, 110. **int**\* pM2, **int** nLeftIndex2, **int** nTopIndex2, **int** nTotalCol2, 111. **int** nCount, **int**\*\* pResult) 112. { 114. \*pResult = **new** **int**[nCount \* nCount]; 115. **for** (**int** i = 0; i < nCount; ++i) 116. { 117. **for** (**int** j = 0; j < nCount; ++j) 118. { 119. GetArrayVal(\*pResult, nCount, i, j) = 0; 120. **for** (**int** k = 0; k < nCount; ++k) 121. { 122. GetArrayVal(\*pResult, nCount, i, j) += GetArrayVal(pM1, nTotalCol1, nLeftIndex1 + i, nTopIndex1 + k) 123. \* GetArrayVal(pM2, nTotalCol2, nLeftIndex2 + k, nTopIndex2 + j); 124. } 125. } 126. } 127. }  130. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 131. \* 函数描述： 递归实现斯特拉森矩阵乘法（n阶方阵） 132. \* 函数参数： pM1——矩阵1 133. nLeftIndex1，nTopIndex1——矩阵1左上角索引点（相对于源矩阵pMl） 134. nTotalCol1——矩阵1实际使用的列数 135. pM2——矩阵2 136. nLeftIndex2, nTopIndex2——矩阵2左上角索引点（相对于源矩阵pM2） 137. nTotalCol2——矩阵2实际使用的列数 138. nCount——方阵阶数n 139. pResult——运算结果矩阵 140. \* 函数返回：void 141. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 142. **void** StrassenMatrix(**int**\* pM1, **int** nLeftIndex1, **int** nTopIndex1, **int** nTotalCol1, 143. **int**\* pM2, **int** nLeftIndex2, **int** nTopIndex2, **int** nTotalCol2, 144. **int** nCount, **int**\*\* pResult) 145. { 146. **if** (nCount == 2) // 如果当前为2阶，不能继续划分则跳出迭代 147. { 148. MatrixMulti(pM1, nLeftIndex1, nTopIndex1, nTotalCol1, 149. pM2, nLeftIndex2, nTopIndex2, nTotalCol2, nCount, pResult); 150. } 151. **else**    // 如果当前大于2阶，拆分成4个大小相等的子矩阵，分别进行迭代 152. { 153. **int**\* pResultM1 = NULL; 154. **int**\* pResultM2 = NULL; 155. **int**\* pResultM3 = NULL; 156. **int**\* pResultM4 = NULL; 157. **int**\* pResultM5 = NULL; 158. **int**\* pResultM6 = NULL; 159. **int**\* pResultM7 = NULL; 161. // M1 = A11 \* (B12 - B22) 162. **int**\* pB12\_B22 = NULL; 163. MatrixAddOrSub(pM2, nLeftIndex2, nTopIndex2 + nCount / 2, nTotalCol2, 164. pM2, nLeftIndex2 + nCount / 2, nTopIndex2 + nCount / 2, nTotalCol2, nCount / 2, &pB12\_B22, **false**); 165. StrassenMatrix(pM1, nLeftIndex1, nTopIndex1, nTotalCol1, 166. pB12\_B22, 0, 0, nCount / 2, nCount / 2, &pResultM1); 168. // M2 = (A11 + A12) \* B22; 169. **int**\* pA11\_A12 = NULL; 170. MatrixAddOrSub(pM1, nLeftIndex1, nTopIndex1, nTotalCol1, 171. pM1, nLeftIndex1, nTopIndex1 + nCount / 2, nTotalCol1, nCount / 2, &pA11\_A12, **true**); 172. StrassenMatrix(pA11\_A12, 0, 0, nCount / 2, 173. pM2, nLeftIndex2 + nCount / 2, nTopIndex2 + nCount / 2, nTotalCol2, nCount / 2, &pResultM2); 175. // M3 = (A21 + A22) \* B11; 176. **int**\* pA21\_A22 = NULL; 177. MatrixAddOrSub(pM1, nLeftIndex1 + nCount / 2, nTopIndex1, nTotalCol1, 178. pM1, nLeftIndex1 + nCount / 2, nTopIndex1 + nCount / 2, nTotalCol1, nCount / 2, &pA21\_A22, **true**); 179. StrassenMatrix(pA21\_A22, 0, 0, nCount / 2, 