

**课 程 实 验 报 告**

**课程名称： 算法设计**

**实验名称： 矩阵链乘法**

**院 系 ：计算机科学与技术**

**专业班级 ： 计卓1701**

**学 号 ： U201714487**

**姓 名 ： 蔡靖涛**

**指导教师 ： 何琨**

**2019 年 11 月 10 日**

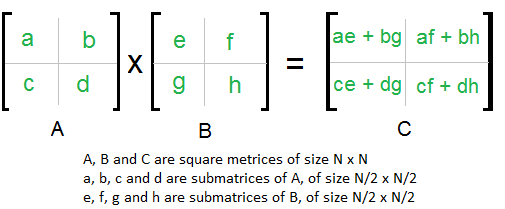
1. **实验内容**

本次实验的内容是基于分治算法思想Strassen’s Matrix Multipliation实现，我所实现的具体内容是由Kotlin语言编写使用基于分治算法思想的求乱序数组第k小元素的算法。

1. **算法描述**

我的算法可以大致分为以下几步：

* + - 1. 首先将相乘的两个n\*n矩阵A,B分为四个大小相等的(n/2)\*(n/2)矩阵a,b,c,d和e,f,g,h，如下图所示



**图2-1 矩阵划分**

* + - 1. 计算Strassen矩阵乘法的七个相乘项p1..7，其中：

p1 = a(f-h) p2 = (a+b)h

p3 = (c+d)e p4 = d(g-e)

p5 = (a+d)(e+h) p6 = (b-d)(g+h)

p7 = (a-c)(e+f)

* + - 1. 计算结果矩阵的四个子矩阵（划分方法同上），其中若按照东南西北方位划分矩阵，计算过程如下：

NW = P5+P4-P2+P6 NE = P1+P2

SW = P3+P4 SE = P1+P5-P3-P7

图示上述过程如下：



**图2-2 Strassen矩阵乘法计算过程**

1. **程序设计**
   1. **数据结构定义（数据成员）**

定义矩阵类：Matrix，其：

* + - 1. 拥有一个val：Int输入参数表征矩阵阶数，初始化要求n>0；
      2. 包含一个数据成员val elem：Array{IntArray(n){0} }，初始化一个n\*n的二维Int数组用以存储矩阵元素，所有元素初值为0；
      3. 包含一个函数成员getSize():Int返回当前Matrix类实例的参数n值，即矩阵大小。
  1. **核心功能实现（函数成员）**
     + 1. fun cutSubMatrix(A: Matrix, i: Int, j: Int, n: Int): Matrix

在矩阵A里以左上顶点为（0，0），右下顶点为（n-1，n-1），将从坐标（i，j）开始，大小为n的子矩阵切出构造为一个新的n阶矩阵并返回。

* + - 1. fun formMatrix(NW: Matrix, NE: Matrix, SW: Matrix, SE: Matrix): Matrix

用四个大小相等的矩阵NW,NE,SW,SE分别作为母矩阵的左上，右上，左下和右下四个部分，拼接成母矩阵并返回。

* + - 1. fun matrixMultiplication(A: Matrix, B: Matrix): Matrix

fun matrixAdd(A: Matrix, B: Matrix): Matrix

fun matrixMinus(A: Matrix, B: Matrix): Matrix

三个矩阵基本运算函数，按序输入两个运算矩阵并返回矩阵乘，加，减运算结果。

* 1. **输入输出与测试模块**
     + 1. fun readMatrix(filename: String): Matrix

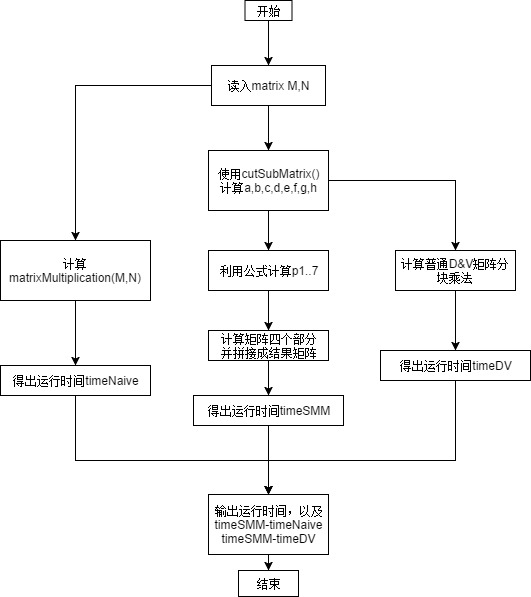
从文本文件读入一个矩阵，文件格式为：第一行为一个整数代表矩阵阶数n，接下来一行是n\*n个整数代表每一行n个数字依次输入，共n行。每个数字之间用空格隔开。

