1. Data sets, test strategies, and results

The goal of the project was to program and compare the runtime complexity of 3 matrix multiplication algorithms: Classic, Divide-and-Conquer, and Strassen's. Each runtime presented is an average of 10 calls of the algorithm. The driver times each method call in milliseconds, averages the output, and prints total time for the current size. The maximum size the program was set to run up to was 2^16 or 65536*65536, however none of the algorithms ran past size 2048*2048. When this report refers to the program "stalling" or "running indefinitely", this signifies that after over 12 hours of continually running the program without disturbances, the program never printed the output for the next size, meaning that it was stuck at the current array size.

2 Data Sets were tested:

$\{0\}$ expanded with 0's as needed

$\begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$ expanded with 2's as needed

Fig 1.1: Classic Algorithm, Matrix Time Data:

Size (n*n)	1	2	4	8	16	32	64	128	256	512	1024	2048
0 Matrix (ms)	0	0	0	0	0	0	4	14	157	1264	11304	192496
1234 Matrix (ms)	0	0	0	0	0	0	3	20	152	1273	11224	192079

Both were run through the Classic matrix multiplication algorithm to compare runtimes, and managed to run up to matrix size 2048x2048 before the program stalled. The runtime values for the two data sets (above) were then compared and determined to be similar enough to assume that the computer will take the same amount of time whether it is multiplying all 0s or nonzero digits. Therefore the following graphs to compare the 3 algorithms were created using the expanding matrix of 0's for uniformity.

Fig 1.2: Classic Algorithm, Time vs Size graph

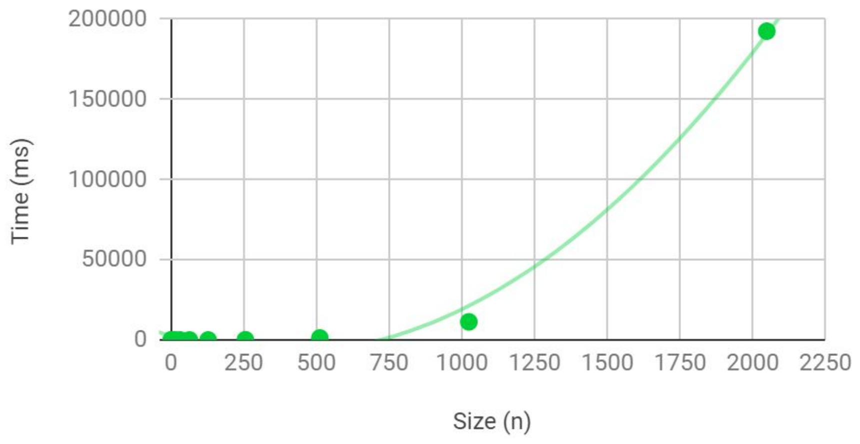
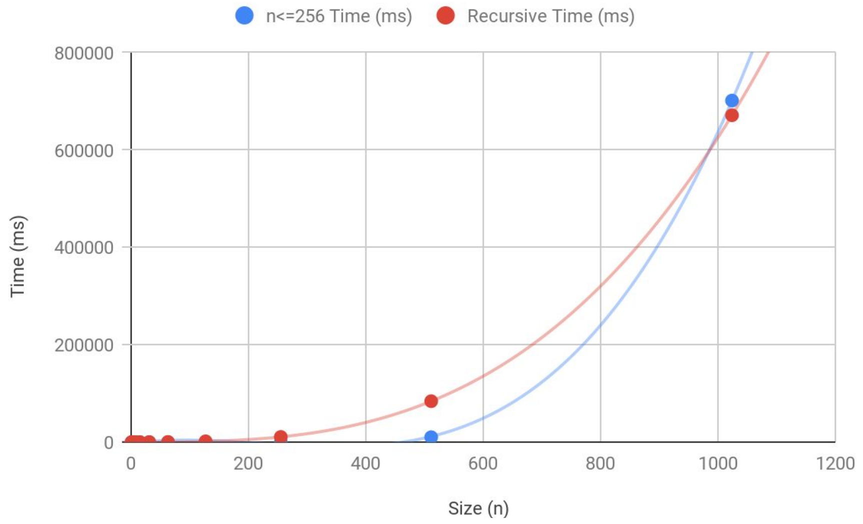


