TÜRKİYE CUMHURİYETİ

YILDIZ TEKNİK ÜNİVERSİTESİ

BİILGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

YAPISAL PROGRAMLAMAYA GİRİŞ

PROJE ÖDEVİ RAPORU

Öğrenci No: **20011037**

Öğrenci Adı Soyadı: **Mehmet Şadi Özcan**

Öğrenci E-Posta: **l1120037@std.yildiz.edu.tr**

Videonun YouTube adresi:
youtube.com/watch?v=wlnwRwj3zwI

Ders/Grup: BLM1012 / Grup 1

Ders Yürütücüsü

Doç. Dr. Fatih AMASYALI

Haziran, 2021

İÇİNDEKİLER

Ödevin İçeriği

Ödevde Strassen algoritmasının bir incelemesi yapılmıştır ve bu algoritma, matris çarpımında kullanılan 2 diğer algoritma ile karşılaştırılmıştır.

Strassen Algoritması

Strassen algoritması, matris çarpımı için kullanılan standart algoritmaya alternatif olarak Alman matematikçi Volker Strassen tarafından oluşturulmuş bir algoritmadır. İlk olarak 1969 yılında yayınlanmıştır. Strassen'in algoritması standart matris çarpım algoritmasına karşı ortaya atılan ilk algoritma olması bakımından önemlidir. Daha sonraki dönemlerde geliştirilecek bazı daha optimal algoritmaların çıkış noktası olarak kabul edilir.

Matris Çarpım Algoritmaları

1-) Standart Matris Çarpım Algoritması

```
for(i=0;i<satir;i++){
    for(j=0;j<sutun2;j++){
        toplam = 0;
        for(k=0;k<sutun;k++){
        toplam += matris1[i][k] * matris2[k][j];
        }
        carpim[i][j] = toplam;
        }
    }
```

Matrislerimizin her ikisinin de n x n boyutlarında olduğu varsayılırsa, program üç kez for döngüsünde n sayıda dönecek ve karmaşıklığımız basit bir şekilde $O(n^3)$ olarak hesaplanacaktır.

2-) Böl ve Fethet Algoritması

Böl ve fethet algoritmalarının temel amacı, ortada bulunan problemi en temel parçasına kadar indirgeyip problemin sonucunu bulmaktır.

Aşağıda söz edilen algoritma ve Strassen algoritmasında çarpımın yapılabilmesi için 2 temel gereklilik vardır.

1- Matrisler nxn'lik olmalıdır.
2- N 2'nin kuvveti olmalıdır.

Eğer matrisler şartları sağlamıyorsa 0 eklenerek bu hale dönüştürülürler.

$$\overset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \overset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \overset{C}{\begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}}$$

A, B ve C, N x N boyutlarında birer matristir.
a, b, c ve d, A matrisinin N/2 x N/2 boyutlarında alt matrisleridir.
e, f, g ve h, B matrisinin N/2 x N/2 boyutlarında alt matrisleridir.

Yukarıdaki şekilde görülen A ve B matrisleri eğer 2x2'lik matrislerse, 8 adet çarpma işlemi yapılacak, bu çarpma işlemleri sonucu elde edilen sayılar toplanıp sonuç matrisindeki ilgili yere yerleştirilecek ve böylece sonuç matrisi elde edilecektir.

A ve B matrisleri 2x2'den büyük matrislerse, (n/2) x (n/2) lik 4 ayrı matrise bölünürler. Daha sonrasında aynı toplama ve çarpma işlemleri **matris toplaması** ve **matris çarpması** şeklinde yapılarak sonuç matrisi elde edilir.

Temel böl ve fethet algoritmasında her adımda matris toplaması yapılmaktadır. NxN ‘lik iki matrisin toplanması **$O(n^2)$** karmaşıklıkla yapılır. Daha sonra 8 adet çarpma işlemi yapılır. Böylece bu algoritmanın karmaşıklığı:

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

$$T(n) = n^2 \sum_{i=0}^{log2(n)} 2^i$$

$$= n^2 \, \frac{2^{log2(n)+1}-1}{2-1} = n^2 \, \frac{2n-1}{1}$$

$$= 2n^3 - n^2$$

$$= O(n^3)$$

olarak hesaplanır. Görüldüğü üzere yine 8 çarpımın olduğu böl ve fethet algoritmasında karmaşıklıkta bir azalma olmamıştır.

