Shubhan Singh

SE-Comps B/Batch C

2022300118

DAA Experiment 3

Aim – Experiment based on divide and conquer approach. To implement Min-Max algorithm using divide and conquer approach and Strassen’s matrix multiplication algorithm.

Details – Divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems.

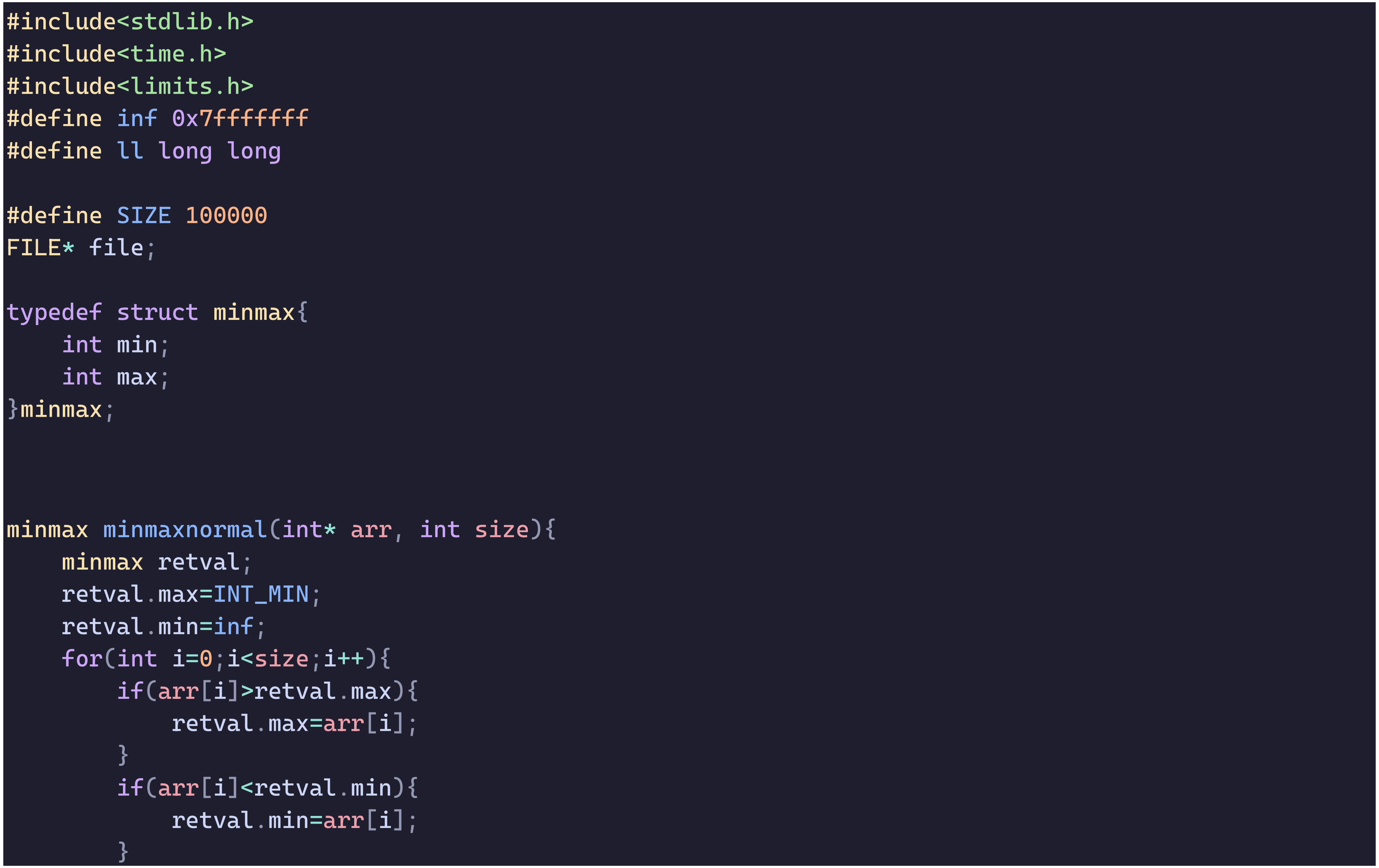
Input –Each student have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100,200,300,...,100000 integer numbers to the min-max algorithm. Also use these numbers as inputs to the matrices to be multiplied by the naïve method and Strassen’s algorithm.

Source code(C language):

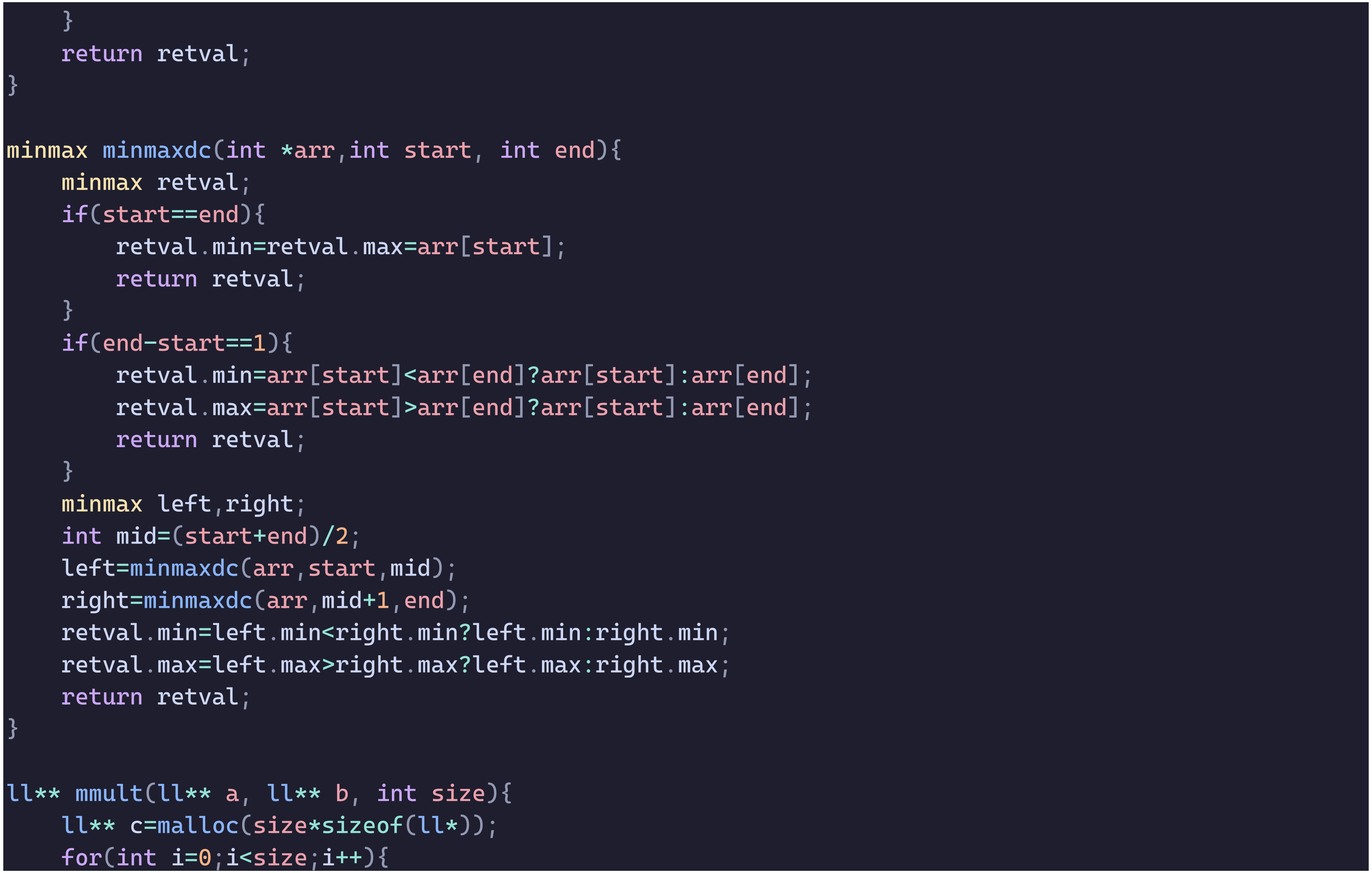
- I have printed the output in a csv file so that the data can be easily imported into excel.

(pseudocode given later)