Fig 2.1: Divide and Conquer, Matrix Time Data

Size (n*n)	1	2	4	8	16	32	64	128	256	512	1024	2048
Recursive Time (ms)	0	0	0	0	0	39	250	1680	10620	84039	671358	-
n<=256 Time (ms)	0	0	0	0	0	0	7	8	203	10118	701494	-

Unlike the Classic algorithm, the Divide and Conquer algorithm stalled indefinitely at matrix size 1024*1024. The time also increased noticeably faster. Because of this, a modified version of the algorithm was also tested where matrices of size 256*256 and smaller were solved using the Classic method instead in an attempt to reduce the number of recursive calls and decrease time.

Fig 2.2: Divide and Conquer, Time vs Size graph

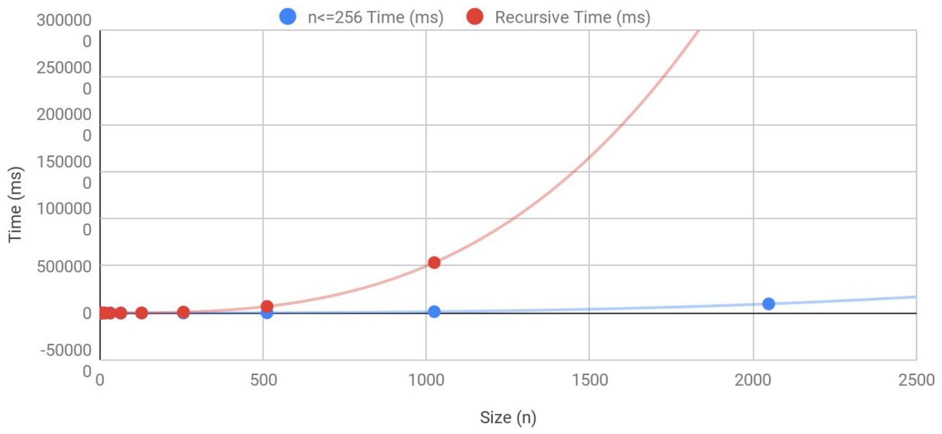


As shown in the graph, the purely recursive divide and conquer approach was slower than the hybrid approach. This is because while in theory the recursive approach is faster, in practice each recursive call is placed on the stack and takes a longer time for the computer to resolve. Therefore a method with recursion would take much longer than the iterative classic approach. However, in the hybrid approach, the graph eventually increases at a faster rate than the recursive approach. This shows that in the long term the purely recursive implementation of divide and conquer is faster than switching to the classic algorithm at a predetermined base case size.

Fig 3.1: Strassen's, Matrix Time Data

Size (n*n)	1	2	4	8	16	32	64	128	256	512	1024	2048
Recursive Time (ms)	0	0	0	7	8	39	211	1308	9990	70530	535124	-
n<=256 Time (ms)	0	0	0	0	0	0	8	46	395	2764	14313	97096

The Strassen's algorithm data also includes both a purely recursive test and a test with base case n<=256 switching to the classic method. Strassen's method was notable in that when a data cap was placed at n<=256, the program ran up to n=2048 before the IDE ran out of heap space, throwing a "java.lang.OutOfMemoryError" error. Whereas in the purely recursive run the program stalled after n=1024 was output.



As shown in the graph, the hybrid implementation of Strassen's algorithm, which included a base case call to the classic algorithm for size n<=256, was much faster than the purely recursive approach. This is likely because, unlike the divide and conquer algorithm that works with the indices of the original array directly, this implementation of Strassen's algorithm uses a divide method to recursively separate each matrix into 4 parts, so each recursive call takes additional time and space to separate the matrix, as explained in the theoretical complexity section below.

2. Theoretical Complexity Comparisons

I. Classic

$T(n) = n^3 = O(n^3)$

The classic algorithm is simple, consisting of 3 nested loops that each iterate up to size n times. Therefore the runtime complexity is $O(n^3)$.