3-) Strassen Algoritması

Strassen Algoritması, az önce ele aldığımız algoritma gibi özyinelemeli (rekörsif) bir algoritma olduğundan bu algoritmayı az önceki algoritmanın bir devamı olarak nitelendirebiliriz. Strassen algoritmasındaki temel fark, her bir çağrıda yapılan 8 çarpma işlemini 7’ye düşürerek problemin karmaşıklığını azaltmaktır.

$$\overset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \overset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \overset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

$$\begin{aligned} p1 &= a(f - h) & p5 &= (a + d)(e + h) \\ p2 &= (a + b)h & p6 &= (b - d)(g + h) \\ p3 &= (c + d)e & p7 &= (a - c)(e + f) \\ p4 &= d(g - e) & & \end{aligned}$$

Şekilde görüldüğü üzere her bir adımda 7 çarpma işlemini tutacak 7 farklı değişken tanımlanır ve bu değişkenlerin toplanıp çıkarılmasıyla sonuç matrisi elde edilir.

Matris 2x2 lik ise bu çarpım ve toplama işlemleri aritmetik olarak, 2x2'den büyük ise matris işlemleri olarak yapılmaktadır.

Strassen Algoritması'nın zaman karmaşıklığı:

$$\mathbf{T(n) = 7\,T(n/2) + n^2}$$

$$\mathbf{T(n)} = \boldsymbol{n^2 \sum_{i=0}^{log2(n)} (\tfrac{7}{4})^i}$$

$$= \boldsymbol{n^2 \frac{\left(\frac{7}{4}\right)^{log2(n)+1} - 1}{\left(\frac{7}{4}\right) - 1}}$$

$$= \boldsymbol{n^2 \frac{n^{log2\left(\frac{7}{4}\right)} \left(\frac{7}{4}\right) - 1}{\frac{3}{4}}}$$

$$= \boldsymbol{n^2 \frac{n^{0,80735} (\frac{7}{4}) - 1}{3/4}}$$

$$= \boldsymbol{\frac{n^{2,80735} (\frac{7}{4}) - n^2}{3/4}}$$

$$= O(n^{2.80735})$$

olarak hesaplanır.

Buna göre, karşılaştırdığımız 3 farklı matris çarpım algoritmasındaki zaman karmaşıklıkları sırasıyla:

Standart Çarpım: O(n^3)

Böl ve Fethet Yaklaşımı: O(n^3)

Strassen Algoritması: O($n^{2.80735}$)

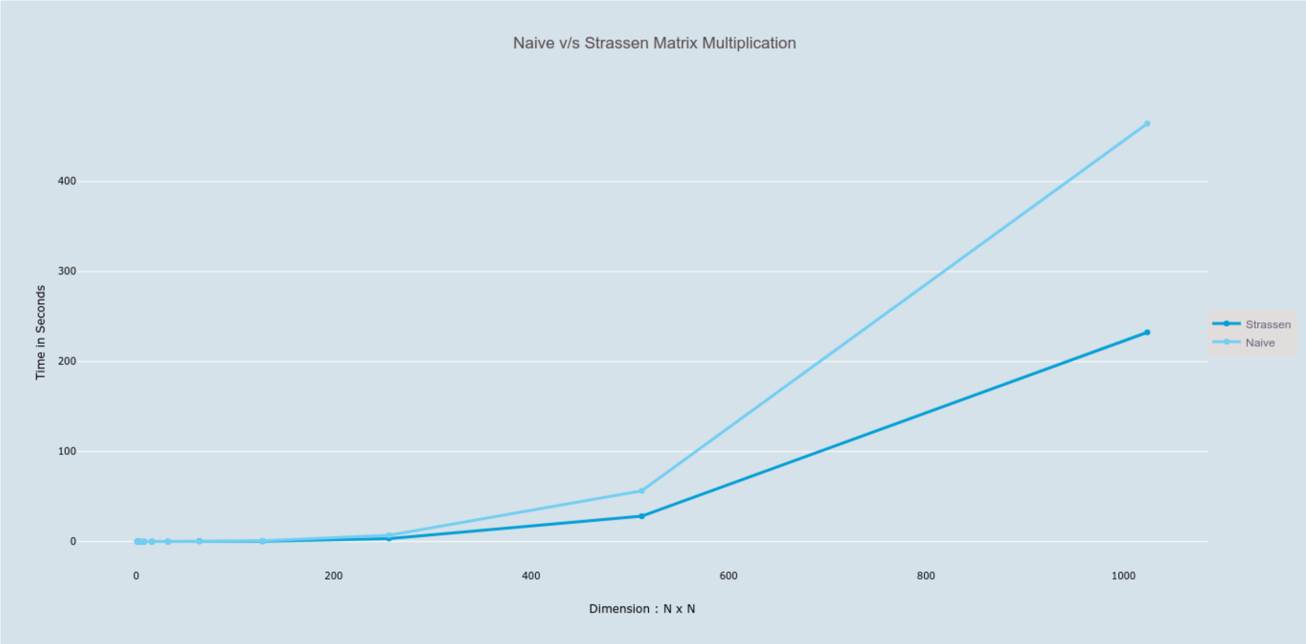
Strassen algoritmasındaki n^3'den $n^{2.80}$'e olan azalma çok büyük bir fark olarak gözükmeyebilir ancak fark üste olduğu için matris boyutu arttıkça aradaki fark artacaktır.