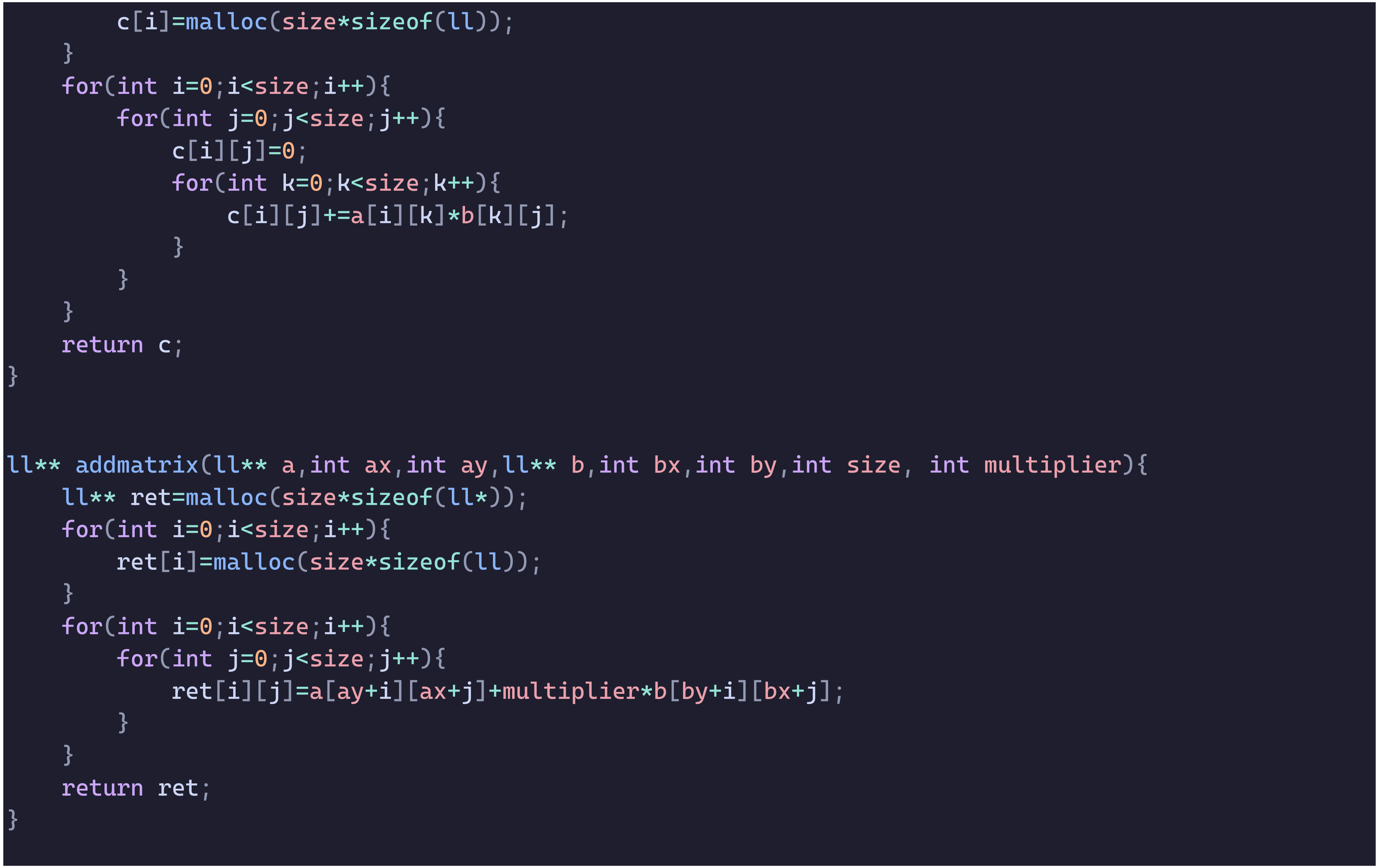
```
#include<stdio.h>
```



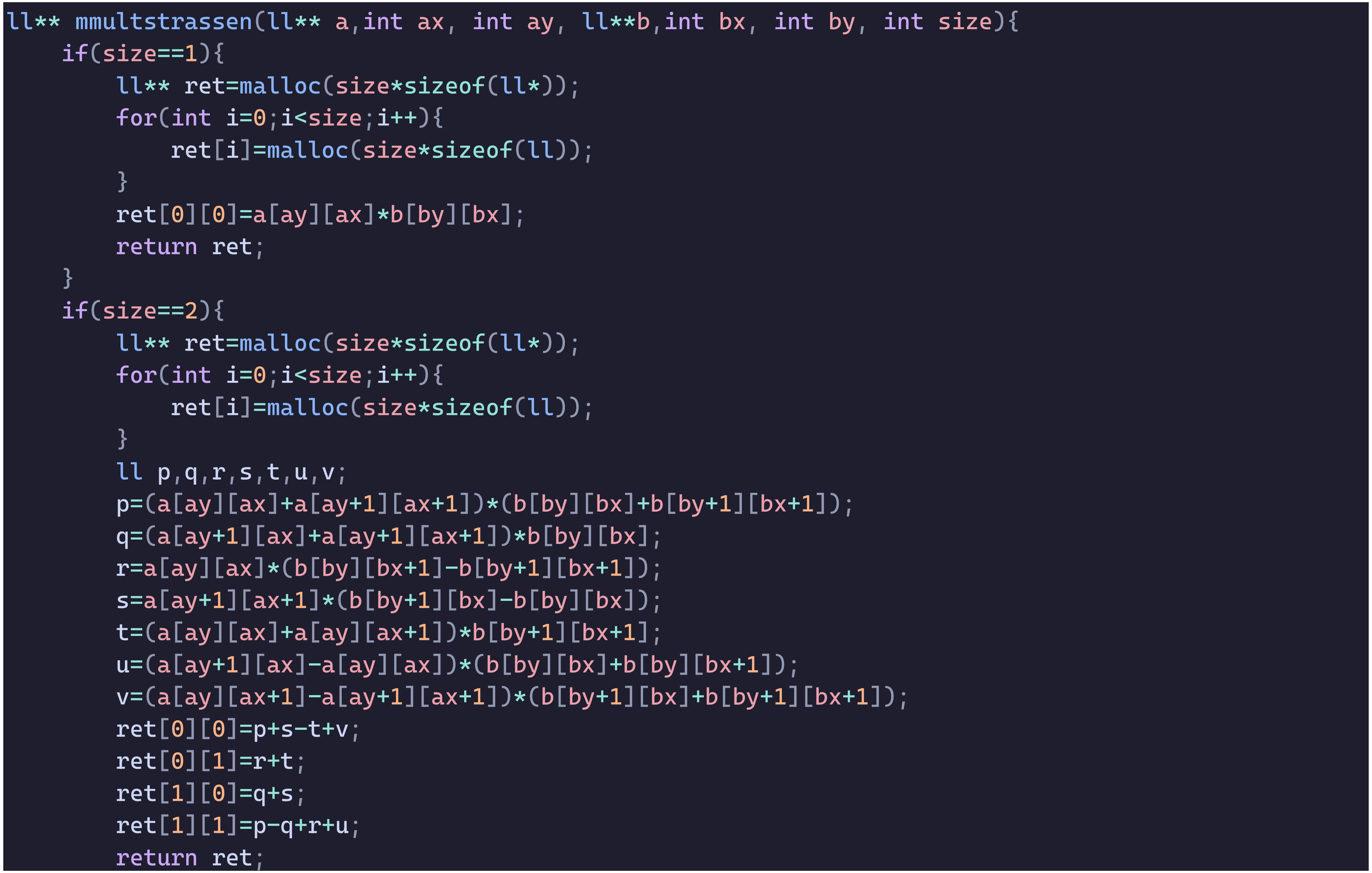
```c
#include<stdlib.h>
#include<time.h>
#include<limits.h>
#define inf 0x7fffffff
#define ll long long

#define SIZE 100000
FILE* file;

typedef struct minmax{
    int min;
    int max;
}minmax;



minmax minmaxnormal(int* arr, int size){
    minmax retval;
    retval.max=INT_MIN;
    retval.min=inf;
    for(int i=0;i<size;i++){
        if(arr[i]>retval.max){
            retval.max=arr[i];
        }
        if(arr[i]<retval.min){
            retval.min=arr[i];
        }
```



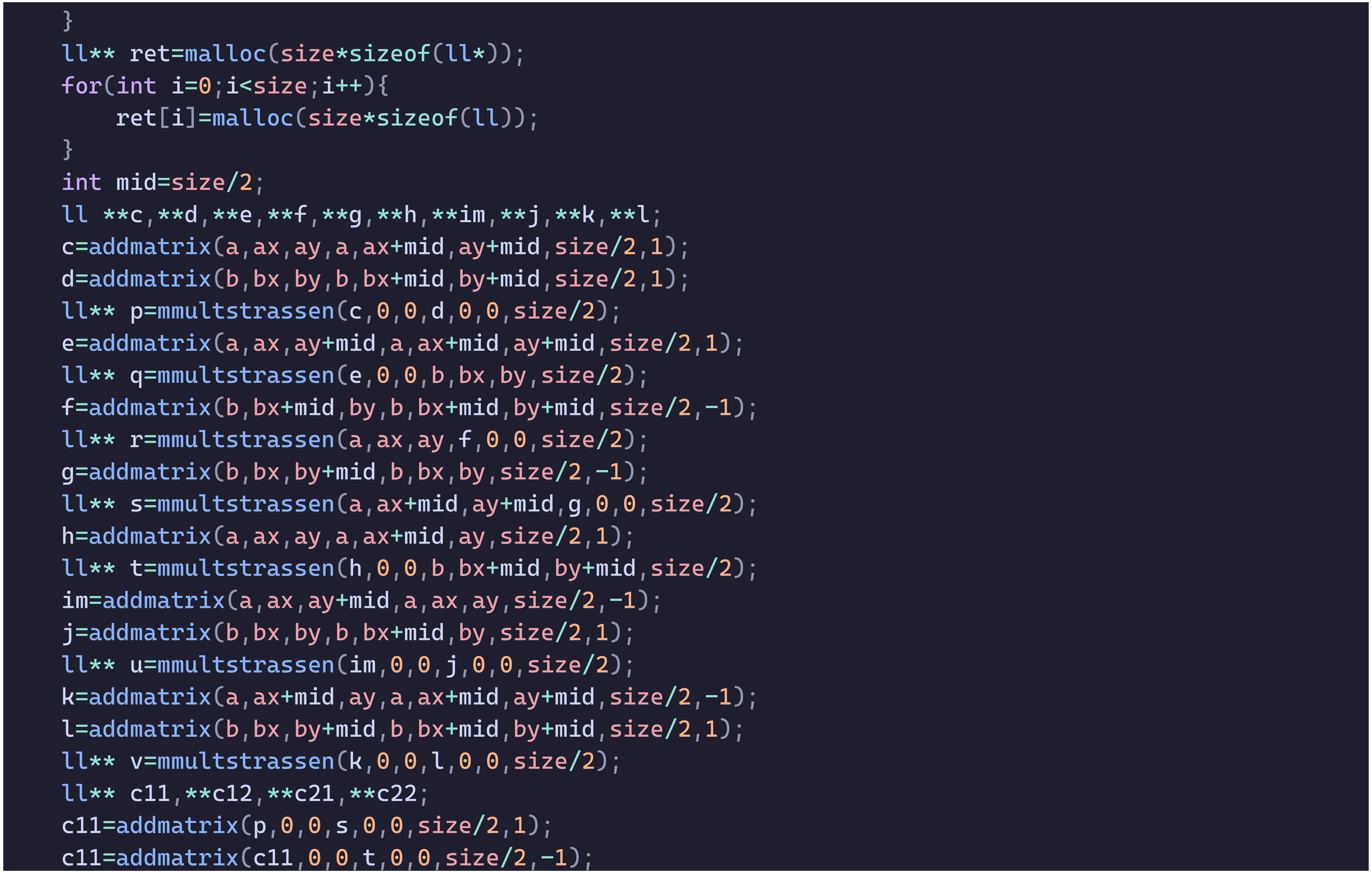
```
    }
    return retval;
}

minmax minmaxdc(int *arr,int start, int end){
    minmax retval;
    if(start==end){
        retval.min=retval.max=arr[start];
        return retval;
    }
    if(end-start==1){
        retval.min=arr[start]<arr[end]?arr[start]:arr[end];
        retval.max=arr[start]>arr[end]?arr[start]:arr[end];
        return retval;
    }
    minmax left,right;
    int mid=(start+end)/2;
    left=minmaxdc(arr,start,mid);
    right=minmaxdc(arr,mid+1,end);
    retval.min=left.min<right.min?left.min:right.min;
    retval.max=left.max>right.max?left.max:right.max;
    return retval;
}

ll** mmult(ll** a, ll** b, int size){
    ll** c=malloc(size*sizeof(ll*));
    for(int i=0;i<size;i++){
```



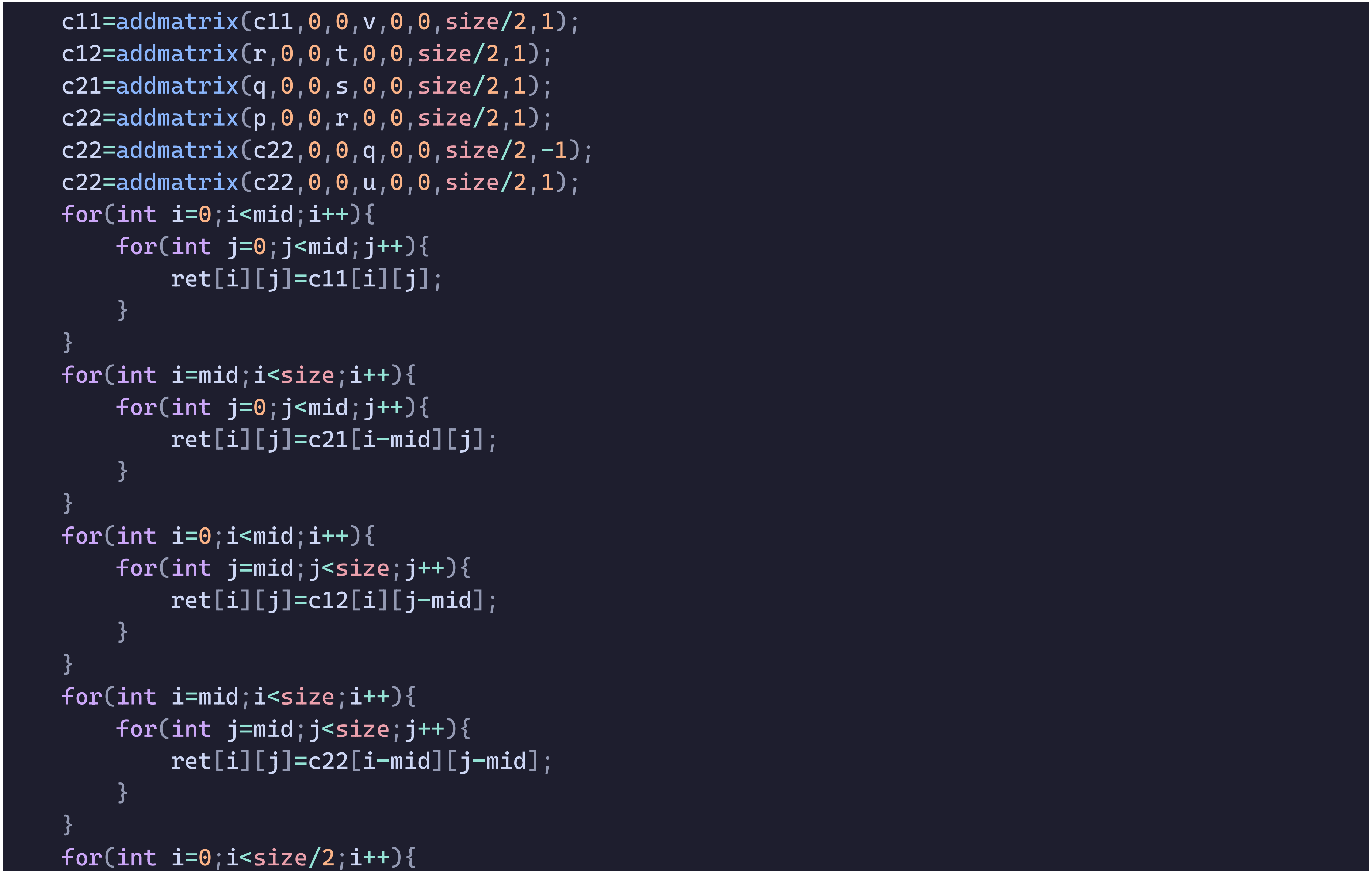
```
        c[i]=malloc(size*sizeof(ll));
    }
    for(int i=0;i<size;i++){
        for(int j=0;j<size;j++){
            c[i][j]=0;
            for(int k=0;k<size;k++){
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
    return c;
}


ll** addmatrix(ll** a,int ax,int ay,ll** b,int bx,int by,int size, int multiplier){
    ll** ret=malloc(size*sizeof(ll*));
    for(int i=0;i<size;i++){
        ret[i]=malloc(size*sizeof(ll));
    }
    for(int i=0;i<size;i++){
        for(int j=0;j<size;j++){
            ret[i][j]=a[ay+i][ax+j]+multiplier*b[by+i][bx+j];
        }
    }
    return ret;
}
```