180. pM2, nLeftIndex2, nTopIndex2, nTotalCol2, nCount / 2, &pResultM3); 182. // M4 = A22 \* (B21 - B11) 183. **int**\* pB21\_B11 = NULL; 184. MatrixAddOrSub(pM2, nLeftIndex2 + nCount / 2, nTopIndex2, nTotalCol2, 185. pM2, nLeftIndex2, nTopIndex2, nTotalCol2, nCount / 2, &pB21\_B11, **false**); 186. StrassenMatrix(pM1, nLeftIndex1 + nCount / 2, nTopIndex1 + nCount / 2, nTotalCol1, 187. pB21\_B11, 0, 0, nCount / 2, nCount / 2, &pResultM4);  190. // M5 = (A11 + A22) \* (B11 + B22) 191. **int**\* pA11\_A22 = NULL; 192. **int**\* pB11\_B22 = NULL; 193. MatrixAddOrSub(pM1, nLeftIndex1, nTopIndex1, nTotalCol1, 194. pM1, nLeftIndex1 + nCount / 2, nTopIndex1 + nCount / 2, nTotalCol1, nCount / 2, &pA11\_A22, **true**); 195. MatrixAddOrSub(pM2, nLeftIndex2, nTopIndex2, nTotalCol2, 196. pM2, nLeftIndex2 + nCount / 2, nTopIndex2 + nCount / 2, nTotalCol2, nCount / 2, &pB11\_B22, **true**); 197. StrassenMatrix(pA11\_A22, 0, 0, nCount / 2, 198. pB11\_B22, 0, 0, nCount / 2, nCount / 2, &pResultM5); 200. // M6 = (A12 - A22) \* (B21 + B22) 201. **int**\* pA12\_A22 = NULL; 202. **int**\* pB21\_B22 = NULL; 203. MatrixAddOrSub(pM1, nLeftIndex1, nTopIndex1 + nCount / 2, nTotalCol1, 204. pM1, nLeftIndex1 + nCount / 2, nTopIndex1 + nCount / 2, nTotalCol1, nCount / 2, &pA12\_A22, **false**); 205. MatrixAddOrSub(pM2, nLeftIndex2 + nCount / 2, nTopIndex2, nTotalCol2, 206. pM2, nLeftIndex2 + nCount / 2, nTopIndex2 + nCount / 2, nTotalCol2, nCount / 2, &pB21\_B22, **true**); 207. StrassenMatrix(pA12\_A22, 0, 0, nCount / 2, 208. pB21\_B22, 0, 0, nCount / 2, nCount / 2, &pResultM6); 210. // M7 = (A11 - A21) \* (B11 + B12) 211. **int**\* pA11\_A21 = NULL; 212. **int**\* pB11\_B12 = NULL; 213. MatrixAddOrSub(pM1, nLeftIndex1, nTopIndex1, nTotalCol1, 214. pM1, nLeftIndex1 + nCount / 2, nTopIndex1, nTotalCol1, nCount / 2, &pA11\_A21, **false**); 215. MatrixAddOrSub(pM2, nLeftIndex2, nTopIndex2, nTotalCol2, 216. pM2, nLeftIndex2, nTopIndex2 + nCount / 2, nTotalCol2, nCount / 2, &pB11\_B12, **true**); 217. StrassenMatrix(pA11\_A21, 0, 0, nCount / 2, 218. pB11\_B12, 0, 0, nCount / 2, nCount / 2, &pResultM7);  221. **int**\* pResultC11 = NULL; 222. **int**\* pResultC12 = NULL; 223. **int**\* pResultC21 = NULL; 224. **int**\* pResultC22 = NULL; 225. **int**\* pResultTemp1 = NULL; 226. **int**\* pResultTemp2 = NULL; 228. // C11 = M5 + M4 - M2 + M6 229. MatrixAddOrSub(pResultM5, 0, 0, nCount / 2, 230. pResultM4, 0, 0, nCount / 2, nCount / 2, &pResultTemp1, **true**); 231. MatrixAddOrSub(pResultTemp1, 0, 0, nCount / 2, 232. pResultM2, 0, 0, nCount / 2, nCount / 2, &pResultTemp2, **false**); 233. MatrixAddOrSub(pResultTemp2, 0, 0, nCount / 2, 234. pResultM6, 0, 0, nCount / 2, nCount / 2, &pResultC11, **true**);  237. // C12 = M1 + M2 238. MatrixAddOrSub(pResultM1, 0, 0, nCount / 2, 