II. Divide and Conquer

The divide and conquer algorithm makes 8 recursive calls on submatrices of size n/2. It also has 4 calls to the add() method on matrices of dimension n/2 to combine results together following the recursive multiplication.

$T(n) = 8T(n/2) + 4(n/2)^2 = 8T(n/2) + 4(n^2/4) = 8T(n/2) + n^2$ //simplify add n^2 constant

$T(n) = \{ \quad 8T(n/2) + n^2 \quad (n>1)$ //recurrence relation

$\quad\quad 1 \quad (n=1) \quad \}$

//Solve for Big O()

$T(n/2) = 8(8T(n/4) + (n/2)^2) + n^2$ //substitute T(n/2)

$T(n/2) = 8^2\, T(n/4) + (2^3n^2/2^2 + n^2) = 8^2\, T(n/4) + 2n^2 + n^2$ //simplify

$T(n/4) = 8^2(8T(n/8) + (n/4)^2) + 2n^2 + n^2 = 8^3T(n/8) + 2^6/2^4n^2 + 2n^2 + n^2 = 8^3T(n/8) + 2^2n^2 + 2n^2 + n^2$

$T(n) = 8^{\mathbf{k}}\, T(n/2^{\mathbf{k}}) + \Sigma(2^{\mathbf{k-1}})n^2$

let k = lg(n)

$= 8^{lgn}\,(T(n/2^{lgn}) + \Sigma(2^{\mathbf{lgn-1}})n^2 = 8^{lgn}\,(T(n/n)) + ((2^{lgn} -1)/(2-1)) = 8^{lgn}\,(T(1)) + (n - 1)$

$8^{lgn} = 2^{3(lgn)} = n^{3(lg2)} = n^3$

∴ $O(n^3)$, therefore the Divide and Conquer algorithm is the same runtime complexity as Classic.

III. Strassen's Algorithm

Strassen's algorithm uses 7 recursive calls of size n/2. There are also 18 iterative additions and subtractions on matrices of size n/2. This implementation also incorporates 8 calls of the divide() function to separate the matrix into 8 submatrices of size n/2. The base case T(2) = 13 because there is 1 comparison, 8 multiplications, and 4 additions when the matrix is at size 2*2.

$T(n) = 7T(n/2) + 18(n/2)^2 + 8(n/2)^2 = 7T(n/2) + 26(n/2)^2$ //simplify add n^2 constant

$T(n) = \{ \quad 7T(n/2) + 26(n/2)^2 \quad (n>2)$ //recurrence relation

$\quad\quad 13 \quad (n<=2) \quad \}$

$T(n/2) = 7T(n/4) + 26*(7/4)n^2 + 26/4n^2$

$T(n/4) = 7^2T(n/8) + 26*(7^2/4^2)n^2 + 26/4n^2$

assume $n = 2^k$ for some integer k

$= 7^{\mathbf{k-1}}\, T(n/2^{\mathbf{k-1}}) + \Sigma((7/4)^{\mathbf{k-2}})*(26/4)n^2$

$<= 7^{\mathbf{k-1}}\, T(2) + (((7/4)^{\mathbf{k}\text{-}1}-1)/((7/4)-1))*(26/4)n^2 = 7^{\mathbf{k-1}}\, T(2) + ((7/4)^{\mathbf{k-1}}-1)*(26/3)*n^2$

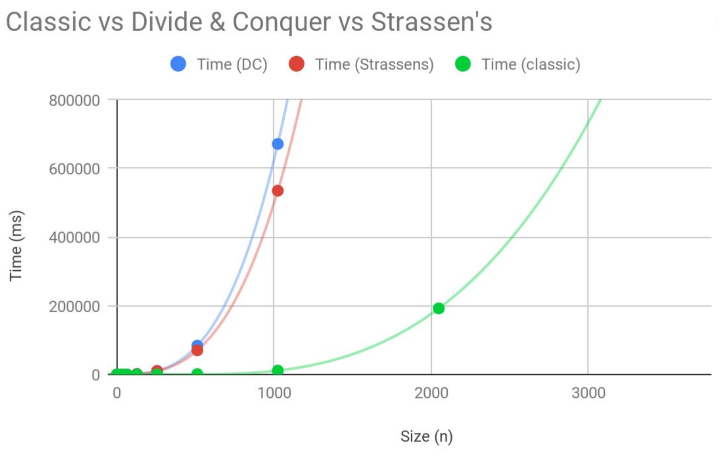
$= 13*7^k + (26/3)n^2*(7/4)^{\mathbf{k}} + (26/3)*(7/4)$