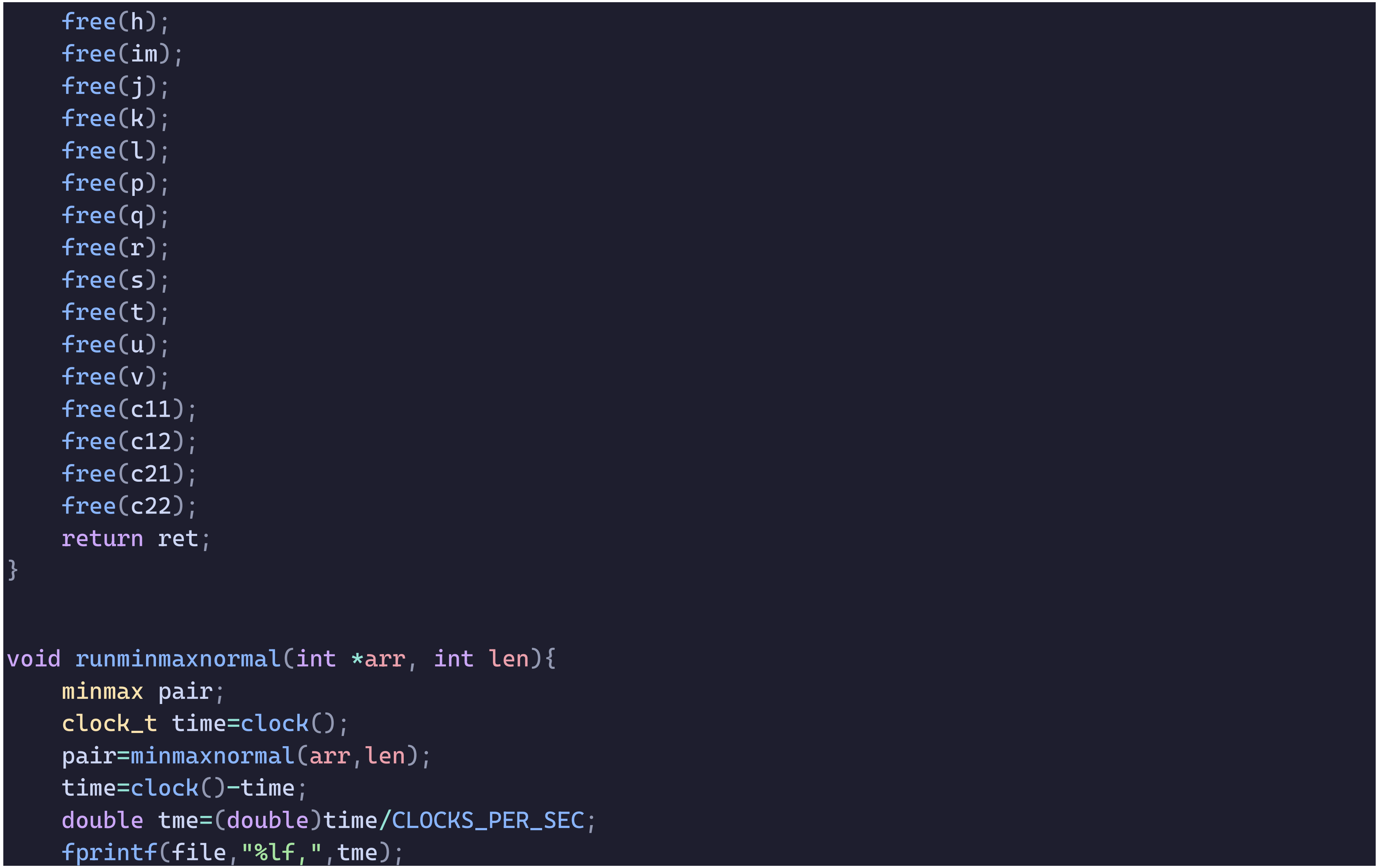
```
ll** mmultstrassen(ll** a,int ax, int ay, ll**b,int bx, int by, int size){
    if(size==1){
        ll** ret=malloc(size*sizeof(ll*));
        for(int i=0;i<size;i++){
            ret[i]=malloc(size*sizeof(ll));
        }
        ret[0][0]=a[ay][ax]*b[by][bx];
        return ret;
    }
    if(size==2){
        ll** ret=malloc(size*sizeof(ll*));
        for(int i=0;i<size;i++){
            ret[i]=malloc(size*sizeof(ll));
        }
        ll p,q,r,s,t,u,v;
        p=(a[ay][ax]+a[ay+1][ax+1])*(b[by][bx]+b[by+1][bx+1]);
        q=(a[ay+1][ax]+a[ay+1][ax+1])*b[by][bx];
        r=a[ay][ax]*(b[by][bx+1]-b[by+1][bx+1]);
        s=a[ay+1][ax+1]*(b[by+1][bx]-b[by][bx]);
        t=(a[ay][ax]+a[ay][ax+1])*b[by+1][bx+1];
        u=(a[ay+1][ax]-a[ay][ax])*(b[by][bx]+b[by][bx+1]);
        v=(a[ay][ax+1]-a[ay+1][ax+1])*(b[by+1][bx]+b[by+1][bx+1]);
        ret[0][0]=p+s-t+v;
        ret[0][1]=r+t;
        ret[1][0]=q+s;
        ret[1][1]=p-q+r+u;
        return ret;
```



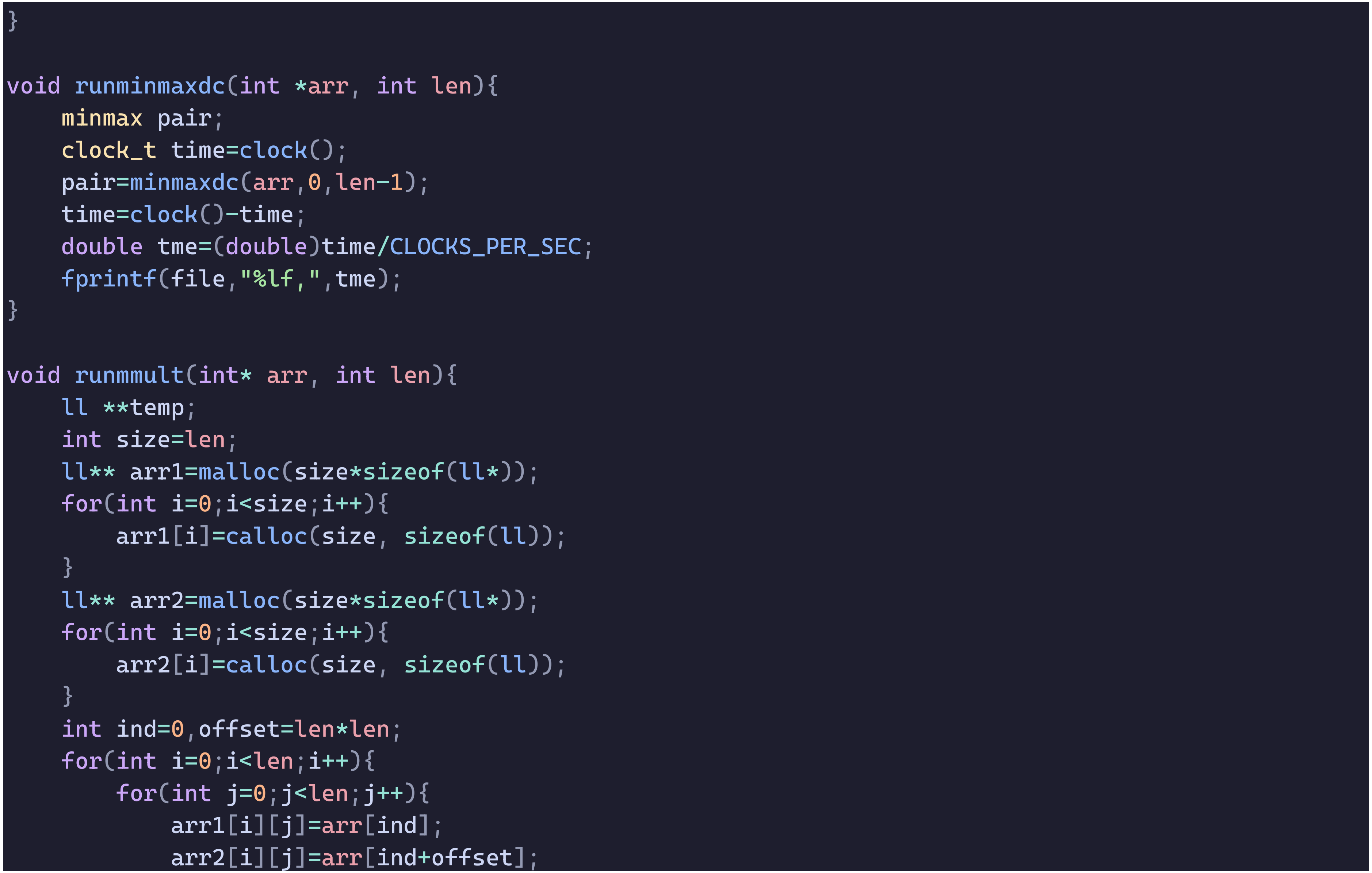
```
    }
    ll** ret=malloc(size*sizeof(ll*));
    for(int i=0;i<size;i++){
        ret[i]=malloc(size*sizeof(ll));
    }
    int mid=size/2;
    ll **c,**d,**e,**f,**g,**h,**im,**j,**k,**l;
    c=addmatrix(a,ax,ay,a,ax+mid,ay+mid,size/2,1);
    d=addmatrix(b,bx,by,b,bx+mid,by+mid,size/2,1);
    ll** p=mmultstrassen(c,0,0,d,0,0,size/2);
    e=addmatrix(a,ax,ay+mid,a,ax+mid,ay+mid,size/2,1);
    ll** q=mmultstrassen(e,0,0,b,bx,by,size/2);
    f=addmatrix(b,bx+mid,by,b,bx+mid,by+mid,size/2,-1);
    ll** r=mmultstrassen(a,ax,ay,f,0,0,size/2);
    g=addmatrix(b,bx,by+mid,b,bx,by,size/2,-1);
    ll** s=mmultstrassen(a,ax+mid,ay+mid,g,0,0,size/2);
    h=addmatrix(a,ax,ay,a,ax+mid,ay,size/2,1);
    ll** t=mmultstrassen(h,0,0,b,bx+mid,by+mid,size/2);
    im=addmatrix(a,ax,ay+mid,a,ax,ay,size/2,-1);
    j=addmatrix(b,bx,by,b,bx+mid,by,size/2,1);
    ll** u=mmultstrassen(im,0,0,j,0,0,size/2);
    k=addmatrix(a,ax+mid,ay,a,ax+mid,ay+mid,size/2,-1);
    l=addmatrix(b,bx,by+mid,b,bx+mid,by+mid,size/2,1);
    ll** v=mmultstrassen(k,0,0,l,0,0,size/2);
    ll** c11,**c12,**c21,**c22;
    c11=addmatrix(p,0,0,s,0,0,size/2,1);
    c11=addmatrix(c11,0,0,t,0,0,size/2,-1);
```



