Chapter 1

Introduction

The multiplication of two matrices is one of the most important Operations in linear algebra. The operation plays an important role in scientific computing. Since different algorithms give various differences in performance, finding a good algorithm seems to be valuable. The investigation of speed up matrix multiplication has become the main focus in scientific computation. The main objective was to design a program, which generates two matrices with various dimensions, and then multiplies the two matrices using the Strassen’s algorithm and the conventional algorithm. The performance of both algorithms was compared in terms of their time complexity and space complexity.

In [linear algebra](https://en.wikipedia.org/wiki/Linear_algebra), the Strassen algorithm, named after [Volker Strassen](https://en.wikipedia.org/wiki/Volker_Strassen), is an [algorithm for matrix multiplication](https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm). It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than [the fastest known algorithms](https://en.wikipedia.org/wiki/Coppersmith%E2%80%93Winograd_algorithm) for extremely large matrices.

The main focus of this paper is to compare the execution time complexity and space complexity between Strassen’s algorithm and the conventional algorithm for matrix multiplication. The aim is to design a program, which generates two matrices withVarious dimensions, and multiplies the two matrices using both the Stassen’s algorithm and the conventional algorithm. The execution time of each algorithm is recorded to evaluate the performance of each algorithm.

The programming language used in this project is C. Some of the main achievements in this project are, successfully divide matrices into blocks, the Strassen’s algorithm was applied to each blocks recursively, and the level of recursion was controlled. The overall finding is that the Stassen’s algorithm is more efficient than conventional algorithm on large size of matrices. However, in scientific computing, memory has to be considered. The results show that Strassen’s algorithm needs more memory allocations than the conventional algorithm, due to the fact in design that more arrays need to be created. Application of Strassen algorithm makes a significant contribution to optimize the algorithm. Therefore, thorough study based on time complexity of matrix multiplication algorithm is very important.

Chapter 2

Types of Matrix Multiplication

**Volker Strassen** first published this algorithm in 1969 and proved that the n3 general matrix multiplication algorithm wasn't optimal. The Strassen algorithm is only slightly better, but its publication resulted in much more research about matrix multiplication that led to faster approaches, such as the Coppersmith–Winograd algorithm.

Until the late 1960s it was believed that computing the product C of two n × n matrices requires essentially a cubic number of operations, as the fastest algorithm known was the naïve algorithm which indeed runs in O(n3) time. In 1969, Strassen excited the research community by giving the first sub cubic time algorithm for matrix multiplication, running in O(n2.8) time. This amazing discovery spawned a long line of research which gradually reduced the matrix multiplication exponent over time.

In 1978, Pan showed ὼ < 2:796, the following year, Bini et al. introduced the notion of border rank and obtained ὼ < 2:78. Sch¨onhage generalized this notion in 1981, proved his theorem (also called the asymptotic sum inequality), and showed that ὼ< 2:548. In 1986, Strassen introduced his laser method which allowed for an entirely new attack on the matrix multiplication problem. He also decreased the bound to ὼ<2:479. Three years later, Coppersmith and Winograd combined Strassen’s technique with a novel form of analysis based on large sets avoiding arithmetic progressions and obtained the famous bound of ὼ<2:376 which has remained unchanged for more than twenty years. In 2003, Cohn and Umans introduced a new, group-theoretic framework for designing and analysis.

In 2005, together with Kleinberg and Szegedy , they obtained several novel matrix multiplication algorithms using the new framework, however they were not able to beat 2:376. In fact, both Coppersmith and Winograd .and Cohn et al. presented conjectures which if true would imply ὼ = 2. Recently, Alon, Shpilka and Umans showed that both the Coppersmith Winograd conjecture and one of the Cohn et al.

Chapter 3

Implementation of Conventional Matrix Multiplication

Conventional matrix multiplication works similar as matrix addition and subtraction. However, the conventional matrix multiplication requires three for loop iterations. Moreover, the most inner one is the row by column matrix multiplication.

The following code illustrates the implementation based on such definition.

for (i=0;i<n;i++)

for (j=0;j<n;j++)

{

for (k=0,t=0;k<n;k++)

t+=A.m[i][k]\*B.m[k][j];

C.m[i][j]=t;

}

The standard method of matrix multiplication of two n × n matrices takes O ( n 3) operations. Strassen’s algorithm is a Divide-and-Conquer algorithm that is asymptotically faster, i.e. O ( n). The usual multiplication of two 2 × 2 matrices takes 8 multiplications and 4 additions. Strassen showed how two 2 × 2 matrices can be multiplied using only 7 multiplications and 18 additions multiplications are much more expensive and it makes sense to trade one multiplication operation for 18 additions.

Divide each n x n matrix into four matrices of size (n/2)x(n/2):

Computing all of requires 8 multiplications and 4 additions.