Strassen Algoritması'nın Dezavantajları

Strassen algoritması kağıt üzerinde daha avantajlı gözükmesine rağmen pratikte istenen sonucu her zaman verememektedir.

Bunun sebepleri:

- Rekörsif çağrıların bir yığılmaya neden olması ve fazla bellek tüketmesi
- Rekörsif çağrıların gecikmeye neden olması
- Karmaşıklık hesabında hesaba katılmayan işlemlerin (örneğin input matrisini 4 parçaya bölüp her bir parçayı yeni matrise yazmak) algoritmanın beklendiğinden yavaş çalışmasına sebep olması
  olarak sıralanabilir.



Standart Algoritma ile Strassen Algoritmasının belirli boyutlardaki matrislerde hız karşılaştırması.

Bununla birlikte şekilden de görüleceği üzere görece küçük matrislerdeki zaman farkı çok büyük olmadığından, küçük boyutlu matrislerin çarpımında Strassen algoritması yerine standart çarpım algoritması daha çok tercih edilir.

Günümüz bilgisayar yapılarında Strassen algoritmasının standart algoritmadan daha hızlı çalışmaya başlayacağı geçiş noktası implementasyona ve bilgisayar donanımına göre değişmektedir. Ancak son dönemlerde yapılan çalışmalarda bu geçiş noktasının giderek arttığı gözlenmektedir. 2010'da yapılan bir araştırmada, Strassen algoritmasının günümüz bilgisayar mimarilerinde, son derece optimize edilmiş standart yönteme göre matris boyutları 1000'i aşana kadar neredeyse hiçbir zaman daha yararlı olmadığı tespit edilmiştir.[kaynak] Daha yakın tarihli (2016) bir araştırma ise 512x512 boyutlu matrisler ve üzerinde %20 civarında bir fayda gözlemlemiştir.[kaynak]

Yöntemlerin Uygulandığı Programın C Kodu

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
void rastgele_matris(int satir, int sutun,int sutun2, int matris[satir][sutun],int
matris2[sutun][sutun2]);
void matrisi_al(int satir, int sutun, int matris[satir][sutun]);
void naive_method(int satir,int sutun, int sutun2, int matris1[satir][sutun],int
matris2[sutun][sutun2],int carpim[satir][sutun2]);
void add(int size, int a[size][size], int b[size][size],int c[size][size]);
void sub(int size,int a[size][size],int b[size][size],int c[size][size]);
void strassen(int n,int a[n][n],int b[n][n],int c[n][n]);
void divide_and_conquer(int n,int a[n][n],int b[n][n],int c[n][n]);

int main(){
	int satir, sutun, sutun2, n, i, j,secim;
	clock_t time1,time2,time3;
	printf("1.matrisin satir ve sutun sayisini sirasiyla giriniz: ");
	scanf("%d",&satir);
	scanf("%d",&sutun);
	printf("\n 2.matrisin sutun sayisini giriniz: ");
	scanf("%d",&sutun2);

	int matrix1[satir][sutun], matrix2[sutun][sutun2],carpim_naive[satir][sutun2];

	do{
	printf("\n 1-) Matrisi elle girmek istiyorum.\n 2-) Rastgele bir matris olustur.\n");
	scanf("%d",&secim);
	}while(secim!=1 && secim!=2);

	switch(secim){
		case 1:
			matrisi_al(satir,sutun,matrix1);
			printf("\n\n");
			matrisi_al(sutun,sutun2,matrix2);
			break;
		case 2:
			rastgele_matris(satir,sutun,sutun2,matrix1,matrix2);
	}
	printf("Standart Matris Carpim Algoritmasi ile Hesaplanan Sonuc:\n");

	time1 = clock();			// zamanı ölçüyoruz.
	naive_method(satir,sutun,sutun2,matrix1,matrix2,carpim_naive);
	time1 = clock() - time1;		// zamanı ölçüyoruz.

			for(i=0;i<satir;i++){		//sonuç matrisimizi yazdırıyoruz.
```

```
                    printf("\n");
                    for(j=0;j<sutun2;j++){
                        printf("%d \t",carpim_naive[i][j]);
                    } }
            printf("\n");