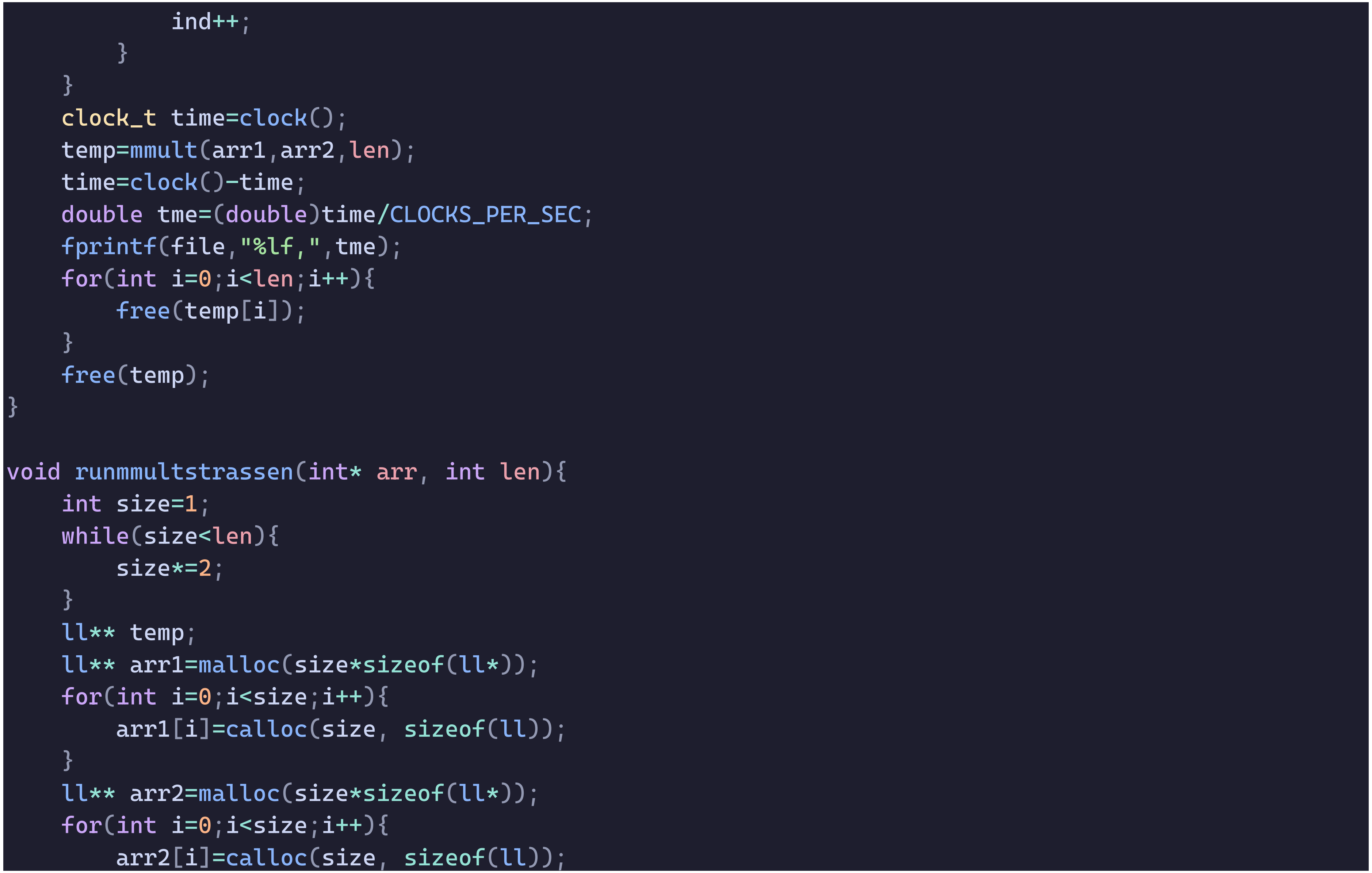
```
    c11=addmatrix(c11,0,0,v,0,0,size/2,1);
    c12=addmatrix(r,0,0,t,0,0,size/2,1);
    c21=addmatrix(q,0,0,s,0,0,size/2,1);
    c22=addmatrix(p,0,0,r,0,0,size/2,1);
    c22=addmatrix(c22,0,0,q,0,0,size/2,-1);
    c22=addmatrix(c22,0,0,u,0,0,size/2,1);
    for(int i=0;i<mid;i++){
        for(int j=0;j<mid;j++){
            ret[i][j]=c11[i][j];
        }
    }
    for(int i=mid;i<size;i++){
        for(int j=0;j<mid;j++){
            ret[i][j]=c21[i-mid][j];
        }
    }
    for(int i=0;i<mid;i++){
        for(int j=mid;j<size;j++){
            ret[i][j]=c12[i][j-mid];
        }
    }
    for(int i=mid;i<size;i++){
        for(int j=mid;j<size;j++){
            ret[i][j]=c22[i-mid][j-mid];
        }
    }
    for(int i=0;i<size/2;i++){
        free(c[i]);
        free(d[i]);
        free(e[i]);
        free(f[i]);
        free(g[i]);
        free(h[i]);
        free(im[i]);
        free(j[i]);
        free(k[i]);
        free(l[i]);
        free(p[i]);
        free(q[i]);
        free(r[i]);
        free(s[i]);
        free(t[i]);
        free(u[i]);
        free(v[i]);
        free(c11[i]);
        free(c12[i]);
        free(c21[i]);
        free(c22[i]);
    }
    free(c);
    free(d);
    free(e);
    free(f);
    free(g);
```



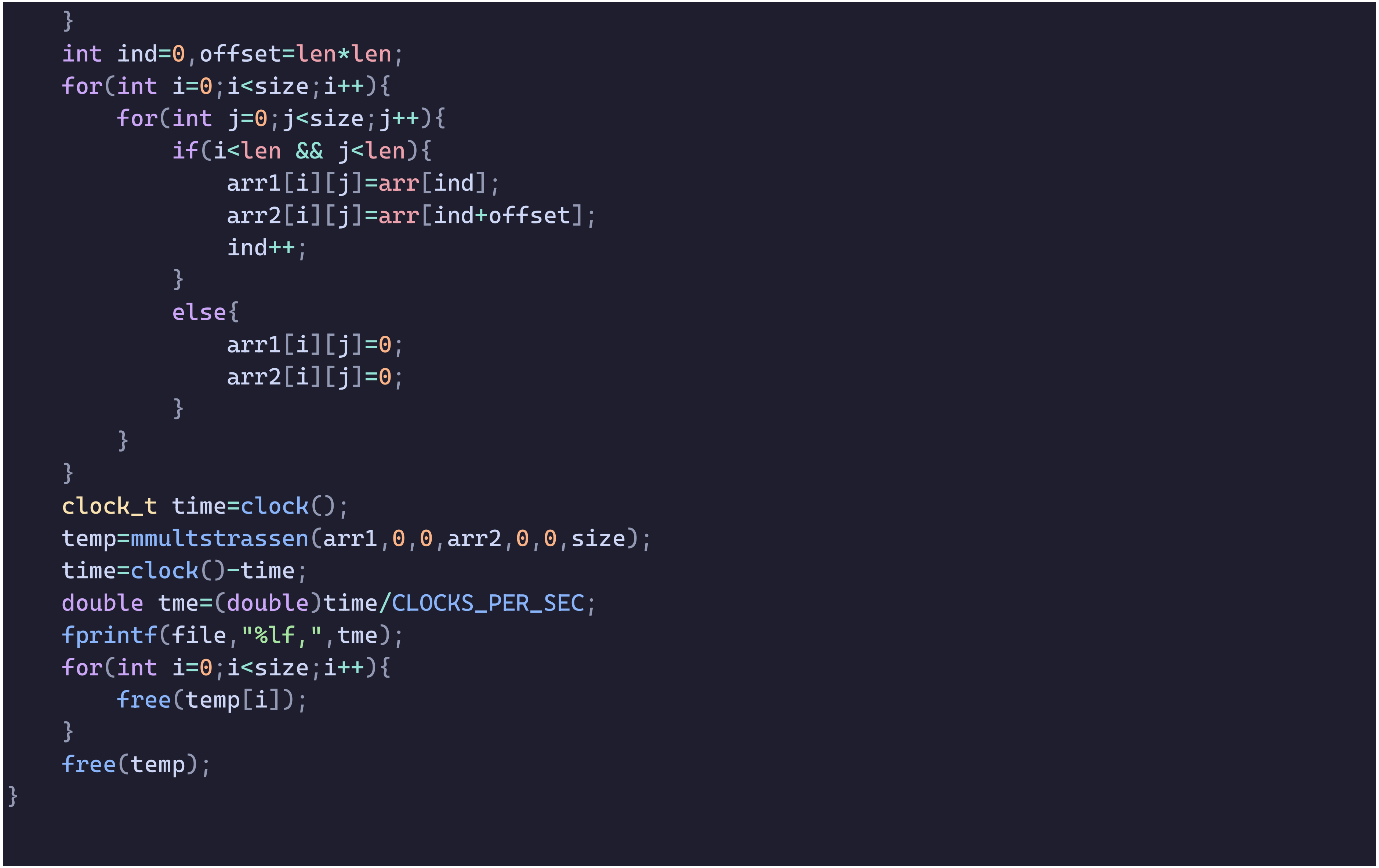
```
    free(h);
    free(im);
    free(j);
    free(k);
    free(l);
    free(p);
    free(q);
    free(r);
    free(s);
    free(t);
    free(u);
    free(v);
    free(c11);
    free(c12);
    free(c21);
    free(c22);
    return ret;
}


void runminmaxnormal(int *arr, int len){
    minmax pair;
    clock_t time=clock();
    pair=minmaxnormal(arr,len);
    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file,"%lf,",tme);
```



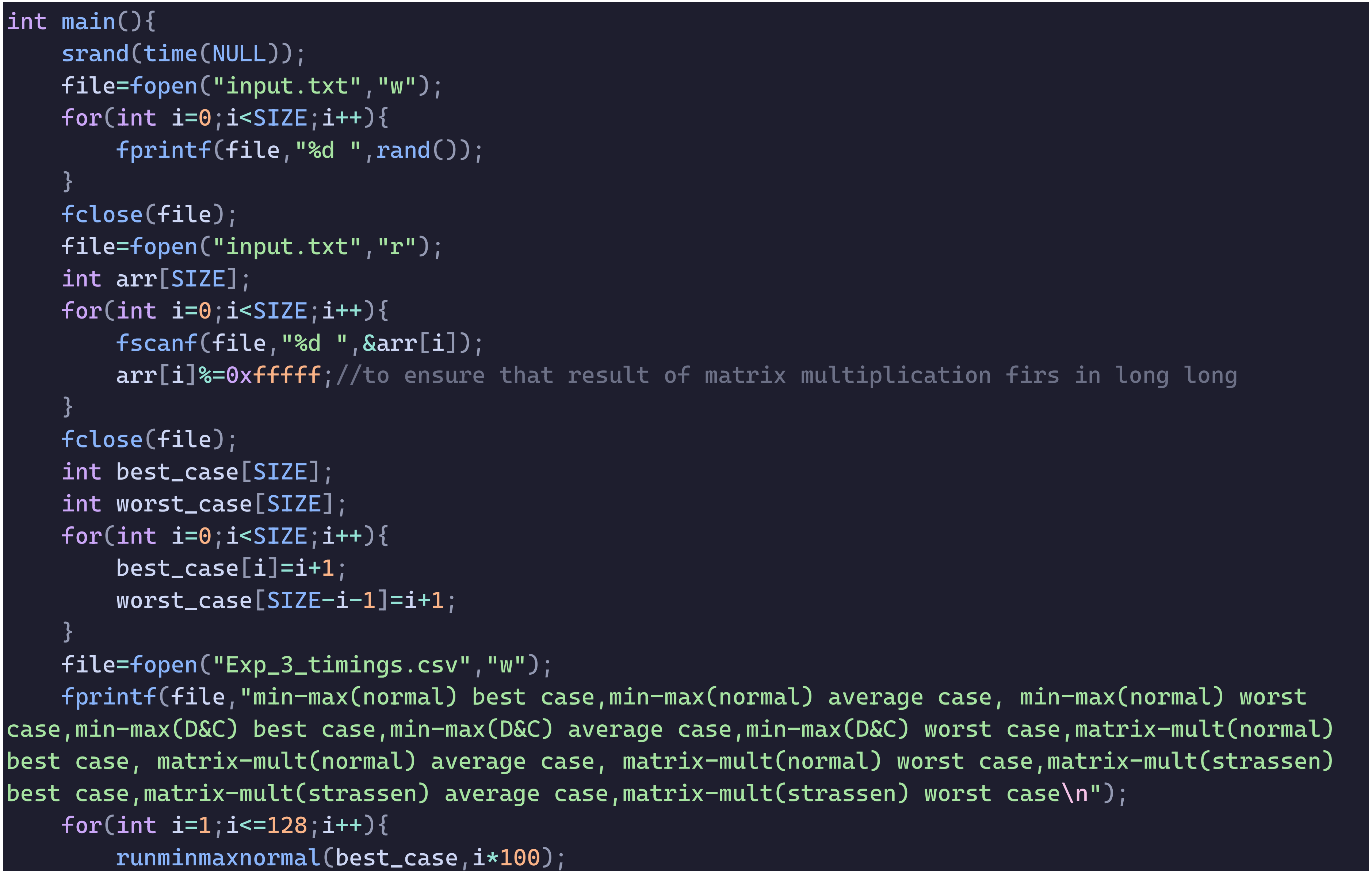
```
}

void runminmaxdc(int *arr, int len){
    minmax pair;
    clock_t time=clock();
    pair=minmaxdc(arr,0,len-1);
    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file,"%lf,",tme);
}

void runmmult(int* arr, int len){
    ll **temp;
    int size=len;
    ll** arr1=malloc(size*sizeof(ll*));
    for(int i=0;i<size;i++){
        arr1[i]=calloc(size, sizeof(ll));
    }
    ll** arr2=malloc(size*sizeof(ll*));
    for(int i=0;i<size;i++){
        arr2[i]=calloc(size, sizeof(ll));
    }
    int ind=0,offset=len*len;
    for(int i=0;i<len;i++){
        for(int j=0;j<len;j++){
            arr1[i][j]=arr[ind];
            arr2[i][j]=arr[ind+offset];
```