Therefore, the total running time is given by

/storage/emulated/0/.polarisOffice5/polarisTemp/image3.emf

T (n) = O (n3)

Chapter 4

Strassen's Algorithm

Strassen's 1969 algorithm, which gives ὼ < 2:81 follows similarly.

A = A1;1 A1;2

A2;1 A2;2

B = B 1;1 B 1;2

B 2;1 B 2;2

C = C 1;1 C 1;2

C 2;1 C 2;2

M1:= (A1;1 + A2;2)(B1;1 + B2;2)

M2:= (A2;1 + A2;2)B1;1

M3:= A1;1(B1;2 - B2;2)

M4:= A2;2(B2;1 - B1;1)

M5:= (A1;1 + A1;2)B2;2

M6:= (A2;1-A1;1)(B1;1 + B1;2)

M7:= (A1;2-A2;2)(B2;1 + B2;2)

C1;1= M1 +M4 -M5 +M7

C1;2= M3 +M5

C2;1= M2 +M4

C2;2= M1- M2 +M3 +M6

Instead of using the 8 multiplications of the trivial approach, Strassen's algorithm only uses 7. Applying a divide and conquer strategy recursively allows matrix multiplication over n = 2N size matrices to be performed using only 7k = 7log2n = nlog27 = O(n2.81) multiplications.

Chapter 5

C Code Implementation

#include<stdio.h>

int main(){

  int a[2][2],b[2][2],c[2][2],i,j;

  int m1,m2,m3,m4,m5,m6,m7;

 printf("Enter the 4 elements of first matrix: ");

  for(i=0;i<2;i++)

      for(j=0;j<2;j++)

          scanf("%d",&a[i][j]);

 printf("Enter the 4 elements of second matrix: ");

  for(i=0;i<2;i++)

      for(j=0;j<2;j++)

          scanf("%d",&b[i][j]);

 printf("\nThe first matrix is\n");

  for(i=0;i<2;i++){

     printf("\n");

      for(j=0;j<2;j++)

          printf("%d\t",a[i][j]);

  }

 printf("\nThe second matrix is\n");

 for(i=0;i<2;i++){

     printf("\n");

      for(j=0;j<2;j++)

          printf("%d\t",b[i][j]);

  }

  m1= (a[0][0] + a[1][1])\*(b[0][0]+b[1][1]);

  m2= (a[1][0]+a[1][1])\*b[0][0];

  m3= a[0][0]\*(b[0][1]-b[1][1]);

  m4= a[1][1]\*(b[1][0]-b[0][0]);

  m5= (a[0][0]+a[0][1])\*b[1][1];

  m6= (a[1][0]-a[0][0])\*(b[0][0]+b[0][1]);

  m7= (a[0][1]-a[1][1])\*(b[1][0]+b[1][1]);

 c[0][0]=m1+m4-m5+m7;

 c[0][1]=m3+m5;

 c[1][0]=m2+m4;

 c[1][1]=m1-m2+m3+m6;

  printf("\nAfter multiplication: \n");

   for(i=0;i<2;i++){

     printf("\n");

      for(j=0;j<2;j++)

          printf("%d\t",c[i][j]);

   }

   return 0;

Chapter 6

Time Complexity

The standard matrix multiplication takes approximately 2N3 (where N = 2n) arithmetic operations (additions and multiplications); the asymptotic complexity is Θ(N3).

The number of additions and multiplications required in the Strassen algorithm can be calculated as follows: let f(n) be the number of operations for a 2n × 2n matrix. Then by recursive application of the Strassen algorithm, we see that f(n) = 7f(n−1) + ℓ4n, for some constant ℓ that depends on the number of additions performed at each application of the algorithm. Hence f(n)=(7+o(1))n, i.e., the asymptotic complexity for multiplying matrices of size N = 2n using the Strassen algorithm is

O([7+o(1)]n ) = o(N log27 +o(1)) =o(n2.8074)

The reduction in the number of arithmetic operations however comes at the price of a somewhat reduced [numerical stability](https://en.wikipedia.org/wiki/Numerical_stability), and the algorithm also requires significantly more memory compared to the naive algorithm. Both initial matrices must have their dimensions expanded to the next power of 2, which results in storing up to four times as many elements, and the seven auxiliary matrices each contain a quarter of the elements in the expanded ones.