* + - 1. fun printMatrix(M: Matrix)

输出矩阵，格式为每行输出在同一行，以制表符\t完成五位数字以内的结果格式化，每行输出完后换行三次。

* + - 1. 输入文件命名统一为matrixM-n.txt以及matrixN-n.txt，其中n代表算例规模大小，输出到控制台，同时输出算法核心运行时间，以及其与简单矩阵相乘方法和简单分治相乘算法的运行时间差。
      2. 算例由generateMatrix程序实现，输入n后循环n^2次由1..n之间随机选择数字输出到文件。

具体程序流程图如下：



**图3-1 程序流程图**

1. **结果测试**
   1. **测试环境**

该程序在运行在Windows 10系统上的IntelliJ IDEA 19.2 Community版本编译通过并成功运行（Java12，Kotlin1.3）。

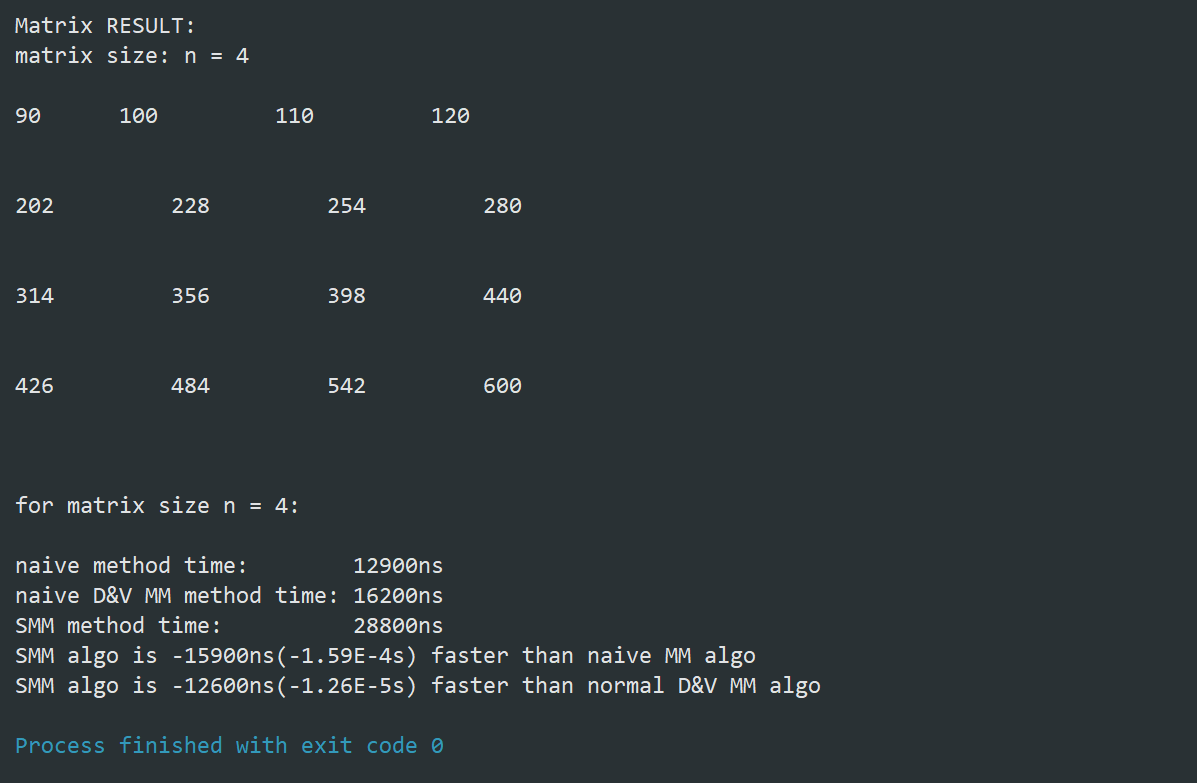
* 1. **测试样例与结果**
     1. **测试样例1**
        1. 输入：