// strassen algoritması n 2'nin kuvveti olmak üzere n x n 'lik matrislerde çalıştığı için matrisler n x
n'lik değilse n x n'lik hale getirilecek.
            n = satir;
        if(sutun>satir){
                n=sutun;
                    }
            if(sutun2>n){
                n=sutun2;
            }
            n = pow(2, ceil(log(n)/log(2)));   // n'i 2'nin en yakın bir üst kuvvetine yuvarladık.
matrisimiz artık n x n ve n 2'nin kuvveti olduğu için strassen algoritması uygulanabilir hale geldi.

            int matris1[n][n],
matris2[n][n],carpim[n][n],matris1_2[n][n],matris2_2[n][n],carpim2[n][n];
            //strassen işleminde kullanmak üzere nxn'lik yeni matrisler oluşturduk ve başta
aldığımız matrisleri yeni matrise kopyalayacağız. matrisler nxn'lik değilse matrisi kopyaladıktan
sonra geri kalan boşlukları 0 ile dolduracağız.
              for(i=0;i<n;i++){
                for(j=0;j<n;j++){
                    if(i>satir-1 || j>sutun-1){
                    matris1[i][j] = 0;
                    matris1_2[i][j] = 0;}

                    else{
                    matris1[i][j] = matrix1[i][j];
                    matris1_2[i][j] = matrix1[i][j];
                    }
                }
                }
            for(i=0;i<n;i++){
                for(j=0;j<n;j++){
                    if(i>sutun-1 || j>sutun2-1){
                    matris2[i][j] = 0;
                    matris2_2[i][j] = 0;
                }
                else{
                    matris2[i][j] = matrix2[i][j];
                    matris2_2[i][j] = matrix2[i][j];
                    }
                }
                }

            time2 = clock(); //zaman ölçümünü başlatıp fonksiyonu çağırıyoruz. sonrasında
geçen zamanı hesaplıyoruz.
            strassen(n,matris1,matris2,carpim);
            time2 = clock() - time2;
```

```c
			time3 = clock();
			divide_and_conquer(n,matris1_2,matris2_2,carpim2);
			time3 = clock() - time3;

			printf("\nStrassen Algoritmasi ile Hesaplanan Sonuc:\n"); //sonucu yazdırıyoruz.
			for(i=0;i<satir;i++){
				printf("\n");
			for(j=0;j<sutun2;j++){
				printf("%d \t",carpim[i][j]);
			} }

			printf("\n\nDivide And Conquer Algoritmasi ile Hesaplanan Sonuc:\n"); //sonucu
yazdırıyoruz.
			for(i=0;i<satir;i++){
				printf("\n");
			for(j=0;j<sutun2;j++){
				printf("%d \t",carpim2[i][j]);
			} }
//yıldızlı gösterimi yapmak için zamanları birbirine göre oranlıyoruz.
		double oran = ((double)time2)/ ((double)time1);
		double oran2 = ((double)time3) / ((double)time1);
		printf ("\n\nProgramin calismasi sirasinda gecen sure: \n\n Standart Algoritma icin:  %.50lf\n
Strassen Algoritmasi icin: %.50lf\n Divide And Conquer Algoritmasi icin:
%.50lf",((double)time1)/CLOCKS_PER_SEC,((double)time2)/CLOCKS_PER_SEC,((double)time3)/CLOCK
S_PER_SEC);

		 if(time1!=0 && time2!=0 && time3!=0){ // program bazen standart algoritma süresini 0
ölçtüğü için oranımız sonsuz oluyor ve sonsuz yıldız atılıyor. bunu engellemek için grafik koşullu
olarak çizilecek.
		printf("\n\n Standart algoritma suresi:            **********");
		printf("\n Strassen algoritma suresi:            " );
		// zaman(strassen) / zaman(standart) oranını az önce bulmuştuk. burada kesirden
kurtulmak için 10 ile çarpıyoruz. Daha sonra bu orana göre fonksiyon süreleri arasındaki farkı
yıldızlarla göstereceğiz.
		oran *= 10; oran2*= 10;
		for(i=0;i<=oran;i++){
			printf("*");
		 }
		printf("\n Divide and Conquer algoritma suresi:   ");
		for(i=0;i<=oran2;i++){
			printf("*");
		} }
		return 0;
}
//rand fonksiyonunu kullanarak istenen boyutta 2 adet rastgele matris üretmek için kullanılan
fonksiyon.
void rastgele_matris(int satir, int sutun,int sutun2, int matris[satir][sutun],int
matris2[sutun][sutun2]){
		int i,j;