Chapter 7

Time Complexity Derivation

T(n)= b ; n<=2

7T(n/2)+an2 ; n>2

General form:

T(n) = aT(n/b) + f(n)

a=7 , b=2 f(n) = an2

w.k.t

n=bk

n=2k , k=log2n

T(n) = 7T(n/2) + an2

= 7[7 T(n/4) + a(n/2)2]+an

= 72T(n/4) + 7 a(n/2)2  + an2

= 73 T(n/8) + 72 a(n/4)2  + 7 a(n/2)2+ an2

= 7kT(n/2k)+7k-1an2/(2k-1)2+7k-2 an2/(2k-2)2  +……………..+ 7an2/4 + an2

= 7kT(n/2k) + an2 {7k-1 /(4k-1) + 7k-2/(4k-2) + ……………..+7 /4 +1}

= 7kT(1) + an2 {1 + 7 /(4) + 72/(42) + ……………..+(7 /4)k-1

<= cn2(7/4)log2n + 7 log2n , where ‘C’ is a constant

= cn log24+ log27- log24 + n log27

= 7 log2n

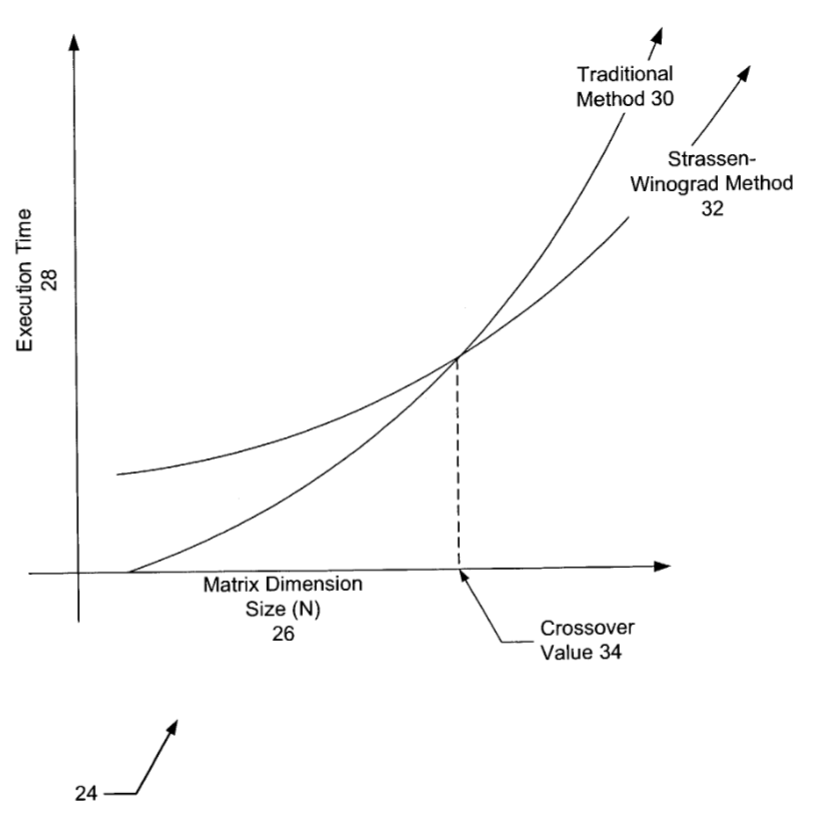
**T (n) = 7k**

**T (n) = O (n log27)**

**T (n) = O (n2.81)**

Chapter 8

Graphical Analysis

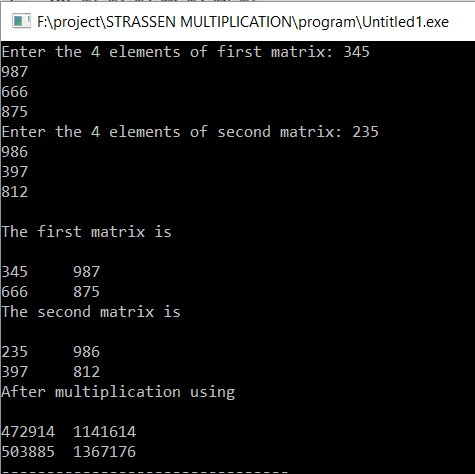


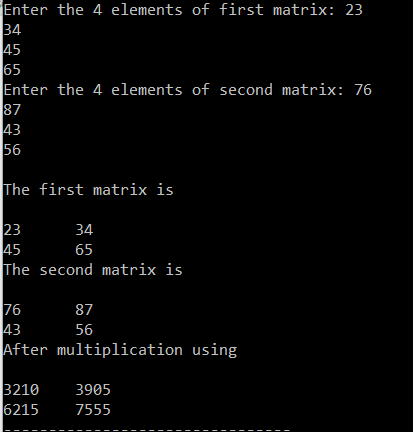
Chapter 9

Results

The C code was implemented on Turbo C.