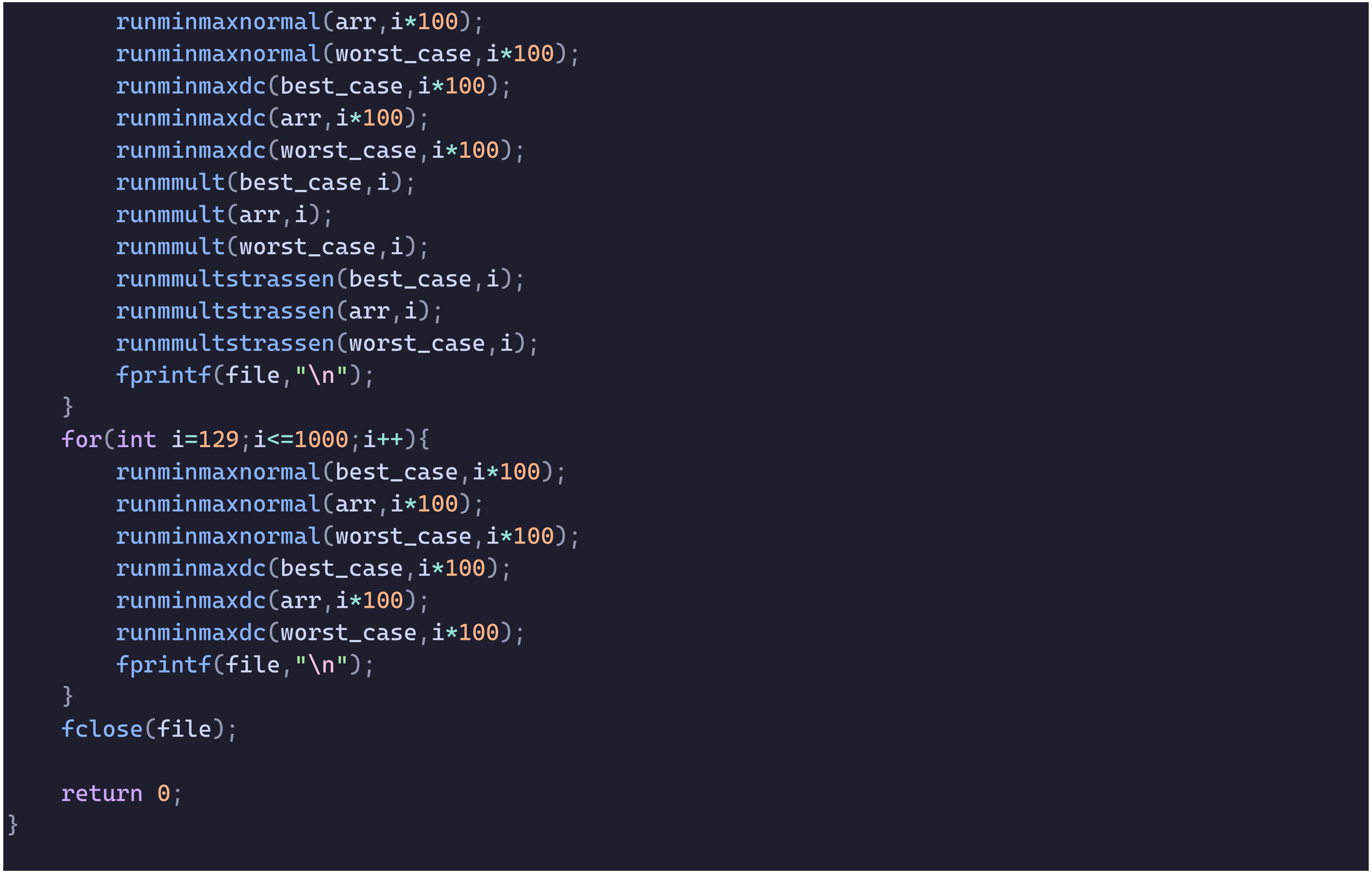
```
            ind++;
        }
    }
    clock_t time=clock();
    temp=mmult(arr1,arr2,len);
    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file,"%lf,",tme);
    for(int i=0;i<len;i++){
        free(temp[i]);
    }
    free(temp);
}

void runmmultstrassen(int* arr, int len){
    int size=1;
    while(size<len){
        size*=2;
    }
    ll** temp;
    ll** arr1=malloc(size*sizeof(ll*));
    for(int i=0;i<size;i++){
        arr1[i]=calloc(size, sizeof(ll));
    }
    ll** arr2=malloc(size*sizeof(ll*));
    for(int i=0;i<size;i++){
        arr2[i]=calloc(size, sizeof(ll));
```



```
    }
    int ind=0,offset=len*len;
    for(int i=0;i<size;i++){
        for(int j=0;j<size;j++){
            if(i<len && j<len){
                arr1[i][j]=arr[ind];
                arr2[i][j]=arr[ind+offset];
                ind++;
            }
            else{
                arr1[i][j]=0;
                arr2[i][j]=0;
            }
        }
    }
    clock_t time=clock();
    temp=mmultstrassen(arr1,0,0,arr2,0,0,size);
    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file,"%lf,",tme);
    for(int i=0;i<size;i++){
        free(temp[i]);
    }
    free(temp);
}
```



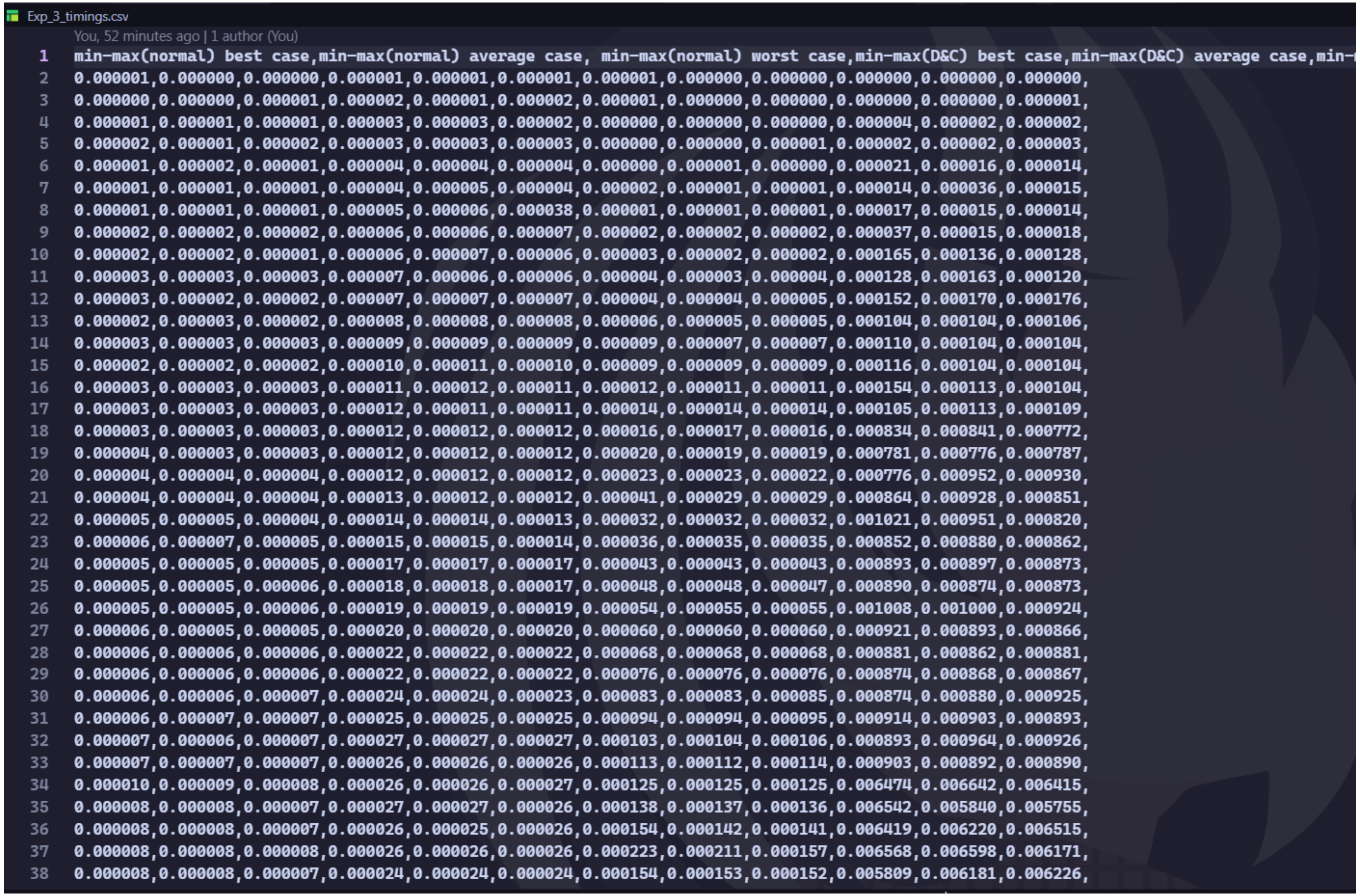
```c
int main(){
    srand(time(NULL));
    file=fopen("input.txt","w");
    for(int i=0;i<SIZE;i++){
        fprintf(file,"%d ",rand());
    }
    fclose(file);
    file=fopen("input.txt","r");
    int arr[SIZE];
    for(int i=0;i<SIZE;i++){
        fscanf(file,"%d ",&arr[i]);
        arr[i]%=0xfffff;//to ensure that result of matrix multiplication firs in long long
    }
    fclose(file);
    int best_case[SIZE];
    int worst_case[SIZE];
    for(int i=0;i<SIZE;i++){
        best_case[i]=i+1;
        worst_case[SIZE-i-1]=i+1;
    }
    file=fopen("Exp_3_timings.csv","w");
    fprintf(file,"min-max(normal) best case,min-max(normal) average case, min-max(normal) worst case,min-max(D&C) best case,min-max(D&C) average case,min-max(D&C) worst case,matrix-mult(normal) best case, matrix-mult(normal) average case, matrix-mult(normal) worst case,matrix-mult(strassen) best case,matrix-mult(strassen) average case,matrix-mult(strassen) worst case\n");
    for(int i=1;i<=128;i++){
        runminmaxnormal(best_case,i*100);
```



```
        runminmaxnormal(arr,i*100);
        runminmaxnormal(worst_case,i*100);
        runminmaxdc(best_case,i*100);
        runminmaxdc(arr,i*100);
        runminmaxdc(worst_case,i*100);
        runmmult(best_case,i);
        runmmult(arr,i);
        runmmult(worst_case,i);
        runmmultstrassen(best_case,i);
        runmmultstrassen(arr,i);
        runmmultstrassen(worst_case,i);
        fprintf(file,"\n");
    }
    for(int i=129;i<=1000;i++){
        runminmaxnormal(best_case,i*100);
        runminmaxnormal(arr,i*100);
        runminmaxnormal(worst_case,i*100);
        runminmaxdc(best_case,i*100);
        runminmaxdc(arr,i*100);
        runminmaxdc(worst_case,i*100);
        fprintf(file,"\n");
    }
    fclose(file);

    return 0;
}
```

Output:

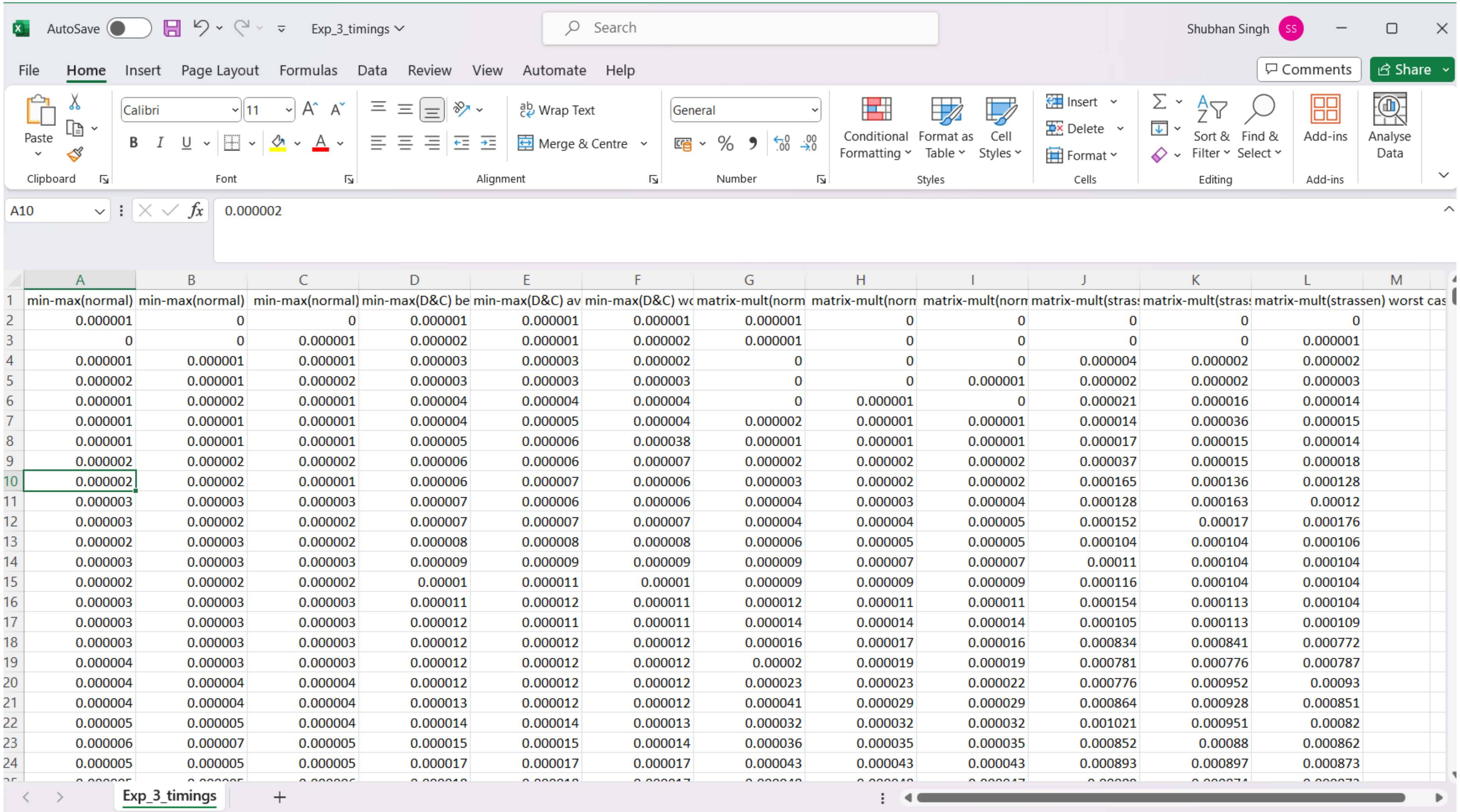
File created



Exp_3_timings.csv

You, 52 minutes ago | 1 author (You)

```
min-max(normal) best case,min-max(normal) average case, min-max(normal) worst case,min-max(D&C) best case,min-max(D&C) average case,min-
0.000001,0.000000,0.000000,0.000001,0.000001,0.000001,0.000001,0.000000,0.000000,0.000000,0.000000,0.000000,
0.000000,0.000000,0.000001,0.000002,0.000001,0.000002,0.000001,0.000000,0.000000,0.000000,0.000000,0.000001,
0.000001,0.000001,0.000001,0.000003,0.000003,0.000002,0.000000,0.000000,0.000000,0.000004,0.000002,0.000002,
0.000002,0.000001,0.000002,0.000003,0.000003,0.000003,0.000000,0.000000,0.000001,0.000002,0.000002,0.000003,
0.000001,0.000002,0.000001,0.000004,0.000004,0.000004,0.000000,0.000001,0.000000,0.000021,0.000016,0.000014,
0.000001,0.000001,0.000001,0.000004,0.000005,0.000004,0.000002,0.000001,0.000001,0.000014,0.000036,0.000015,
0.000001,0.000001,0.000001,0.000005,0.000006,0.000038,0.000001,0.000001,0.000001,0.000017,0.000015,0.000014,
0.000002,0.000002,0.000002,0.000006,0.000006,0.000007,0.000002,0.000002,0.000002,0.000037,0.000015,0.000018,
0.000002,0.000002,0.000001,0.000006,0.000007,0.000006,0.000003,0.000002,0.000002,0.000165,0.000136,0.000128,
0.000003,0.000003,0.000003,0.000007,0.000006,0.000006,0.000004,0.000003,0.000004,0.000128,0.000163,0.000120,
0.000003,0.000002,0.000002,0.000007,0.000007,0.000007,0.000004,0.000004,0.000005,0.000152,0.000170,0.000176,
0.000002,0.000003,0.000002,0.000008,0.000008,0.000008,0.000006,0.000005,0.000005,0.000104,0.000104,0.000106,
0.000003,0.000003,0.000003,0.000009,0.000009,0.000009,0.000009,0.000007,0.000007,0.000110,0.000104,0.000104,
0.000002,0.000002,0.000002,0.000010,0.000011,0.000010,0.000009,0.000009,0.000009,0.000116,0.000104,0.000104,
0.000003,0.000003,0.000003,0.000011,0.000012,0.000011,0.000012,0.000011,0.000011,0.000154,0.000113,0.000104,
0.000003,0.000003,0.000003,0.000012,0.000011,0.000011,0.000014,0.000014,0.000014,0.000105,0.000113,0.000109,
0.000003,0.000003,0.000003,0.000012,0.000012,0.000012,0.000016,0.000017,0.000016,0.000834,0.000841,0.000772,
0.000004,0.000003,0.000003,0.000012,0.000012,0.000012,0.000020,0.000019,0.000019,0.000781,0.000776,0.000787,
0.000004,0.000004,0.000004,0.000012,0.000012,0.000012,0.000023,0.000023,0.000022,0.000776,0.000952,0.000930,
0.000004,0.000004,0.000004,0.000013,0.000012,0.000012,0.000041,0.000029,0.000029,0.000864,0.000928,0.000851,
0.000005,0.000005,0.000004,0.000014,0.000014,0.000013,0.000032,0.000032,0.000032,0.001021,0.000951,0.000820,
0.000006,0.000007,0.000005,0.000015,0.000015,0.000014,0.000036,0.000035,0.000035,0.000852,0.000880,0.000862,
0.000005,0.000005,0.000005,0.000017,0.000017,0.000017,0.000043,0.000043,0.000043,0.000893,0.000897,0.000873,
0.000005,0.000005,0.000006,0.000018,0.000018,0.000017,0.000048,0.000048,0.000047,0.000890,0.000874,0.000873,
0.000005,0.000005,0.000006,0.000019,0.000019,0.000019,0.000054,0.000055,0.000055,0.001008,0.001000,0.000924,
0.000006,0.000005,0.000005,0.000020,0.000020,0.000020,0.000060,0.000060,0.000060,0.000921,0.000893,0.000866,
0.000006,0.000006,0.000006,0.000022,0.000022,0.000022,0.000068,0.000068,0.000068,0.000881,0.000862,0.000881,
0.000006,0.000006,0.000006,0.000022,0.000022,0.000022,0.000076,0.000076,0.000076,0.000874,0.000868,0.000867,
0.000006,0.000006,0.000007,0.000024,0.000024,0.000023,0.000083,0.000083,0.000085,0.000874,0.000880,0.000925,
0.000006,0.000007,0.000007,0.000025,0.000025,0.000025,0.000094,0.000094,0.000095,0.000914,0.000903,0.000893,
0.000007,0.000006,0.000007,0.000027,0.000027,0.000027,0.000103,0.000104,0.000106,0.000893,0.000964,0.000926,
0.000007,0.000007,0.000007,0.000026,0.000026,0.000026,0.000113,0.000112,0.000114,0.000903,0.000892,0.000890,
0.000010,0.000009,0.000008,0.000026,0.000026,0.000027,0.000125,0.000125,0.000125,0.006474,0.006642,0.006415,
0.000008,0.000008,0.000007,0.000027,0.000027,0.000026,0.000138,0.000137,0.000136,0.006542,0.005840,0.005755,
0.000008,0.000008,0.000007,0.000026,0.000025,0.000026,0.000154,0.000142,0.000141,0.006419,0.006220,0.006515,
0.000008,0.000008,0.000008,0.000026,0.000026,0.000026,0.000223,0.000211,0.000157,0.006568,0.006598,0.006171,
0.000008,0.000008,0.000007,0.000024,0.000024,0.000024,0.000154,0.000153,0.000152,0.005809,0.006181,0.006226,
```