		srand(time(0));
```

```c
	for(i=0;i<satir;i++){
		for(j=0;j<sutun;j++){
			matris[i][j] = rand() % 99 + 1;
		}
	}
	for(i=0;i<sutun;i++){
		for(j=0;j<sutun2;j++){
			matris2[i][j] = rand() % 99 + 1;
		}
	}
	printf("Olusturulan matris:\n");
	for(i=0;i<satir;i++){
		printf("\n");
		for(j=0;j<sutun;j++){
			printf("%d \t",matris[i][j]);
		}
	}
	printf("\n\nOlusturulan matris:\n");
	for(i=0;i<sutun;i++){
		printf("\n");
		for(j=0;j<sutun2;j++){
			printf("%d \t",matris2[i][j]);
		}
	}
	printf("\n\n");
	return;
}
//kullanıcı matrisi kendi girmek isterse istenen boyutlarda matrisi alan fonksiyon.
void matrisi_al(int satir, int sutun, int matris[satir][sutun]){
	int i,j;
	for(i=0; i<satir; i++){
      for(j=0; j<sutun; j++){
         printf("[%d][%d]) Sayiyi giriniz: ", i+1, j+1);
         scanf("%d", &matris[i][j]);
      }
}
	printf("\nOlusturulan matris:\n");
	for(i=0;i<satir;i++){
		printf("\n");
		for(j=0;j<sutun;j++){
			printf("%d \t",matris[i][j]);
		}
	}
	printf("\n\n");

	return;
}
// standart matris çarpım algoritması.
void naive_method(int satir,int sutun, int sutun2, int matris1[satir][sutun],int
matris2[sutun][sutun2],int carpim[satir][sutun2]){
	int i,j,k,p;
	float toplam=0;
```

```
        for(i=0;i<satir;i++){
                for(j=0;j<sutun2;j++){
                        toplam = 0;
                        for(k=0;k<sutun;k++){
                        toplam += matris1[i][k] * matris2[k][j];
                        }
                        carpim[i][j] = toplam;
                        }
                }

return;
}
//size x size boyutlarındaki iki matrisi toplayıp c matrisine yazan fonksiyon.
void add(int size, int a[size][size], int b[size][size],int c[size][size]){
  int i,j;
  for(i=0;i<size;i++){
        for(j=0;j<size;j++){
      c[i][j] = a[i][j] + b[i][j];
                }
  }
  return;
}
//size x size boyutlarındaki a matrisinden b matrisini çıkarıp c matrisine yazan fonksiyon.
void sub(int size,int a[size][size],int b[size][size],int c[size][size]){
  int i,j;
  for(i=0;i<size;i++){
         for(j=0;j<size;j++){
              c[i][j]= a[i][j] - b[i][j];
         }
    }
    return;
}
//DIVIDE AND CONQUER FONKSİYONU
void divide_and_conquer(int n,int a[n][n],int b[n][n],int c[n][n]){
        int i,j,m=n/2;    //her aşamanın başında nxn 'lik matristen n/2 x n/2 boyutlarında 4 yeni
matris oluşturulur.
        int     c00[m][m],c01[m][m],c10[m][m],c11[m][m];

        if(n==2){         // rekörsif fonksiyonun base case'i. //yapılması gereken 8 çarpım işlemini
yapıp bunları farklı değişkenlere kaydediyoruz.
                int ae = a[0][0] * b[0][0];
                int bg = a[0][1] * b[1][0];
                int af = a[0][0] * b[0][1];
                int bh = a[0][1] * b[1][1];
                int ce = a[1][0] * b[0][0];
                int dg = a[1][1] * b[1][0];
                int cf = a[1][0] * b[0][1];
                int dh = a[1][1] * b[1][1];
```