239. pResultM2, 0, 0, nCount / 2, nCount / 2, &pResultC12, **true**); 241. // C21 = M3 + M4 242. MatrixAddOrSub(pResultM3, 0, 0, nCount / 2, 243. pResultM4, 0, 0, nCount / 2, nCount / 2, &pResultC21, **true**); 245. // C22 = M5 + M1 - M3 - M7 246. MatrixAddOrSub(pResultM5, 0, 0, nCount / 2, 247. pResultM1, 0, 0, nCount / 2, nCount / 2, &pResultTemp1, **true**); 248. MatrixAddOrSub(pResultTemp1, 0, 0, nCount / 2, 249. pResultM3, 0, 0, nCount / 2, nCount / 2, &pResultTemp2, **false**); 250. MatrixAddOrSub(pResultTemp2, 0, 0, nCount / 2, 251. pResultM7, 0, 0, nCount / 2, nCount / 2, &pResultC22, **false**);  254. // 构造结果 255. \*pResult = **new** **int**[nCount \* nCount]; 256. **for** (**int** i = 0; i < nCount / 2; ++i) 257. { 258. **for** (**int** j = 0; j < nCount / 2; ++j) 259. { 260. GetArrayVal(\*pResult, nCount, i, j) = GetArrayVal(pResultC11, nCount / 2, i, j); 261. GetArrayVal(\*pResult, nCount, i, j + nCount / 2) = GetArrayVal(pResultC12, nCount / 2, i, j); 262. GetArrayVal(\*pResult, nCount, i + nCount / 2, j) = GetArrayVal(pResultC21, nCount / 2, i, j); 263. GetArrayVal(\*pResult, nCount, i + nCount / 2, j + nCount / 2) = GetArrayVal(pResultC22, nCount / 2, i, j); 265. } 266. } 268. //释放内存 269. DeleteMatrix(&pResultM1); 270. DeleteMatrix(&pResultM2); 271. DeleteMatrix(&pResultM3); 272. DeleteMatrix(&pResultM4); 273. DeleteMatrix(&pResultM5); 274. DeleteMatrix(&pResultM6); 275. DeleteMatrix(&pResultM7); 276. DeleteMatrix(&pA11\_A12); 277. DeleteMatrix(&pA21\_A22); 278. DeleteMatrix(&pB12\_B22); 279. DeleteMatrix(&pB21\_B11); 280. DeleteMatrix(&pA11\_A22); 281. DeleteMatrix(&pB11\_B22); 282. DeleteMatrix(&pA12\_A22); 283. DeleteMatrix(&pB21\_B22); 284. DeleteMatrix(&pA11\_A21); 285. DeleteMatrix(&pB11\_B12); 286. DeleteMatrix(&pResultTemp1); 287. DeleteMatrix(&pResultTemp2); 288. DeleteMatrix(&pResultC11); 289. DeleteMatrix(&pResultC12); 290. DeleteMatrix(&pResultC21); 291. DeleteMatrix(&pResultC22); 293. } 294. } 296. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 297. \* 函数描述： 打印矩阵 298. \* 函数参数： pM——矩阵指针 299. nRow——矩阵行规模 300. nCol——矩阵列规模 301. \* 函数返回：void 302. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 303. **void** PrintMatrix(**int**\* pM, **int** nRow, **int** nCol) 304. { 305. **for** (**int** i = 0; i < nRow; ++i) 306. { 307. **for** (**int** j = 0; j < nCol; ++j) 308. { 309. cout << GetArrayVal(pM, nCol, i, j) << " "; 310. } 311. cout << endl; 312. } 313. } 315. **int** main() 316. { 317. srand(0);   // 随机数种子 319. **int**\* pM1 = NULL; 320. **int**\* pM2 = NULL; 321. **int**\* pMResult1 = NULL; 322. **int** nRow1, nCol1, nRow2, nCol2; 324. // 测试4阶方阵乘法 325. nRow1 = 4; 326. nCol1 = 4; 327. nRow2 = 4; 328. nCol2 = 4; 330. // 测试8阶方阵乘法 331. //nRow1 = 8; 332. //nCol1 = 8; 333. //nRow2 = 8; 