Here are some of the results obtained for our algorithm:





Chapter 10

Advantages

Strassen algorithm is always better than the general algorithm for matrix multiplication in terms of time complexity. And the advantage of the Strassen algorithm will became more obvious with the increase of dimension of matrix. Strassen algorithm adopts the method of recursive algorithms, which is helpful to the implementation and analysis of the algorithm. However, the method make the algorithm require a great number of dynamic two dimensional arrays so as to assign memory space. Which enhance the time complexity and space complexity when the dimension of matrix is smaller. However, the general algorithm for matrix multiplication is no problem on this aspect. Tests show when the dimension of matrix is less than 500, this situation will get even worse. In order to solve this problem, we can specify a numeric value (such as 500) for algorithm routine, and we will adopt Strassen algorithm or the general algorithm for matrix multiplication based on the comparisons between the dimension and the numeric value.

Chapter 11

Limitations

Generally Strassen’s Method is not preferred for practical applications for following reasons:

* The biggest drawback of Strassen’s matrix multiplication is that it is restricted only for square matrices with 2n dimensions.
* For Sparse matrices, there are better methods especially designed for them.
* The sub matrices in recursion take extra space.
* Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen’s algorithm than in Naive Method.

Chapter 12

Conclusion

Matrix multiplication is an important computational kernel, and its performance can dictate the overall performance of many applications. Strassen's algorithm for matrix multiplication achieves lower arithmetic complexity, , than the conventional algorithm, O(n3), at the cost of worse locality of reference. Furthermore, since Strassen's algorithm is based on divide-and-conquer, an implementation must handle odd-size matrices, and reduce recursion overhead by terminating the recursion before it reaches individual matrix elements. These issues make it difficult to obtain efficient implementations of Strassen's algorithm. The complex testing between conventional matrix multiplication and Strassen’s algorithm was successful. However, there are still some changes can be made for further investigation and evaluation.

Firstly, from the testing, it can be observed that the Strassen’s algorithm was not efficient when applied to small size matrix. In that, preceding the recursion all the way down to one level would result in significant recursion overhead. Nevertheless, it would still be interesting to know at which point Strassen’s algorithm can beat conventional matrix multiplication.

For 512<n, Strassen’s approach costs the least number of steps. For n<7, the naïve algorithm for matrix multiplication is preferred.  
As an example, a 6x6 matrix requires 482 steps for the method of 4- Russians, but 468 steps for the naïve multiplication. For 6<n<513, the method of 4-Russians is most efficient.

RUSSIAN (EGYPTIAN) PEASANT’S MULTIPLICATION

Chapter 1

Introduction

Since the invention of multiplication, many have strived to improvise the basic algorithm. As numbers and mathematics grew more complex so did the algorithms. Hence there was increase in the time complexity and decreased efficiencies.

The evidence can be found in computer graphics, coordinate transformations such as scaling, rotation, and translation for robotics, the speed for solving those problems are only depends on the execution times of the algorithm.

The most common type of algorithms followed basically 3 steps namely, multiply, shift, and add. This also requires knowing more multiplication tables even for a single digit.

This lead to inventions of new algorithm’s late in 1800’s.

The programming language used this project is C. Some of the main achievements in this project are, successfully divide matrices into blocks, the Russian’s algorithm was applied to each blocks recursively, and the level of recursion was controlled. The overall finding is that the Russian’s algorithm is more efficient than conventional algorithm on large size of matrices. However, in scientific computing, memory has to be considered. The results show that Russian’s algorithm needs more memory allocations than the conventional algorithm, due to the fact in design that more arrays need to be created .Application of Russian algorithm makes a significant contribution to optimize the algorithm. Therefore, thorough study based on time complexity of multiplication algorithm is very important. This paper talks about the time complexity of Russian algorithm and general algorithm for basic multiplication, and makes a comparison between the two algorithm routines so as to discuss the advantages and disadvantages of Russian algorithm.

Chapter 2

Different Types of Multiplication

In mathematics, ancient Egyptian multiplication (also known as Egyptian multiplication, Ethiopian multiplication, Russian multiplication, or peasant multiplication), one of two multiplication methods used by scribes, was a systematic method for multiplying two numbers that does not require the [multiplication table](https://en.wikipedia.org/wiki/Multiplication_table), only the ability to multiply and [divide by 2](https://en.wikipedia.org/wiki/Division_by_2), and to [add](https://en.wikipedia.org/wiki/Addition). It decomposes one of the [multiplicands](https://en.wikipedia.org/wiki/Multiplicand) (generally the larger) into a sum of [powers of two](https://en.wikipedia.org/wiki/Powers_of_two) and creates a table of doublings of the second multiplicand. This method may be called mediation and duplation, where [mediation](https://en.wikipedia.org/wiki/Division_by_two) means halving one number and duplation means doubling the other number. It is still used in some areas.

Although in ancient Egypt the concept of base 2 did not exist, the algorithm is essentially the same algorithm as long multiplication after the multiplier and multiplicand are converted to [binary](https://en.wikipedia.org/wiki/Binary_numeral_system). The method as interpreted by conversion to binary is therefore still in wide use today as implemented by binary multiplier circuits in modern computer processors.