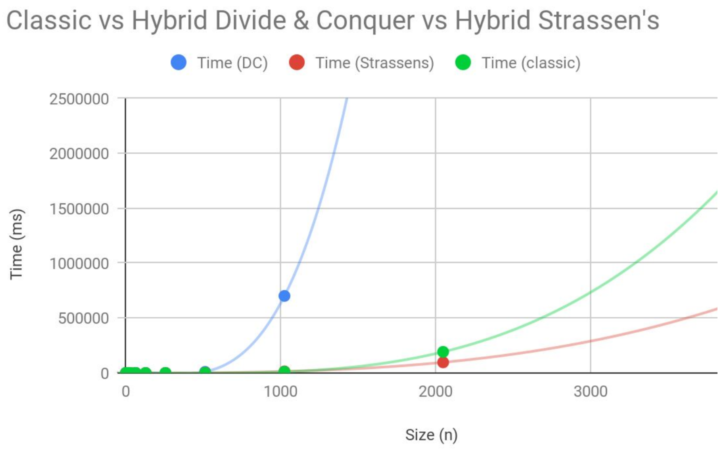
$= 13*7^{lgn} + (26/3)*n^2*(7/4)^{lgn} = 13*7^{lgn} + (26/3)n^2(n^{lg(7/4)})$

$= 13*n^{lg7} + cn^{lg7} = (13+c)n^{lg7} = n^{lg7}$

∴ $O(n^{lg7}) = O(n^{2.81})$, which is a faster runtime when compared to the previous algorithms.

3. Classical Matrix Multiplication vs Divide-and-Conquer Matrix Multiplication





As shown in the graphs above, the classic algorithm performed better than the divide and conquer algorithm in real time despite sharing the same $O(n^3)$ theoretical runtime complexity. This difference can likely be attributed to the additional runtime cost of the recursive calls involved in the divide and conquer algorithm. The fact that the classic algorithm is faster than the divide and conquer algorithm holds true for both the purely recursive implementation and the hybrid implementation that switches to the classic algorithm for sizes of $n<=256$.

As expected, the pure recursive implementation of Strassen's algorithm also outperforms the divide and conquer algorithm due to the reduced number of recursive calls. However, the increased cost of the purely recursive algorithm is significantly reduced once a base case to switch to the classic algorithm is added, as shown in the hybrid Strassen's graph comparison. Furthermore, once the $n<=256$ base case is added to Strassen's algorithm it outperforms even the classic algorithm at larger sizes, which can be attributed to the fact that at larger numbers the added cost of 256^3 from the classic base case becomes negligible compared to the faster runtime of $O(n^{2.81})$ versus $O(n^3)$.

4. Strength and Constraints

As shown in the theoretical complexity calculations, the original divide and conquer algorithm has a runtime complexity of $O(n^3)$, meaning that it has a theoretically equal runtime to classic matrix multiplication. However, due to the fact that recursive functions have to deal with processing additional code in the stack with each call of the recursive function, in reality the divide and conquer algorithm took much more time than the classic algorithm. One strength of this implementation was the use of index calculation within the original matrix to change the values, which eliminated the need to copy the data to temporary arrays for each addition in the algorithm. A constraint may be that the code is written to work only for matrices of size 2^n, however there is a resize() method included that could be used in any test cases to pad the matrix with 0's until it reaches the appropriate size.

In theory, Strassen's algorithm theoretically appears to perform better than the classic matrix multiplication algorithm. As shown by the data, a purely recursive implementation of Strassen's algorithm is still slower than the iterative 3-loops in classic matrix multiplication. The strength of Strassen's algorithm is in the reduction of recursive calls compared to the original divide and conquer algorithm, which reduces the theoretical complexity to less than $O(n^3)$. One constraint of the algorithm in this implementation was the fact that additional space had to be allocated to store the temporary matrices each time each matrix was divided into 4 submatrices. This may have been fixed with a refactoring of the code to allow the code to work with the matrix indices directly, however in the interest of simplicity the division into submatrices was the optimal decision.