```
//yaptığımız çarpma işlemlerinde bulduğumuz değişkenleri, algoritmada belirtildiği gibi toplama ve çıkarma işlemlerine tabii tutarak bulduğumuz sonuçları sonuç matrisine kaydediyoruz.
                    c[0][0]= ae+bg;
                    c[0][1]= af+bh;
                    c[1][0]= ce+dg;
                    c[1][1]= cf+dh;
          }
          else{     //fonkiyonun recursive case'i.
                    int a00[m][m], a01[m][m], a10[m][m], a11[m][m],       b00[m][m], b01[m][m], b10[m][m], b11[m][m]; //baştaki matrislerin 4 parçasını bölüp kaydetmek üzere kullanılacak matrisler.
                    int ae1[m][m],bg1[m][m],af1[m][m],bh1[m][m],ce1[m][m],dg1[m][m],cf1[m][m],dh1[m][m],sum1[m][m],sum2[m][m],sub1[m][m],sub2[m][m];
                    for(i=0;i<m;i++){
                              for(j=0;j<m;j++){         //baştaki matrisleri 4 parçaya bölüp yeni oluşturduğumuz matrislere kaydediyoruz.
                                        a00[i][j] = a[i][j];
                                        b00[i][j] = b[i][j];
                                        a01[i][j] = a[i][j+m];
                                        b01[i][j] = b[i][j+m];
                                        a10[i][j] = a[i+m][j];
                                        b10[i][j] = b[i+m][j];
                                        a11[i][j] = a[i+m][j+m];
                                        b11[i][j] = b[i+m][j+m];
                              }
                    }
//base case'de sayılarlar yaptığımız çarpım işlemlerini burada matrislerle yapıyoruz.
                    divide_and_conquer(m,a00,b00,ae1);
                    divide_and_conquer(m,a01,b10,bg1);
                    divide_and_conquer(m,a00,b01,af1);
                    divide_and_conquer(m,a01,b11,bh1);
                    divide_and_conquer(m,a10,b00,ce1);
                    divide_and_conquer(m,a11,b10,dg1);
                    divide_and_conquer(m,a10,b01,cf1);
                    divide_and_conquer(m,a11,b11,dh1);

                    add(m,ae1,bg1,c00);
                    add(m,af1,bh1,c01);
                    add(m,ce1,dg1,c10);
                    add(m,cf1,dh1,c11);


                    for(i=0;i<m;i++){         //bulduğumuz c00, c01, c10 ve c11 matrislerini sonuç matrisinde birleştiriyoruz.
                              for(j=0;j<m;j++){
                                        c[i][j] = c00[i][j];
                                        c[i][j+m] = c01[i][j];
                                        c[i+m][j] = c10[i][j];
                                        c[i+m][j+m] = c11[i][j];
                              }
                    }
```

```
}
	return;
}
//STRASSEN FONKSİYONU
void strassen(int n,int a[n][n],int b[n][n],int c[n][n]){
	int i,j,m=n/2;	//her aşamanın başında nxn 'lik matristen n/2 x n/2 boyutlarında 4 yeni matris oluşturulur.
	int	c00[m][m],c01[m][m],c10[m][m],c11[m][m];

	if(n==2){		// rekörsif fonksiyonun base case'i.
		int P = (a[0][0]+a[1][1]) * (b[0][0]+b[1][1]);
		int Q = (a[1][0]+a[1][1]) * b[0][0];
		int R = a[0][0] * (b[0][1]-b[1][1]);		//strassen algoritmasında yapılması gereken 7 adet çarpım işlemini yapıp bunları farklı değişkenlere kaydediyoruz.
		int S = a[1][1] * (b[1][0]-b[0][0]);
		int T = (a[0][0]+a[0][1]) * b[1][1];
		int U = (a[1][0]-a[0][0]) * (b[0][0]+b[0][1]);
		int V = (a[0][1]-a[1][1]) * (b[1][0]+b[1][1]);
//yaptığımız çarpma işlemlerinde bulduğumuz değişkenleri, algoritmada belirtildiği gibi toplama ve çıkarma işlemlerine tabii tutarak bulduğumuz sonuçları sonuç matrisine kaydediyoruz.
		c[0][0]= P+S-T+V;
		c[0][1]= R+T;
		c[1][0]= Q+S;
		c[1][1]= P+R-Q+U;
	}
	else{	//fonkiyonun recursive case'i.
		int a00[m][m], a01[m][m], a10[m][m], a11[m][m],	b00[m][m], b01[m][m], b10[m][m], b11[m][m]; //baştaki matrislerin 4 parçasını bölüp kaydetmek üzere kullanılacak matrisler.
		int p[m][m],q[m][m],r[m][m],s[m][m],t[m][m],u[m][m],v[m][m],sum1[m][m],sum2[m][m],sub1[m][m],sub2[m][m];
		for(i=0;i<m;i++){
			for(j=0;j<m;j++){		//baştaki matrisleri 4 parçaya bölüp yeni oluşturduğumuz matrislere kaydediyoruz.
				a00[i][j] = a[i][j];
				b00[i][j] = b[i][j];
				a01[i][j] = a[i][j+m];
				b01[i][j] = b[i][j+m];
				a10[i][j] = a[i+m][j];
				b10[i][j] = b[i+m][j];
				a11[i][j] = a[i+m][j+m];
				b11[i][j] = b[i+m][j+m];
			}
		}
// BASE CASE'DE SAYILARLA YAPTIĞIMIZ İŞLEMLERİ BURADA MATRİSLERLE YAPIYORUZ.