Chapter 3

Methodology

Using multiplication to directly calculate the frequency was not as big a problem as initially thought. One factor simplifying the direct multiplication was that the Hertz/count multiplier is a constant. Using a shift and add multiplication technique simplified the remainder of the multiplication “problem.”

Instead of using the MUL instruction for a multiplication of 8X8 unsigned multiply with a 16-bit product requiring a 40 bit (or larger) result and a multiplier of 32 bits, choosing shift and add algorithm which was easier to understand and code. An added advantage is that the algorithm is easily extended for any number of multiplicand bytes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Chapter 4  Russian peasant multiplication  In the Russian peasant method, the powers of two in the decomposition of the multiplicand are found by writing it on the left and progressively halving the left column, discarding any remainder, until the value is 1 (or -1, in which case the eventual sum is negated), while doubling the right column as before. Lines with even numbers on the left column are struck out, and the remaining numbers on the right are added together.  For example, to multiply 238 by 13, the smaller of the numbers (to reduce the number of steps), 13, is written on the left and the larger on the right. The left number is progressively halved (discarding any remainder) and the right one doubled, until the left number is 1: | | | | |
|  | 13 |  | 238 |  |
|  | 6 | (Remainder discarded) | 476 |  |
|  | 3 |  | 952 |  |
|  | 1 | (Remainder discarded) | 1904 |  |
| Lines with even numbers on the left column are struck out, and the remaining numbers on the right are added, giving the answer as 3094: | | | | |
|  | 13 |  | 238 |  |
|  | 6 |  | 476 |  |
|  | 3 |  | 952 |  |
|  | 1 |  | 1904 |  |
|  |  |  |  |  |
|  |  |  |  |  |
| The algorithm can be illustrated with the binary representation of the numbers: | | | | |
| 1101 | (13) | 11101110 | (238) |  |
| 110 | (6) | 111011100 | (476) |  |
| 11 | (3) | 1110111000 | (952) |  |
| 1 | (1) | 11101110000 | (1904) |  |
|  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | (238) |
| × |  | | | | | | | | 1 | 1 | 0 | 1 | (13) |
|  | | | | | | | | | | | | |  |
|  | | | | | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | (238) |
|  | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | (0) |
|  | | | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | (952) |
| + |  | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | (1904) |
|  | | | | | | | | | | | | |  |
|  | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | (3094) |

Chapter 5

Russian Peasant’s Algorithm

ALGORITHM Russian ( P, Q , struct stack x)

// To multiply 2 numbers using Russian peasant`s algorithm

//Input: Two positive numbers P and Q

//Output: product of P and Q

Prod =0

x.top=1

i= x.top

if (p==0) or q==0)

prod =0

else

while (p>=1)

if( (p%2)!=0)

i++

x.a[i]=q

p=p/2

q=q\*2

for j form 0 to I , do

prod = prod+ x.a[j]

return prod

Chapter 6

Algorithm Working

The shift and add algorithm we used is commonly known as the Russian Peasant algorithm. For those (few) of you not familiar with this algorithm, we will attempt to provide a simple description:

1. In this description, we use the term “multiplier” to designate one of the numbers to be multiplied and “multiplicand” to designate the other number. Of course, both numbers are multiplicands.

2. Multiplication is performed in a sequence of steps. For each step, one multiplicand (which I call the multiplier) is shifted right one bit (divided by 2) and the other multiplicand is shifted left one bit (multiplied by 2).

3. Thus, at each step, the product of the “multiplier” and the multiplicand is the same. When the multiplier is divided down to 1, the multiplicand contains the approximate result.

4. The result is approximate because each time a 1 is shifted right from the least significant bit of the multiplier, there is a reminder that is equal to the value of the previous multiplicand. This is because one multiplicand’s worth of the result is “lost” when the multiplier bit is shifted right.

5. The remainders generated when shifting odd numbered multipliers can be accumulated as the algorithm progresses. When shifting is complete, the accumulated remainders can be added to the multiplicand result to yield an accurate result.

6. In the general case, the number of additions is variable and depends on the number of bits in the multiplier. Also, the total number of shifts varies, depending on the position of the most significant bit in the multiplier. Because the multiplier is a known constant, it is possible to simplify the shifting calculation by noting the position of the most significant 1 bit and performing a fixed number of shifts.