正确性检查，输入两个相同的矩阵4, 1..16，矩阵文件如下：



**图4-1-1 sample1测试文件**

* + - 1. 测试结果：

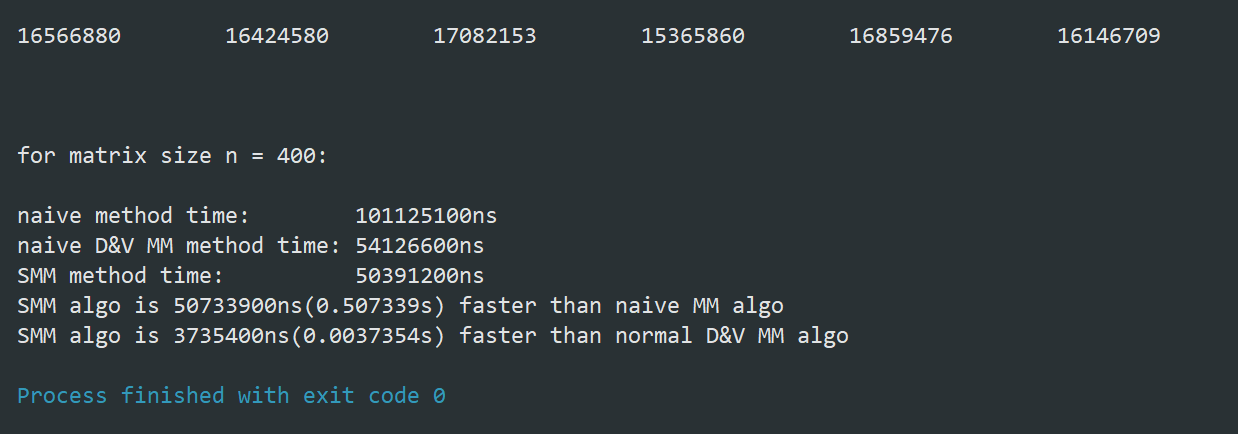


**图4-1-2 sample1测试结果**

* + 1. **测试样例2**
       1. 输入：

matrixM-400.txt，matrixN-400.txt

* + - 1. 测试结果：



**图4-2 sample2测试结果**

* + 1. **测试样例时间曲线**
       1. 输入：

输入对于问题规模分别为100，200，400，600，800的矩阵文件进行测试，得出三种方法的用时曲线。（每个规模的算例均测5次取平均以确保程序得到较稳定的结果）

* + - 1. 测试结果：

**图4-3 算法时间曲线（x: matrix size, y: time/ns）**

1. **时间复杂度分析**

一般的矩阵分治算法需要进行八次规模为N/2的矩阵乘法，由：

T(N) = 8T(N/2)+O(N^2)，由Master’s Theorem知复杂度为O(N^3)

而SMM算法只需要计算p1..7一共七次矩阵乘法，由：

T(N) = 7T(N/2)+O(N^2)，由Master’s Theorem知复杂度为O(N^log7)

约为O(N^2.8)，小于O(N^3)

注意：由于SMM算法需要占用额外空间处理子矩阵，所以在问题规模很小的时候比简单矩阵相乘更费时。但是矩阵规模不断增大，到矩阵乘法的代价以及远远大于申请空间和创建矩阵新对象的代价时，SMM的时间优势才会显现。

1. **源代码**

import java.io.File

import javax.swing.text.html.HTML.Attribute.N

//DATA STRUCTURE OF MATRIX---2D ARRAY AND getSize() FUNCTION

class Matrix(val n: Int) {

//if n=0, initialization failure

init {

require(n > 0) { println("invalid matrix initialization") }

}

//initialize n\*n Integer array elem

var elem = Array(n) { IntArray(n) { 0 } }

fun getSize(): Int {

return n

}

}

//CONSTRUCT SUB MATRIX FROM EXISTING MATRIX-----------------------------------//

//CONSTRUCT MATRIX FROM EXISTING SUB MATRIX

fun cutSubMatrix(A: Matrix, i: Int, j: Int, n: Int): Matrix {

if (n > A.getSize()) {

println("subMatrix initiation failure")