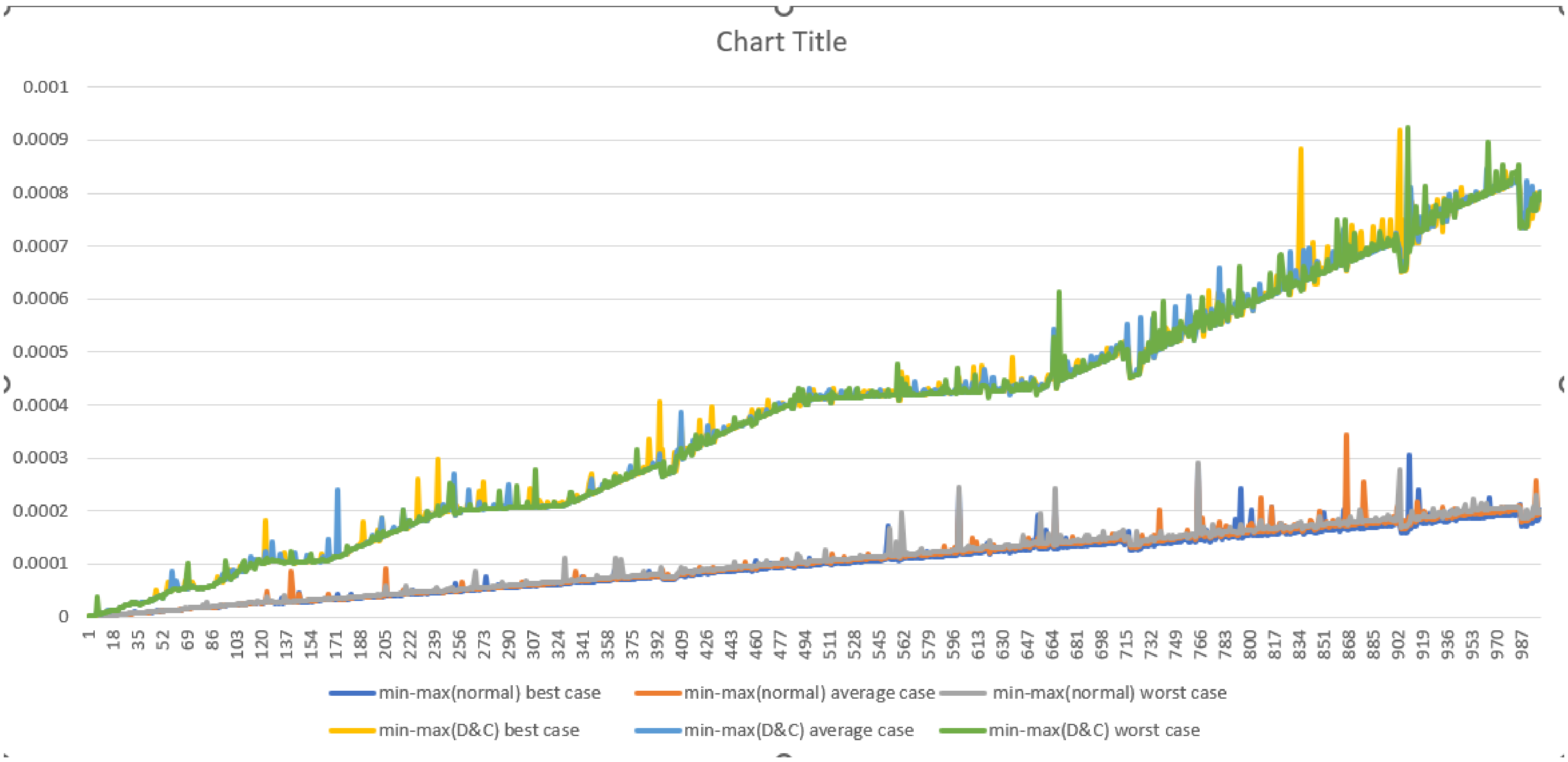


A10 0.000002

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	min-max(normal)	min-max(normal)	min-max(normal)	min-max(D&C) be	min-max(D&C) av	min-max(D&C) wc	matrix-mult(norm	matrix-mult(norm	matrix-mult(norm	matrix-mult(strass	matrix-mult(strass	matrix-mult(strassen) worst cas	
2	0.000001	0	0	0.000001	0.000001	0.000001	0.000001	0	0	0	0	0	
3	0	0	0.000001	0.000002	0.000001	0.000002	0.000001	0	0	0	0	0.000001	
4	0.000001	0.000001	0.000001	0.000003	0.000003	0.000002	0	0	0	0.000004	0.000002	0.000002	
5	0.000002	0.000001	0.000002	0.000003	0.000003	0.000003	0	0	0.000001	0.000002	0.000002	0.000003	
6	0.000001	0.000002	0.000001	0.000004	0.000004	0.000004	0	0.000001	0	0.000021	0.000016	0.000014	
7	0.000001	0.000001	0.000001	0.000004	0.000005	0.000004	0.000002	0.000001	0.000001	0.000014	0.000036	0.000015	
8	0.000001	0.000001	0.000001	0.000005	0.000006	0.000038	0.000001	0.000001	0.000001	0.000017	0.000015	0.000014	
9	0.000002	0.000002	0.000002	0.000006	0.000006	0.000007	0.000002	0.000002	0.000002	0.000037	0.000015	0.000018	
10	0.000002	0.000002	0.000001	0.000006	0.000007	0.000006	0.000003	0.000002	0.000002	0.000165	0.000136	0.000128	
11	0.000003	0.000003	0.000003	0.000007	0.000006	0.000006	0.000004	0.000003	0.000004	0.000128	0.000163	0.00012	
12	0.000003	0.000002	0.000002	0.000007	0.000007	0.000007	0.000004	0.000004	0.000005	0.000152	0.00017	0.000176	
13	0.000002	0.000003	0.000002	0.000008	0.000008	0.000008	0.000006	0.000005	0.000005	0.000104	0.000104	0.000106	
14	0.000003	0.000003	0.000003	0.000009	0.000009	0.000009	0.000009	0.000007	0.000007	0.00011	0.000104	0.000104	
15	0.000002	0.000002	0.000002	0.00001	0.000011	0.00001	0.000009	0.000009	0.000009	0.000116	0.000104	0.000104	
16	0.000003	0.000003	0.000003	0.000011	0.000012	0.000011	0.000012	0.000011	0.000011	0.000154	0.000113	0.000104	
17	0.000003	0.000003	0.000003	0.000012	0.000011	0.000011	0.000014	0.000014	0.000014	0.000105	0.000113	0.000109	
18	0.000003	0.000003	0.000003	0.000012	0.000012	0.000012	0.000016	0.000017	0.000016	0.000834	0.000841	0.000772	
19	0.000004	0.000003	0.000003	0.000012	0.000012	0.000012	0.00002	0.000019	0.000019	0.000781	0.000776	0.000787	
20	0.000004	0.000004	0.000004	0.000012	0.000012	0.000012	0.000023	0.000023	0.000022	0.000776	0.000952	0.00093	
21	0.000004	0.000004	0.000004	0.000013	0.000012	0.000012	0.000041	0.000029	0.000029	0.000864	0.000928	0.000851	
22	0.000005	0.000005	0.000004	0.000014	0.000014	0.000013	0.000032	0.000032	0.000032	0.001021	0.000951	0.00082	
23	0.000006	0.000007	0.000005	0.000015	0.000015	0.000014	0.000036	0.000035	0.000035	0.000852	0.00088	0.000862	
24	0.000005	0.000005	0.000005	0.000017	0.000017	0.000017	0.000043	0.000043	0.000043	0.000893	0.000897	0.000873	

CSV file

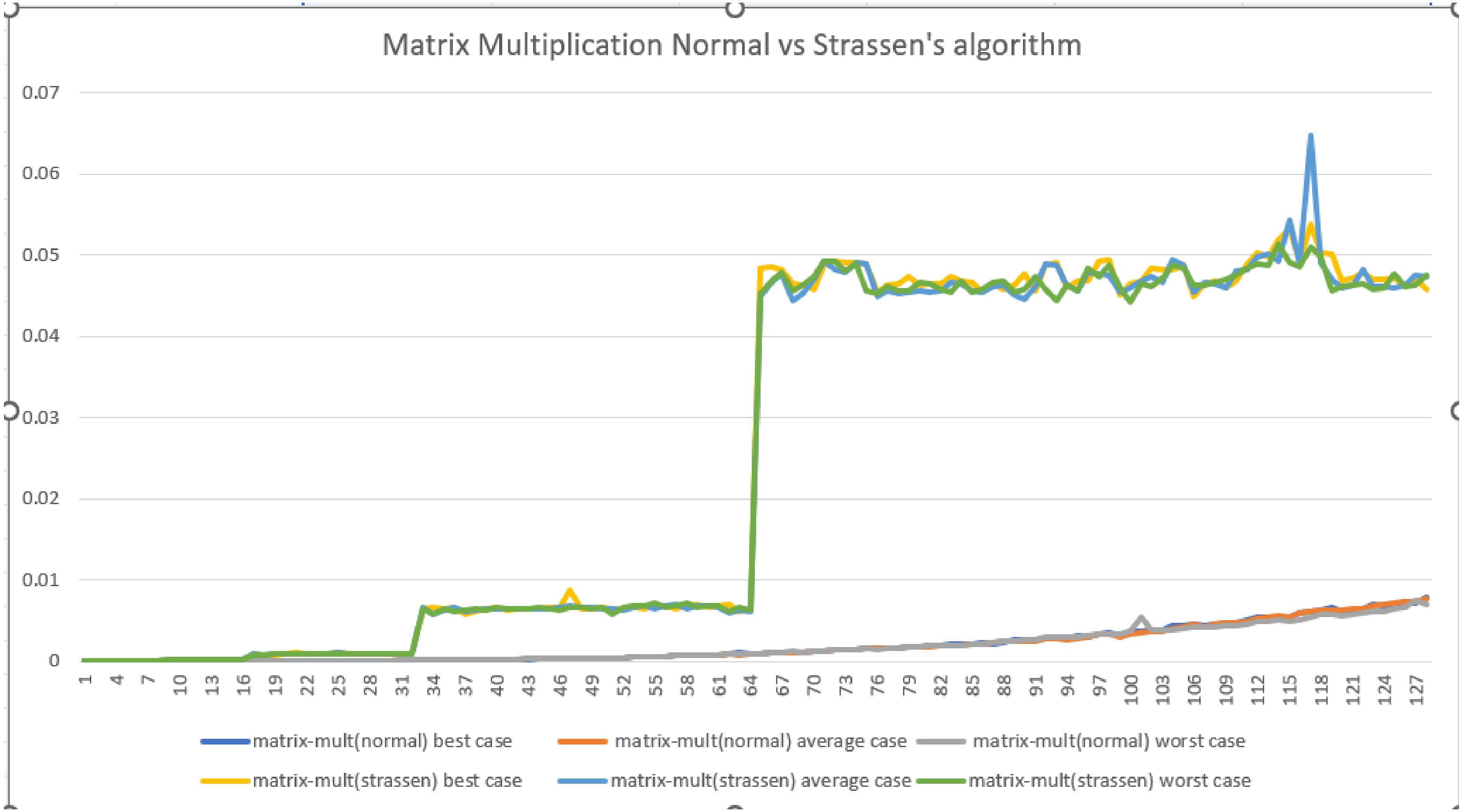
PLOTS: Min-Max



Both the normal implementation of the min-max algorithm as well as the divide and conquer approach work in O(n) time complexity, however, the divide and conquer algorithm Is slower due to extra overhead costs such as various function calls.

The best,worst and average cases for both algorithms are the same, as can be seen in the plots. Sorted and reverse sorted arrays were used as best and worst case.

Matrix Multiplication:



Here, even though strassen's algorithm has better complexity on paper, it incurs a lot of overhead costs for allocating the auxiliary arrays and extra operations, this at these input sizes, It seems to be slower.

The graph for Strassen's algorithm looks like steps because we have to expand the matrix and fill it with 0s if its size is not a power of 2.

Conclusion:

- Both the implementations of the mic-max algorithm have O(n) time complexity.
- Strassen’s algorithm, even though having better **asymptotic** time complexity, performs worse at the sizes we tested it at (1x1 to 128x128 matrix).
- Both divide and conquer algorithms seem to be slower because of the extra memory allocations and recursive calls. They also have worst space complexity, as they need extra stack space. Strassen’s algorithm also needs O(n^2) auxiliary space to store the temporary matrices.

Pseudocode:

1. Pseudocode for finding the minimum and maximum elements in an array using a simple linear scan:

```
Initialize min to infinity and max to negative infinity
For each element in the array
  If the element is less than min, update min to the element
  If the element is greater than max, update max to the element
Return min and max
```

2. pseudocode for divide-and-conquer approach:

```
Function MinMax(arr, low, high)
  If there is only one element
    Return the element as both min and max
  If there are two elements
```

```
    Return the smaller one as min and the larger one as max
  Else
    Calculate mid
    Call MinMax for the left half and get the min and max like, MinMax(arr,low,mid)
    Call MinMax for the right half and get the min and max like, MinMax(arr,mid+1,high)
    Return min(minLeft, minRight) as min and max(maxLeft, maxRight) as max
End Function
```

3. Pseudocode for matrix multiplication (naïve method):

```
Function matrixMultiply(A, B)
  Let C be a new matrix of the same size as A and B
  For i from 0 to number of rows in A
    For j from 0 to number of columns in B
      Set C[i][j] to 0
      For k from 0 to number of columns in A (or rows in B)
        Add A[i][k] * B[k][j] to C[i][j]
  Return C
End Function
```

4. Pseudocode for Strassen’s matrix multiplication algorithm:

```
Function strassenMultiply(A, B)

  If the size of A and B is 1

    Return a 1x1 matrix with the single element A[0][0] * B[0][0]

  Else
```

```
Divide A into four submatrices: A11, A12, A21, A22
Divide B into four submatrices: B11, B12, B21, B22

Calculate seven products of submatrices:
P1 = strassenMultiply(A11, B12 - B22)
P2 = strassenMultiply(A11 + A12, B22)
P3 = strassenMultiply(A21 + A22, B11)
P4 = strassenMultiply(A22, B21 - B11)
P5 = strassenMultiply(A11 + A22, B11 + B22)
P6 = strassenMultiply(A12 - A22, B21 + B22)
P7 = strassenMultiply(A11 - A21, B11 + B12)

Calculate the four quadrants of the result matrix:
C11 = P5 + P4 - P2 + P6
C12 = P1 + P2
C21 = P3 + P4
C22 = P5 + P1 - P3 - P7

Combine the four quadrants into a single matrix C
```

```
    Return the result matrix C
End Function
```

In this pseudocode, the "-" and "+" operations between matrices represent element-wise subtraction and addition, respectively. The function strassenMultiply is called recursively to calculate the seven products P1 to P7. The result matrix C is then formed by combining the four quadrants C11, C12, C21, and C22.

Theory:

- **Strassen’s matrix multiplication algorithm**

The standard or naive matrix multiplication algorithm takes O(n^3) time complexity, where n is the dimension of the matrices. Strassen's algorithm, on the other hand, reduces the time complexity to approximately O(n^log2(7)), which is roughly O(n^2.81). This is achieved by reducing the number of recursive multiplication calls from 8 to 7.

The algorithm works as follows:

1. The given matrices A and B are divided into four sub-matrices each.
2. Seven products of these sub-matrices are calculated using specific formulas.
3. These seven products are then combined using addition and subtraction operations to get the four quadrants of the result matrix.
4. The four quadrants are combined to form the final result matrix.

It's important to note that Strassen's algorithm is more efficient than the standard matrix multiplication for large matrices. However, for smaller matrices or sparse matrices, the overhead of additional addition and subtraction operations and the recursive calls can make it less efficient than the standard algorithm.

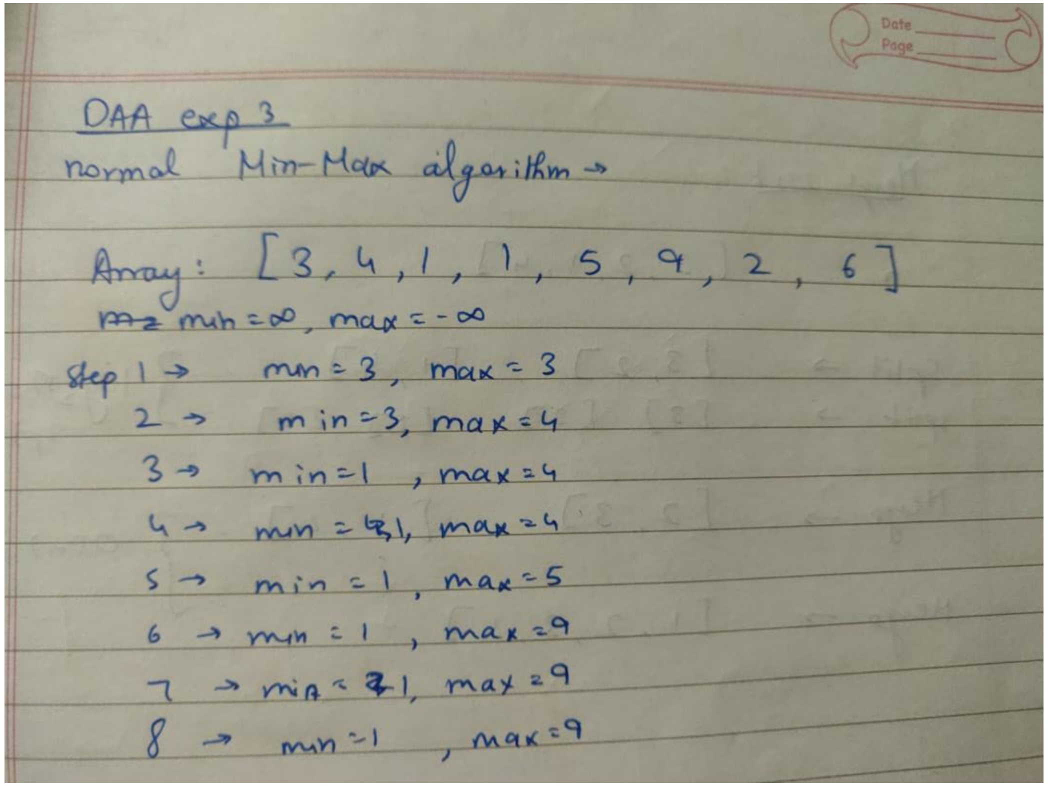
Also, Strassen's algorithm assumes that the matrices are square and have dimensions that are a power of 2. If this is not the case, the matrices must be padded with zeros to make them fit this requirement, which can add to the complexity.