Chapter 7

C Code

#include<stdio.h>

#include<conio.h>

struct stack

{

longint a[10];int top;

}s;

longint \*allocate();

void accept(long int \*,long int \*);

longint multiply(long int \*,long int\*,struct stack \*);

void display(long int);

void main()

{

struct stack \*x=&s;

longint \*p,\*q,product;

clrscr();

p=allocate();

q=allocate();

accept(p,q);

product=multiply(p,q,x);

display(product);

}

longint \*allocate()

{

longint \*a=(long int \*)malloc(sizeof(long int));

return a;

}

void accept(long int \*p,longint \*q)

{

int i;

printf("Enter 2 nos b/w 0 and 999:\n");

scanf("%ld%ld",&(\*p),&(\*q));

if(((\*p)>999)||((\*q)>999))

{

for(i=1;i>0;i++)

{

printf("\nRussian multiplication is applicable only for nos b/w 0 and 999.\nSoplz enter no b/w 0 and 999 only:\n");

printf("\nOr enter -1 to exit:\n");

scanf("%ld",&(\*p));

if((\*p)==-1)

{

exit(0);

}

else

{

scanf("%ld",&(\*q));

if((\*p)<=999&&(\*q)<=999)

{

i=-1;

}

} }

}

}

longint multiply(long int \*p,longint \*q,struct stack \*x)

{

inti,j;longint prod=0;

(\*x).top=-1;

i=(\*x).top;

if(\*p==0||\*q==0)

{ prod=0;

}

else

{ while(\*p>=1)

{

if(((\*p)%2)!=0)

{

++i;

(\*x).a[i]=\*q;

}

\*p=(\*p)/2.0;

\*q=(\*q)\*2;

}}

for(j=0;j<=i;j++)

{prod=prod+(\*x).a[j];

}

return prod;

}

void display(long int c)

{ printf("\nProduct: %ld",c);

getch();}

Chapter 8

GUI Implementation

#include<stdio.h>

#include<conio.h>

#include<graphics.h>

#include<dos.h>

struct stack

{

long int a1[20],a2[20];int top1,top2;

}s;

long int \*allocate();

void accept(long int \*,long int \*);

void multiply(long int \*,long int\*,struct stack \*);

void main()

{

struct stack \*x=&s;

long int \*p,\*q;

int grd,grm;

detectgraph(&grd,&grm);

initgraph(&grd,&grm,"");

p=allocate();

q=allocate();

accept(p,q);

multiply(p,q,x);

closegraph();

return;

}

long int \*allocate()

{

long int \*a=(long int \*)malloc(sizeof(long int));

return a;

}

void accept(long int \*p,long int \*q)

{

int i;

printf("Enter 2 nos b/w 0 and 999:\n");

scanf("%ld%ld",&(\*p),&(\*q));

if(((\*p)>999)||((\*q)>999))

{

for(i=1;i>0;i++)

{

printf("\nRussian multiplication is applicable only for nos b/w 0 and 999.\nSo plz enter no b/w 0 and 999 only:\n");

printf("\nOr enter -1 to exit:\n");

scanf("%ld",&(\*p));

if((\*p)==-1)

{

exit(0);

}

else

{

scanf("%ld",&(\*q));

if((\*p)<=999&&(\*q)<=999)

{

i=-1;

}

} }

}

}

void multiply(long int \*p,long int \*q,struct stack \*x)

{

int i,j,c,v=0,k=0,b=0,r,d=0,a=0,w=0,n=0,y=0,t=0,m=0,g=0,l=0,h=0,z=0,f=250;long int sum=0;

int xps,yps,num,xpos,ypos,xp,yp,xs,ys,width=0;

char c9[90],c1[90],c2[90],c3[90],c4[90],c5[90],c6[90],s[20],c8[90];

num=getmaxcolor();

xpos=getmaxx()/2;

ypos=getmaxy()/3;

xps=xpos;

yps=ypos;

xs=xpos;

ys=ypos;

setbkcolor(BLUE);

setcolor(YELLOW);

(\*x).top1=-1;

(\*x).top2=-1;

i=(\*x).top1;

j=(\*x).top2;

if(\*p==0||\*q==0)

{

sum=0;

}

else

{ setcolor(WHITE);

outtextxy(xpos-30,ypos-30,"Successive Multiplication");

outtextxy(xpos-15,ypos-15," Array");

outtextxy(xpos-230,ypos-30,"Successive Division");

outtextxy(xpos-200,ypos-15," Array");

setcolor(YELLOW);

while(\*p>=1)

{

i++;

(\*x).a1[i]=\*p;

\*p=(\*p)/2;

sprintf(c1,"%ld",(\*x).a1[i]);

outtextxy((xpos/2)-20,ypos+10,c1);

ypos=ypos+10;

j++;

(\*x).a2[j]=\*q;

\*q=(\*q)\*2;

sprintf(c2,"%ld",(\*x).a2[j]);

outtextxy(xpos+10,ypos,c2);

ypos=ypos+10;

}

xpos=getmaxx()/2;

ypos=getmaxy()/3;

a=i;

b=i;

m=j;

while((a+1)>=0)