return A

}

var B = Matrix(n)

var i1 = i

var j1 = j

while (i1 < i + n) {

j1 = j

while (j1 < j + n) {

B.elem[i1 - i][j1 - j] = A.elem[i1][j1]

j1++

}

i1++

}

return B

}

fun formMatrix(NW: Matrix, NE: Matrix, SW: Matrix, SE: Matrix): Matrix {

val m = NW.getSize()

val n = m + m

var M = Matrix(n)

var i = 0

var j = 0

//copy NorthWest

i = 0

while (i < m) {

j = 0

while (j < m) {

M.elem[i][j] = NW.elem[i][j]

j++

}

i++

}

//copy NorthEast

i = 0

while (i < m) {

j = m

while (j < n) {

M.elem[i][j] = NE.elem[i][j - m]

j++

}

i++

}

//copy SouthWest

i = m

while (i < n) {

j = 0

while (j < m) {

M.elem[i][j] = SW.elem[i - m][j]

j++

}

i++

}

//copy SouthEast

i = m

while (i < n) {

j = m

while (j < n) {

M.elem[i][j] = SE.elem[i - m][j - m]

j++

}

i++

}

return M

}

//----------------------------------------------------------------------------//

//CALCULATION BETWEEN MATRIX--------------------------------------------------//

fun matrixMultiplication(A: Matrix, B: Matrix): Matrix {

if (A.getSize() != B.getSize()) {

println("Matrix size doesn't fit")

}

val n = A.getSize()

val C = Matrix(n)

var i = 0

var j = 0

var k = 0

while (i < n) {

j = 0

while (j < n) {

k = 0

while (k < n) {

C.elem[i][j] += A.elem[i][k] \* B.elem[k][j]

k++

}

j++

}

i++

}

return C

} //---checked

fun matrixAdd(A: Matrix, B: Matrix): Matrix {

if (A.getSize() != B.getSize()) {

println("Matrix size doesn't fit")

}

val n = A.getSize()

val C = Matrix(n)

var i = 0

var j = 0

while (i < n) {

j = 0

while (j < n) {

C.elem[i][j] = A.elem[i][j] + B.elem[i][j]

j++

}

i++

}

return C

}

fun matrixMinus(A: Matrix, B: Matrix): Matrix {

if (A.getSize() != B.getSize()) {

println("Matrix size doesn't fit")

}

val n = A.getSize()

var C = Matrix(n)

var i = 0

var j = 0

while (i < n) {

j = 0

while (j < n) {

C.elem[i][j] += A.elem[i][j] - B.elem[i][j]

j++

}

i++

}

return C

}

//----------------------------------------------------------------------------//

//IO-OP OF MATRIX-------------------------------------------------------------//

fun readMatrix(filename: String): Matrix {

val line = File(filename).readLines()

val n = line[0].toInt()

val readElem = line[1].split(" ").toList().toTypedArray()

if (readElem.size != n \* n) {

println("file reading error")

return Matrix(0)

}

var M = Matrix(n)

var i = 0

var j = 0

var k = 0

while (i < n) {

j = 0

while (j < n) {

M.elem[i][j] = readElem[k].toInt()

j++

k++

}

i++

}

return M

}

fun printMatrix(M: Matrix) {

val n = M.getSize()

if (n <= 0) {

println("invalid matrix input!")

return

}

println("matrix size: n = $n\n")

var i = 0

var j = 0

while (i < n) {

j = 0

while (j < n) {

print("${M.elem[i][j]} \t\t")

j++

}

i++

println("\n\n")

}

println()

}

//-----------------------------------------------------------------------------//

fun main(args: Array<String>) {

//file read

println("input filename of matrix M:")

val filename1 = readLine()!!

println("input filename of matrix N:")

val filename2 = readLine()!!

val M = readMatrix(filename1)

val N = readMatrix(filename2)

var n = 0

if (M.getSize() == N.getSize()) {

n = M.getSize()

} else {

println("file reading error")

return

}

val m = n / 2

println("Matrix for Multiplication:")

println("Matrix M:")

printMatrix(M)

println("Matrix N:")

printMatrix(N)