5. Java Code

Matrix Implementation Class

```
class MatrixMx
{
    public MatrixMx() {    }

    //classic
    public int[][] classic(int[][] A, int[][] B)
    {
        int dim = A.length, sum = 0;
        int multiply[][] = new int[dim][dim];
        for (int c = 0; c < dim; c++)
        {
            for (int d = 0; d < dim; d++)
            {
                for (int k = 0; k < dim; k++)
                {
                    sum = sum + A[c][k]*B[k][d];
                }
                multiply[c][d] = sum;
                sum = 0;
            }
        }
        return multiply;
    }

    public void print(int[][] m)
    {
        for(int[] row:m)                   {
            System.out.print("|\t");
            for(int col:row)
                System.out.print(col+"\t");
            System.out.println("|");       }
        System.out.println();
    }

public int[][] resize(int[][] m, int newdims,
int filler)
    {
        int dim = m.length;
        int[][] rem = new int[newdims][newdims];
        for(int r=0;r<newdims;r++)
        {
            for(int c=0;c<newdims;c++)
            {
                if(r<dim && c<dim)
                    rem[r][c] = m[r][c];
                else
                    rem[r][c] = filler;
            }
        }
        return rem;
    }

private void add(int[][] added, int[][] A,
int[][] B, int row, int col, int dim)
    {
        for(int r=0; r<dim; r++)
        {
            for(int c=0; c<dim; c++)
            {
                added[r+row][c+col]=A[r][c]+B[r][c];
            }
        }
    }
```

```
//divide and conquer

public int[][] divconquer(int[][] A, int[][] B)
{
    int dim = A.length;
    return dcrecurse(A, B, 0,0,0,0, dim);
}

private int[][] dcrecurse(int[][] A, int[][] B, int
rowA, int colA, int rowB, int colB, int dim)
{
  int[][] multiply = new int[dim][dim];
  if(dim==1)
        multiply[0][0] = A[rowA][colA] * B[rowB][colB];
  //else if(dim<=256) //used in hybrid time test
  //    return classic(A,B)
  else
  {
   int newdims = dim/2;
   add(multiply,
   dcrecurse(A,B,rowA,colA,rowB,colB,newdims),
   dcrecurse(A,B,rowA,colA+newdims,rowB+newdims,colB,
        newdims)
   ,0,0,newdims); //m[0][0] = 00*00 + 01*10

   add(multiply,
   dcrecurse(A,B,rowA,colA,rowB,colB+newdims,newdims),
   dcrecurse(A,B,rowA,colA+newdims,rowB+newdims,
        colB+newdim s,newdims)
   ,0,newdims,newdims); //m[0][1] = 00*01 + 01*11

   add(multiply,
   dcrecurse(A,B,rowA+newdims,colA,rowB,colB,newdims),
   dcrecurse(A,B,rowA+newdims,colA+newdims,rowB+newdims,
       colB,newdims),
   newdims,0,newdims); //m[1][0] = 10*00 + 11*10

   add(multiply,
   dcrecurse(A,B,rowA+newdims,colA,rowB,colB+newdims,
       newdims),
   dcrecurse(A,B,rowA+newdims,colA+newdims,rowB+newdims,
       colB+newdims,newdims),
   newdims,newdims,newdims); //m[1][1] = 10*01 + 11*11
  }
  return multiply;
}
```

```
//strassens

    public int[][] strassens(int[][] A, int[][] B)
    {
        int dim = A.length;
        return strc(A,B,dim);
    }

    private int[][] strc(int[][] A, int[][] B, int dim)
    {
        int[][] multiply = new int[dim][dim];
        if(dim==2)
        {
       multiply[0][0] = A[0][0]*B[0][0]+A[0][1]*B[1][0];
       multiply[0][1] = A[0][0]*B[0][1]+A[0][1]*B[1][1];
       multiply[1][0] = A[1][0]*B[0][0]+A[1][1]*B[1][0];
       multiply[1][1] = A[1][0]*B[0][1]+A[1][1]*B[1][1];
        }
        //else if(dim<=256) //used in hybrid time test
        //    return classic(A,B);
        else
        {
            int newdims = dim/2;

            int[][] A00 = divide(A,0,0);
            int[][] A01 = divide(A,0,newdims);
            int[][] A10 = divide(A,newdims,0);
            int[][] A11 = divide(A,newdims,newdims);

            int[][] B00 = divide(B,0,0);
            int[][] B01 = divide(B,0,newdims);
            int[][] B10 = divide(B,newdims,0);
            int[][] B11 = divide(B,newdims,newdims);

            int[][] m1 =strc(A00,sub(B01,B11),newdims);
            int[][] m2 =strc(add(A00,A01),B11,newdims);
            int[][] m3 =strc(add(A10,A11),B00,newdims);
            int[][] m4 =strc(A11,sub(B10,B00),newdims);
  int[][] m5 = strc(add(A00,A11),add(B00,B11),newdims);
  int[][] m6 = strc(sub(A01,A11),add(B10,B11),newdims);
  int[][] m7 = strc(sub(A00,A10),add(B00,B01),newdims);

   add(multiply,m5,add(m6,sub(m4,m2)),0,0,newdims);
   //00=5+4-2+6
   add(multiply,m1,m2,0,newdims,newdims); //01=1+2
   add(multiply,m3,m4,newdims,0,newdims); //10=3+4
   add(multiply,m5,sub(sub(m1,m3),m7),newdims,newdims,
   newdims); //11=5+1-3-7
   }
        return multiply;
}
```

```
private int[][] divide(int[][] M, int row, int col)
    {
        int dims = M.length/2;
        int[][] div = new int[dims][dims];
        for(int i = 0;i<dims;i++)
        {
            for(int j = 0;j<dims; j++)
            {
                div[i][j] = M[i+row][j+col];
            }
        }
        return div;
    }

    private int[][] add(int[][] A, int[][] B)
    {
        int dim = A.length;
        int[][] added = new int[dim][dim];
        for(int r=0; r<dim; r++)
        {
            for(int c=0; c<dim; c++)
            {
                added[r][c] = A[r][c] + B[r][c];
            }
        }
        return added;
    }

    private int[][] sub(int[][] A, int[][] B)
    {
        int dim = A.length;
        int[][] subbed = new int[dim][dim];
        for(int r=0; r<dim; r++)
        {
            for(int c=0; c<dim; c++)
            {
                subbed[r][c] = A[r][c] - B[r][c];
            }
        }
        return subbed;
    }
```