{

setcolor(WHITE);

while(b>=0)

{

sprintf(c5,"a[%d]",b);

b--;

outtextxy(xs-230,ys+(b\*20)+30,c5);

}

setcolor(WHITE);

rectangle(xpos-190,ypos,xpos-120,ypos+v);

v=v+20;

a--;

}

while((j+1)>=0)

{

setcolor(WHITE);

while(m>=0)

{ sprintf(c6,"b[%d]",m);

m--;

outtextxy(xs-35,ys+(m\*20)+30,c6);

}

setcolor(WHITE);

rectangle(xpos,ypos,xpos+70,ypos+g);

g=g+20;

j--;

}

while(k<=i)

{

xp=getmaxx()/2;

yp=getmaxy()/3;

delay(1000);

if(((\*x).a1[k])%2!=0)

{

d++;

xp=getmaxx()/2;

yp=getmaxy()/3;

setcolor(BROWN);

outtextxy(xp+76,yp+(k\*20)+10,"Selected");

setcolor(RED);

rectangle(xps,yps+((d-1)\*20),xps+70,yps+(d\*20));

setcolor(WHITE);

outtextxy(xp-310,yp+250,"Product=");

setcolor(YELLOW);

t=(\*x).a2[k];

h=0;

while(t!=0)

{

h++;

t=t/10;

}

w=w+h;

if(k==i)

{

sprintf(c8,"%ld",(\*x).a2[k]);

n=-f+(w\*14);

outtextxy(xp+n,yp+250,c8);

}

else

{

sprintf(c3,"%ld+",(\*x).a2[k]);

n=-f+(w\*14);

outtextxy(xp+n,yp+250,c3);

}

sum=sum+(\*x).a2[k];

k++;

}

else

{ d++;

k++;

}

}

xp=getmaxx()/2;

yp=getmaxy()/3;

setcolor(BROWN);

sprintf(c3,"=%ld",sum);

outtextxy(xp-f-4,yp+260,c3);

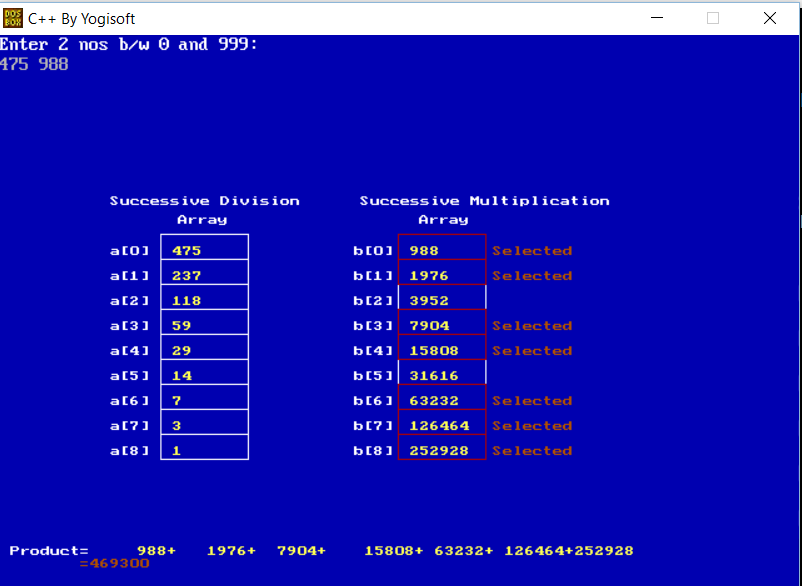
}

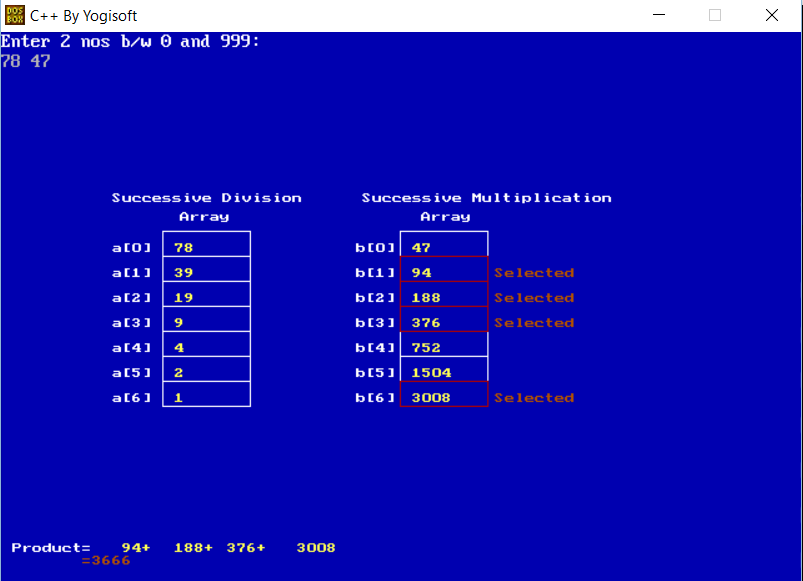
getch();