//naive matrix multi answer for comparison----------------------------//

val startTime = System.nanoTime()

val ans = matrixMultiplication(M, N)

val endTime = System.nanoTime()

println("naive ANSWER:")

printMatrix(ans)

//-----------------------------------------------------------------//

//construct Strassen factor --- checked

val A = cutSubMatrix(M, 0, 0, m)

val B = cutSubMatrix(M, 0, m, m)

val C = cutSubMatrix(M, m, 0, m)

val D = cutSubMatrix(M, m, m, m)

val E = cutSubMatrix(N, 0, 0, m)

val F = cutSubMatrix(N, 0, m, m)

val G = cutSubMatrix(N, m, 0, m)

val H = cutSubMatrix(N, m, m, m)

//naive D&V multi answer----------------------------------------------------------------------------//

val startTime1 = System.nanoTime()

val NorthWest1 = matrixAdd(matrixMultiplication(A, E), matrixMultiplication(B, G)) //NorthWest = ae+bg

val NorthEast1 = matrixAdd(matrixMultiplication(A, F), matrixMultiplication(B, F)) //NorthEast = af+bh

val SouthWest1 = matrixAdd(matrixMultiplication(C, E), matrixMultiplication(D, G)) //SouthWest = ce+dg

val SouthEast1 = matrixAdd(matrixMultiplication(C, F), matrixMultiplication(D, H)) //SouthEast = cf+dh

val endTime1 = System.nanoTime()

val result1 = formMatrix(NorthWest1, NorthEast1, SouthWest1, SouthEast1)

println("naive D&V MM ANSWER:")

printMatrix(result1)

//--------------------------------------------------------------------------------------------------//

//Strassen's Matrix Multiplication------------------------------------------------------//

//Step1 form factor 1-7

val startTime2 = System.nanoTime()

val P1 = matrixMultiplication(A, matrixMinus(F, H)) //p1 = a(f-h)

val P2 = matrixMultiplication(matrixAdd(A, B), H) //p2 = (a+b)h

val P3 = matrixMultiplication(matrixAdd(C, D), E) //p3 = (c+d)e

val P4 = matrixMultiplication(D, matrixMinus(G, E)) //p4 = d(g-e)

val P5 = matrixMultiplication(matrixAdd(A, D), matrixAdd(E, H)) //p5 = (a+d)(e+h)

val P6 = matrixMultiplication(matrixMinus(B, D), matrixAdd(G, H)) //p6 = (b-d)(g+h)

val P7 = matrixMultiplication(matrixMinus(A, C), matrixAdd(E, F)) //p7 = (a-c)(e+f)

//Step2 form 4 sub matrix

val NorthWest = matrixAdd(matrixAdd(P5, P6), matrixMinus(P4, P2)) //NW = P5+P4-P2+P6

val NorthEast = matrixAdd(P1, P2) //NE = P1+P2

val SouthWest = matrixAdd(P3, P4) //SW = P3+P4

val SouthEast = matrixAdd(matrixMinus(P1, P3), matrixMinus(P5, P7)) //SE = P1+P5-P3-P7

val endTime2 = System.nanoTime()

//Step3 form result matrix

val result = formMatrix(NorthWest, NorthEast, SouthWest, SouthEast)

//-------------------------------------------------------------------------------------//

println("Matrix RESULT:")

printMatrix(result)

println("for matrix size n = $n:\n")

println("naive method time: " + (endTime - startTime) + "ns")

println("naive D&V MM method time: " + (endTime1 - startTime1) + "ns")

println("SMM method time: " + (endTime2 - startTime2) + "ns")

println("SMM algo is ${(endTime - startTime) - (endTime2 - startTime2)}ns(${((endTime - startTime) - (endTime2 - startTime2)) / 100000000.00}s) faster than naive MM algo")

println("SMM algo is ${(endTime1 - startTime1) - (endTime2 - startTime2)}ns(${((endTime1 - startTime1) - (endTime2 - startTime2)) / 1000000000.0}s) faster than normal D&V MM algo")

// println("SMM algo is %${((endTime - startTime)- (endTime1 - startTime1))/(endTime - startTime)\*100} faster than normal D&V MM algo")

}