Time and space complexities of normal and divide and conquer Min-max algorithm:

- **Normal (Linear Scan) Min-Max Algorithm:**

  Time Complexity: O(n) - We scan through the array once, comparing each element with the current min and max.

  Space Complexity: O(1) - We only need to store the current min and max, so the space complexity is constant.



Date ______
Page ______

DAA exp 3

normal Min-Max algorithm →

Array: [3, 4, 1, 1, 5, 9, 2, 6]

~~ma~~ min = ∞, max = -∞

step 1 → min = 3, max = 3

2 → min = 3, max = 4

3 → min = 1 , max = 4

4 → min = ~~3~~1, max = 4

5 → min = 1 , max = 5

6 → min = 1 , max = 9

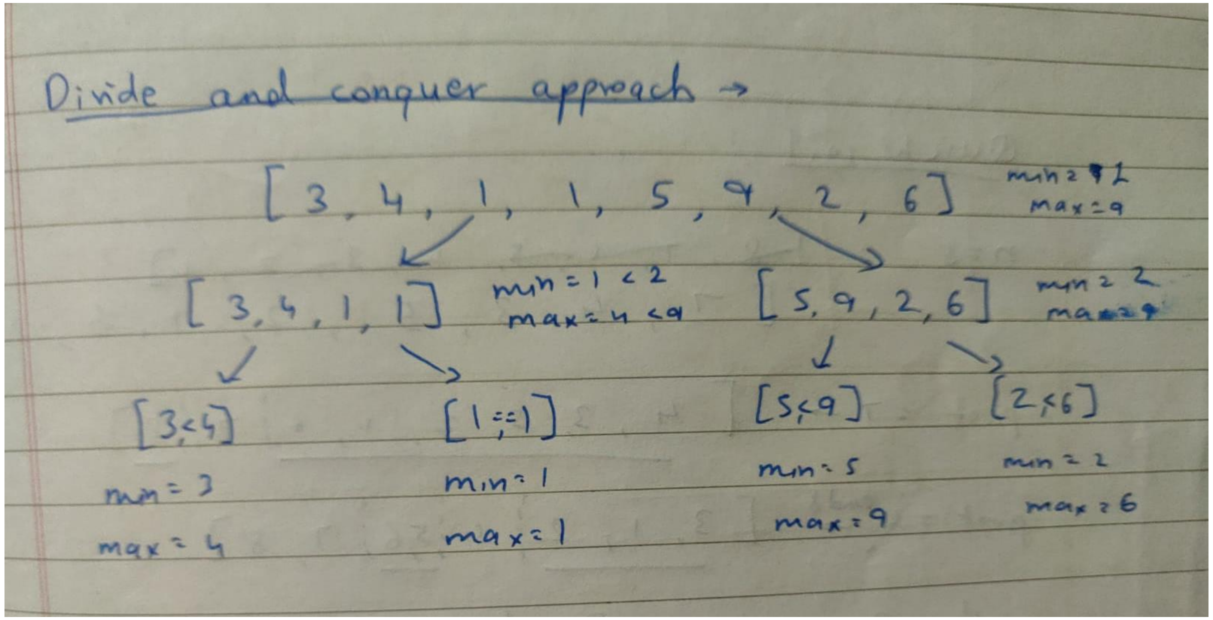
7 → min = ~~7~~1, max = 9

8 → min = 1 , max = 9

- **Divide and Conquer Min-Max Algorithm:**

  Time Complexity: O(n) - Although we divide the problem into smaller parts, we still have to compare each element once.

  Space Complexity: O(log n) - The depth of the recursion tree is log(n), so we need that much space for the call stack.



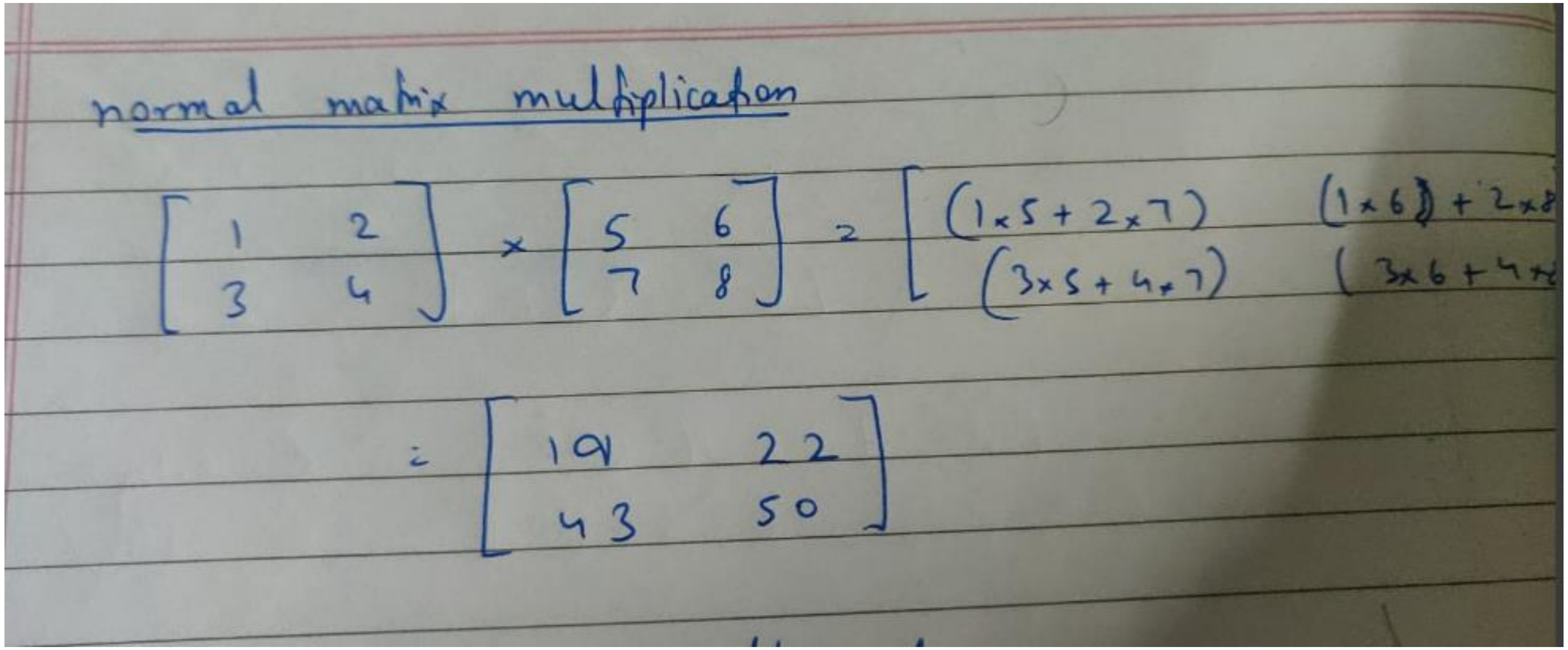
both algorithms have the same time complexity, but the divide and conquer version has a larger space complexity due to the recursive calls.

<u>Time and space complexities of normal and Strassen's matrix multiplication algorithm:</u>

- **Normal Matrix Multiplication Algorithm:**

  Time Complexity: O(n^3) - For each element in the resulting matrix, we need to perform a dot product operation which takes O(n) time. Since the resulting matrix has n^2 elements, the total time complexity is O(n^3).

  Space Complexity: O(n^2) - We need to store the resulting matrix, which has n^2 elements.



normal matrix multiplication

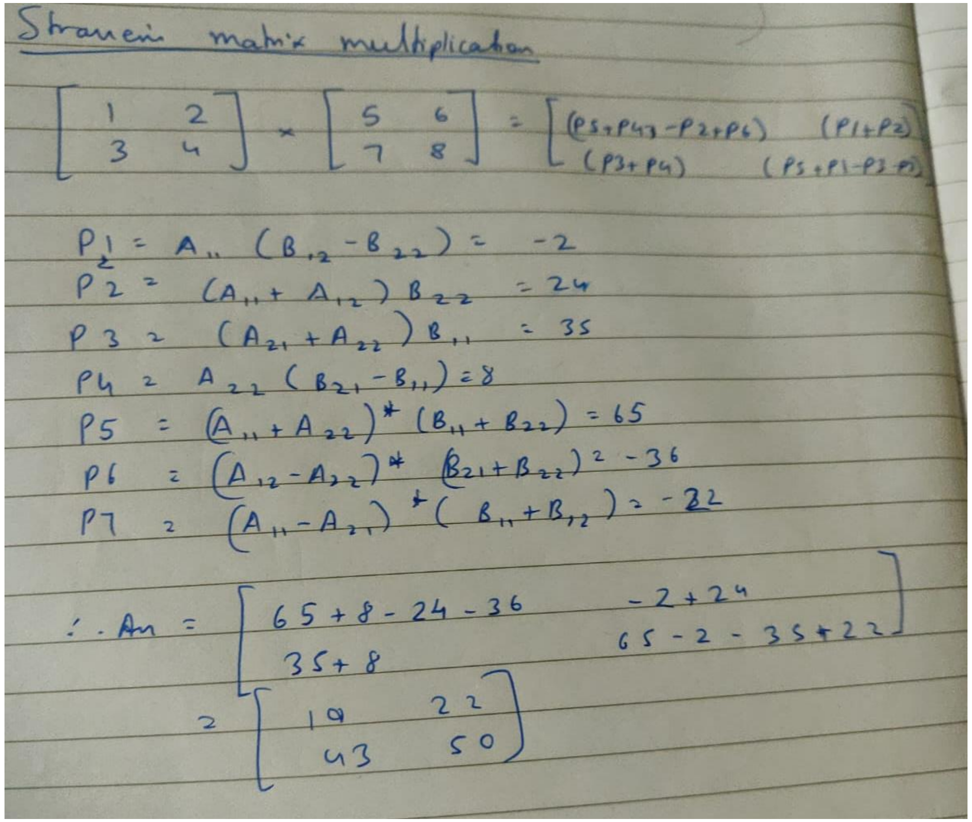
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1\times5+2\times7) & (1\times6)+2\times8 \\ (3\times5+4\times7) & (3\times6+4\times \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

- **Strassen's Matrix Multiplication Algorithm:**

  Time Complexity: O(n^log2(7)) - The algorithm divides the matrix into 4 sub-matrices, and performs 7 multiplications recursively, adding several addition steps to save a multiplication. Hence the time complexity.

Space Complexity: O(n^2) - Despite the divide and conquer approach, we still need to store the resulting matrix, which has n^2 elements. Additionally, the algorithm requires storage for the intermediate matrices, but this does not change the asymptotic space complexity.



Strassen matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (P5+P4-P2+P6) & (P1+P2) \\ (P3+P4) & (P5+P1-P3-P7) \end{bmatrix}$$

$P1 = A_{11}(B_{12} - B_{22}) = -2$

$P2 = (A_{11} + A_{12}) B_{22} = 24$

$P3 = (A_{21} + A_{22}) B_{11} = 35$

$P4 = A_{22}(B_{21} - B_{11}) = 8$

$P5 = (A_{11} + A_{22}) * (B_{11} + B_{22}) = 65$

$P6 = (A_{12} - A_{22}) * (B_{21} + B_{22}) = -36$

$P7 = (A_{11} - A_{21}) * (B_{11} + B_{12}) = -22$

$$\therefore An = \begin{bmatrix} 65+8-24-36 & -2+24 \\ 35+8 & 65-2-35+22 \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Strassen's algorithm has a lower time complexity than the normal matrix multiplication algorithm, making it more efficient for large matrices