}

Chapter 9

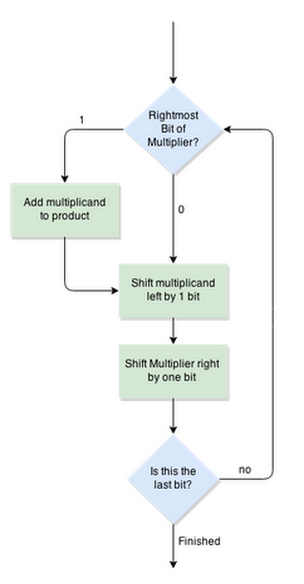
GUI Results





Chapter 10

Flowchart

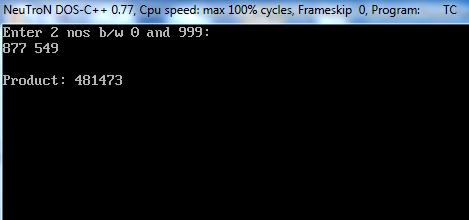


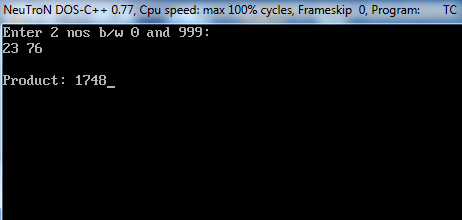
Chapter 11

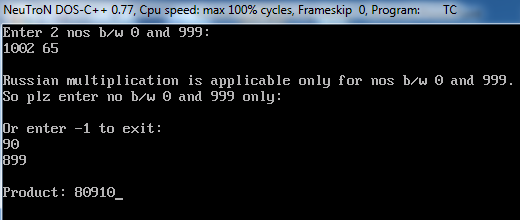
Results

As evident from the algorithm, for numbers greater than 999 are not applicable fr Russian Multiplication.

Here are some of the results obtained for the algorithm:







Chapter 12

Efficiency

The efficiency of basic multiplication is low compared to this algorithm. The efficiency in this algorithm is increased as it involves dividing and summing up the result. Hence the efficiency for this algorithm will be:

O (log2(n))

Chapter 13

Applications

This algorithm is used in real world applications to make multiplication much easier to do in your mind. Since only divide and addition operations are involved this algorithm can be applied to various situations and problems, thus saving time in complex problems. Application of Russian algorithm makes a significant contribution to optimize the algorithm. Therefore, thorough study based on time complexity of multiplication algorithm is very important.

Chapter 14

Drawbacks

Every algorithm has some drawback. The drawback in basic algorithm was its decreased efficiency, which is solve in Russian algorithm. This algorithm also have drawback in its implementation. The drawbacks involved are:

Russian Peasant Multiplication method is not applicable for numbers greater than 1000.

This algorithm will not work for decimals.

Chapter 15

Conclusion

Although using the MUL instruction may have produced faster and/or smaller code, Russian Peasant multiplication was easier for me to understand and implement. Extending the algorithm for larger multiplicands is very straightforward, this is important to me for future applications. Implementation of the algorithm resulted in a small amount of simple code. The process was also very instructive.

To understand ancient Egyptian multiplication and division, Ahmes' 2/n table aliquot part arithmetic operational steps must be translated into modern arithmetic statements. Ahmes multiplication and division methods were inverse to each other, provided vivid examples of the arithmetic relationships. The details of two rational number conversion methods were detailed, one for n/p, n/pq, 2/p and 2/pq and another for hard to convert n/p rational numbers that were parsed into solvable 2/p and (n-2)/p statements.

 Multiplication is an important computational kernel, and its performance can dictate the overall performance of many applications.

Chapter 16

References

* <https://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication>
* <http://mathforum.org/dr.math/faq/faq.peasant.html>
* <http://www.solipsys.co.uk/new/RussianPeasantMultiplication.html>
* Wikipedia,"Ordinary matrix product", [Online] Available:

http://en.wikipedia.org/wiki/Matrix\_multiplication - cite\_

note-0>

* T.H.Cormen., C.Leiserson.,andC.Stein.(2001). Introduction to Algorithms.U.S.A:The MIT Press.
* www.youtube.com/watch?v=xrUCL7tGKaI
* E.Horowitz., S.Sahni.,and S.Rajasekaran.(1996). Computer Algorithms/C++. U.S.A.: Compter Science Press.
* Sara Baase., and A.Gelder.(2000). Computer Algorithms:Introduction to Design and Analysis. U.S.A: Addison-Wesley.