Driver Class

```
public class Main
{
    static MatrixMx mxtester = new MatrixMx();

    public static void main(String[] args)
    {
        int[][] test = {{1,2},{3,4}};
        int[][] test0 = {{0}};
        data(test0, 0);
    }

    public static void data(int[][] m, int filler)
    {
        int dims = 2;
        int[][] resz = mxtester.resize(m,dims,filler);
        for(int i=0;i<16;i++)
        {
            timeCounter(resz);
            dims*=2;
            resz = mxtester.resize(m,dims,filler);
        }
    }
public static void timeCounter(int[][] m)
    {
       long startTime, runTime, avg = 0;
       for(int i=0;i<10;i++)
       {
          startTime = System.currentTimeMillis();
         //uncomment algorithm to test
          //testClassic(m);
          //testDC(m);
          //testStrassens(m);
          runTime=System.currentTimeMillis()-startTime;
          avg+=runTime;
       }
       avg/=2;
       int dim = m.length;
       System.out.println(dim +"*"+dim+" runtime:"+avg);
    }

    public static void testClassic(int[][] m)
    {
        //System.out.println("classic");
        int[][] classic = mxtester.classic(m,m);
        //mxtester.print(classic);
    }

    public static void testDC(int[][] m)
    {
        //System.out.println("divide & conquer");
        int[][] DC = mxtester.divconquer(m,m);
        //mxtester.print(DC);
    }

    public static void testStrassens(int[][] m)
    {
        //System.out.println("Strassen's");
        int[][] str = mxtester.strassens(m,m);
        //mxtester.print(str);
    }
}
```