		add(m,a00,a11,sum1);	//p = (a00 + a11) * (b00+b11)
		add(m,b00,b11,sum2);
		strassen(m,sum1,sum2,p);

		add(m,a10,a11,sum1);	//q = (a10+a11) * b00
```

```
			strassen(m,sum1,b00,q);

			sub(m,b01,b11,sub1);   //r = (b01-b11) * a00
			strassen(m,a00,sub1,r);

			sub(m,b10,b00,sub2);   //s = (b10-b00) * a11
			strassen(m,a11,sub2,s);

			add(m,a00,a01,sum1);  //t = (a00+a01) * b11
			strassen(m,sum1,b11,t);

			sub(m,a10,a00,sub1); //u = (a10-a00)   * (b00+b01)
			add(m,b00,b01,sum1);
			strassen(m,sub1,sum1,u);

			sub(m,a01,a11,sub2); //v = (a01-a11) * (b10+b11)
			add(m,b10,b11,sum2);
			strassen(m,sub2,sum2,v);

			add(m,p,s,c00);				// c00 = p + s - t + v
			add(m,c00,v,c00);
			sub(m,c00,t,c00);

			add(m,r,t,c01);				// c01 = r + t

			add(m,q,s,c10);				// c10 = q + s

			add(m,p,r,c11);				// c11 = p + r + u - q
			add(m,c11,u,c11);
			sub(m,c11,q,c11);

			for(i=0;i<m;i++){			//bulduğumuz c00, c01, c10 ve c11 matrislerini
sonuç matrisinde birleştiriyoruz.

				for(j=0;j<m;j++){
					c[i][j] = c00[i][j];
					c[i][j+m] = c01[i][j];
					c[i+m][j] = c10[i][j];
					c[i+m][j+m] = c11[i][j];
				}
			}
}
		return;
}
```

Programın Çalışmasına İlişkin Ekran Görüntüleri

```
1.matrisin satir ve sutun sayisini sirasiyla giriniz: 4
4

 2.matrisin sutun sayisini giriniz: 4

 1-) Matrisi elle girmek istiyorum.
 2-) Rastgele bir matris olustur.
2
Olusturulan matris:

79      17      90      51
34      58      22      7
50      71      8       2
42      96      27      38

Olusturulan matris:

78      85      87      71
22      54      14      31
83      64      17      87
32      55      5       9

Standart Matris Carpim Algoritmasi ile Hesaplanan Sonuc:

15638   16198   8896    14425
5978    7815    4179    6189
6190    8706    5490    6465
8845    12572   5647    8649

Strassen Algoritmasi ile Hesaplanan Sonuc:

15638   16198   8896    14425
5978    7815    4179    6189
6190    8706    5490    6465
8845    12572   5647    8649

Divide And Conquer Algoritmasi ile Hesaplanan Sonuc:

15638   16198   8896    14425
5978    7815    4179    6189
6190    8706    5490    6465
8845    12572   5647    8649

Programin calismasi sirasinda gecen sure:

 Standart Algoritma icin:  0.000003000000000000000007600257229123386082392244134
 Strassen Algoritmasi icin: 0.000015000000000000000038001286145616930411961220670
 Divide And Conquer Algoritmasi icin: 0.000007999999999999999637984894607090069484911509 79

 Standart algoritma suresi:            **********
 Strassen algoritma suresi:            **************************************************
 Divide and Conquer algoritma suresi:  ***************************
```

```
1.matrisin satir ve sutun sayisini sirasiyla giriniz: 2
2

 2.matrisin sutun sayisini giriniz: 2

 1-) Matrisi elle girmek istiyorum.
 2-) Rastgele bir matris olustur.
1
[1][1]) Sayiyi giriniz: 2
[1][2]) Sayiyi giriniz: 6
[2][1]) Sayiyi giriniz: 7
[2][2]) Sayiyi giriniz: 8

Olusturulan matris:

2       6
7       8



[1][1]) Sayiyi giriniz: 2
[1][2]) Sayiyi giriniz: 1
[2][1]) Sayiyi giriniz: 8
[2][2]) Sayiyi giriniz: 9

Olusturulan matris:

2       1
8       9

Standart Matris Carpim Algoritmasi ile Hesaplanan Sonuc:

52      56
78      79

Strassen Algoritmasi ile Hesaplanan Sonuc:

52      56
78      79

Divide And Conquer Algoritmasi ile Hesaplanan Sonuc:

52      56
78      79

Programin calismasi sirasinda gecen sure:

 Standart Algoritma icin:  0.00000399999999999999981899244730354503474245575489
 Strassen Algoritmasi icin: 0.00000300000000000000007600257229123386082392244134
 Divide And Conquer Algoritmasi icin: 0.00000300000000000000007600257229123386082392244134

 Standart algoritma suresi:             **********
 Strassen algoritma suresi:             ********
 Divide and Conquer algoritma suresi:   ********
```

```
1.matrisin satir ve sutun sayisini sirasiyla giriniz: 4
7

 2.matrisin sutun sayisini giriniz: 1

 1-) Matrisi elle girmek istiyorum.
 2-) Rastgele bir matris olustur.
2
Olusturulan matris:

51      58      89      71      48      87      79
50      82      34      98      86      23      91
82      51      1       28      60      51      42
73      74      89      97      62      11      22

Olusturulan matris:

84
7
54
35
65
43
4

Standart Matris Carpim Algoritmasi ile Hesaplanan Sonuc:

19158
16983
14540
19442

Strassen Algoritmasi ile Hesaplanan Sonuc:

19158
16983
14540
19442

Divide And Conquer Algoritmasi ile Hesaplanan Sonuc:

19158
16983
14540
19442

Programin calismasi sirasinda gecen sure:

 Standart Algoritma icin:  0.000003000000000000000007600257229123386082392244134
 Strassen Algoritmasi icin: 0.000036000000000000000091203086749480632988706929609
 Divide And Conquer Algoritmasi icin: 0.000021999999999999999427974933796647860617667902226

 Standart algoritma suresi:            **********
 Strassen algoritma suresi:            ************************************************************************************
 Divide and Conquer algoritma suresi:  *******************************************************************************
```

Programda Yapılan Zaman Analizi

Bu bölümde, programda sırasıyla 2x2, 4x4, 16x16, 32x32, 64x64, 128x128, 256x256 boyutunda matrislerin çarpımı sonucu ortaya çıkan zaman verilerinin değerlendirmesi yapılacak.

Matris Boyutları	Farklı Denemelerde Çıkan **Zaman(Strassen)/Zaman(Standart)** Oranları										Ortalama Oran
2x2	1,00	1,33	0,67	1,33	1,00	1,00	0,66	1,33	1,33	1,00	1,06
4x4	5,00	6,33	4,33	7,00	7,33	5,75	5,33	4,33	5,66	4,66	5,57
8x8	5,42	7,66	6,16	5,66	6,33	7,16	5,20	4,89	6,28	7,14	6,19
16x16	6,67	6,37	6,75	6,96	6,85	6,33	6,81	4,89	4,65	3,56	5,99
32x32	7,21	5,09	4,11	3,77	5,90	6,04	6,53	6,39	6,21	5,14	5,64
64x64	5,74	6,11	6,21	5,08	4,30	4,66	5,35	5,78	5,37	4,80	5,34
128x128	4,29	4,07	4,42	5,03	4,52	4,71	4,60	4,38	3,79	4,45	4,42
256x256	2,40	2,22	4,29	2,67	2,49	2,89	2,35	2,62	2,37	2,64	2,69

Tabloda her matris boyutu için 10 kez program çalıştırılmış ve Strassen algoritması ile standart algoritma kullanıldığında geçen zamanlar birbirine oranlanmıştır. Tablodaki verilere göre, Strassen algoritmasının 2x2'lik matrislerin çarpımında standart algoritmayla ortalama olarak aynı sürede tamamlandığı, matris boyutu büyüdükçe Zaman(Strassen) / Zaman(Standart) oranının 5-6 civarlarına çıktığı, daha sonra yavaş bir azalma gösterdiği ve 256x256'lık matrislerde 2,69 değerine indiği gözlenmektedir.

512x512 ve daha büyük matrislerde çarpım birkaç bilgisayarda denenmiş olmasına rağmen yazdığım program belirli bir noktadan sonra hata vererek kapandığından ötürü kesin bir sonuca varılamamaktadır. Ancak Strassen Algoritması'nın Dezavantajları bölümündeki araştırmaların doğru olduğu kabul edilirse, en son 2,69 olarak hesaplanan değerin giderek azalıp belirli bir noktada 1'in altına ineceği düşünülebilir.

Raporu Hazırlarken Yararlanılan Kaynaklar

https://en.wikipedia.org/wiki/Strassen_algorithm

https://mlhtnc.github.io/strassen-algorithm.html

https://medium.com/swlh/strassens-matrix-multiplication-algorithm-936f42c2b344

https://www.onlinegdb.com/online_c_compiler

http://hilite.me/