334. //nCol2 = 8; 336. // 随机构建两个矩阵，并打印 337. CreateMatrix(&pM1, nRow1, nCol1); 338. CreateMatrix(&pM2, nRow2, nCol2); 339. cout << "\nMatrix A:" << endl; 340. PrintMatrix(pM1, nRow1, nCol1); 341. cout << "\nMatrix B:" << endl; 342. PrintMatrix(pM2, nRow2, nCol2); 344. // 使用普通方法计算矩阵乘法 345. cout << "\nGeneral matrix multiplication:" << endl; 346. MatrixMulti(pM1, 0, 0, nRow1, pM2, 0, 0, nRow1, nRow1, &pMResult1); 347. PrintMatrix(pMResult1, nRow1, nCol2); 349. // 使用斯特拉森方法计算矩阵乘法 350. cout << "\nStrassen matrix multiplication:" << endl; 351. StrassenMatrix(pM1, 0, 0, nRow1, pM2, 0, 0, nRow1, nRow1, &pMResult1); 352. PrintMatrix(pMResult1, nRow1, nCol2); 354. //内存释放 355. DeleteMatrix(&pM1); 356. DeleteMatrix(&pM2); 357. DeleteMatrix(&pMResult1); 358. cout << "-----------------------------------" << endl; 360. **return** 0; 361. } 362. 分析时间复杂度   设是斯特拉森（Strassen）算法对两个阶方阵进行运算所需的时间，Strassen算法中共使用了7次递归调用的矩阵乘法（即在生成的7个矩阵的时候），共使用了18次矩阵加减运算（在生成的时候使用了6次矩阵加法和4次矩阵减法，在生成的时候使用了6次矩阵加法和2次矩阵减法）。所以其递归表达式如下：  令进行求解：，即斯特拉森算法的时间复杂度为。而传统的矩阵乘法中有三层for循环，因此显然其时间复杂度为。由此可见，Strassen矩阵乘法的计算时间复杂性比普通算法有着较大改进。 | | | | | |
| 三、测试数据和执行结果 （在给定数据下，执行操作、算法和程序的结果，可使用数据、图表、截图等给出）   1. 棋盘覆盖问题   图 1和图 2分别展示的是棋盘规模为4和棋盘规模为8的情况。其中相同的L型骨牌使用了相同的数字来表示，起始特殊骨牌的位置在（4, 4）坐标，即图中0号位置。通过图中结果可以看出算法的运算结果是正确的。    图 1 棋盘规模为4    图 2 棋盘规模为8   1. 斯特拉森矩阵乘法   图 3和图 4分别展示的是阶数为4的矩阵乘法和阶数为8的矩阵乘法。算法首先随机生成两个阶数为的矩阵为运算做准备，并将它们打印出来，分别为MatrixA和MatrixB。之后使用普通矩阵乘法和斯特拉森矩阵乘法分别对其进行计算并打印结果。从图中看，斯特拉森矩阵乘法的运算结果与普通矩阵乘法的运算结果相同，说明算法正确。    图 3 阶数为4的矩阵乘法    图 4 阶数为8的矩阵乘法 | | | | | |
| 四、实验结果分析及总结（对实验的结果是否达到预期进行分析，总结实验的收获和存在的问题等）  通过本次实验，我对分治策略有了更深入、更系统的了解。同时，我选择了用C++语言来实现这些排序算法，对C++中指针的传递、二维数组的传递方式及其指针表达方式、内存分配等代码实现也进行了复习。  **（一）棋盘覆盖问题**  对于棋盘覆盖问题，这是二维的分治策略，将一个大棋盘均分为4小块，对每一小块判断是否存在特殊方格和L骨牌覆盖，在覆盖时如果子棋盘依然可分，则递归地进行分治求解，直到每一个子棋盘均为1为止。  在这个问题中，最困难的地方就是使用索引对棋盘进行划分，算法中统一使用左上角的方格对子棋盘进行定位，在递归的时候不仅要注意定位方格的位置改变，还要注意子棋盘规模的改变。  对棋盘覆盖问题的时间复杂度进行分析，得出覆盖一个棋盘所需时间。  **（二）斯特拉森矩阵乘法**  对于斯特拉森矩阵乘法，这同样是一个二维情景下的分治策略。其本质与棋盘覆盖类似，也是将矩阵均分为4小块进行运算。通过小矩阵块进行一系列的矩阵加减运算和乘法运算（这里进行递归求解），最后组合得出最终结果。不同之处在于，棋盘覆盖递归的时候只有一个二维数组，而斯特拉森矩阵乘法在递归传参的时候需要传递两个二维数组的信息，因此函数的参数多了一倍，代码量也更大，算法也更加繁琐，但是本质和棋盘覆盖问题是想通的。  因为算法代码量巨大，过程繁琐，加上自己对二维数组在函数中指针调用不老练，我先在网上找了一个代码进行参考，代码链接为：<https://blog.csdn.net/s634772208/article/details/46594707>。我并未直接复制原始代码，而是对原始代码进行了修改整合，并根据实验要求对其简化并修改，逐句理解并对代码进行注释。在这个过程中，我对斯特拉森矩阵乘法有了更加深刻的理解，同时也学习到了原作者对二维数组在C++语言中的传递方法和调用方法。  对斯特拉森矩阵乘法的递归次数和矩阵加减运算次数进行计数，并时间复杂度进行分析，得出对两个阶（阶）方阵进行运算所需的时间，斯特拉森矩阵乘法中共使用了7次递归调用的矩阵乘法和18次矩阵加减运算。其时间复杂度为，而传统矩阵乘法的时间复杂度为。由此可见，Strassen矩阵乘法的计算时间复杂性比传统算法有着较大改进。  本次实验的两个问题都是分治法中的经典问题。通过实验，我总结得出：分治法的基本思想是将一个规模很大的问题分解为许多规模较小的子问题，这些子问题互相独立且与原问题相同。对这些子问题递归地进行求解后合并，得到原问题的解。 | | | | | |
| 教  师  评  阅 | 实验内容和设计（A-E）： | | |  | |
| 操作过程、算法或代码（A-E）： | | |  | |
| 实验结果（A-E）： | | |  | |
| 实验分析和总结（A-E）： | | |  | |
| 实验成绩（A-E）：  反馈评语： | | | | |