Algorithms Homework 1 Report

2020. 4. 19.
20161349 Jaeho Kim

Intro

Matrix multiplication is a central operation requiring multiple numerical operations. The usual matrix multiplication we learn in high school is an iterative algorithm that takes a complexity[1] of $O(n^3)$ when multiplying the size of $n \times n$ matrices. An alternative algorithm is the divide and conquer method where it relies on partitioning the matrix into blocks. The divide and conquer algorithm divides the matrices into 4 sub-matrices and performs 8 multiplications with matric size $n/2$ and 4 additions. As the algorithm is recursive the time complexity would be $T(n) = 8T(n/2) + O(n^2)$. Unfortunately, this also takes an algorithm complexity of $O(n^3)$ which is the same as the usual matrix multiplication. And here, Strassen comes to the rescue. Strassen proposes to reduce the recursive calls from 8 to 7, leading to time complexity of $T(n) = 7T(n/2) + O(n^2)$. This has an algorithm complexity of $O(n^{log7})$ leading to a faster implementation of matrix multiplication.

In our homework, we are given an input.txt file which consists of integer Matrix A and B. We are required to implement 4 matrix multiplication algorithms which are schoolbook multiplication, naïve divide and conquer, Strassen's method, and optimized Strassen's method. Matrix sizes will only be given with size that equals the power of 2. Execution time should be compared between algorithms. Here is the detail of my implementation environment.

Language	Python3.6
OS	Ubuntu 16.04
CPU	Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz

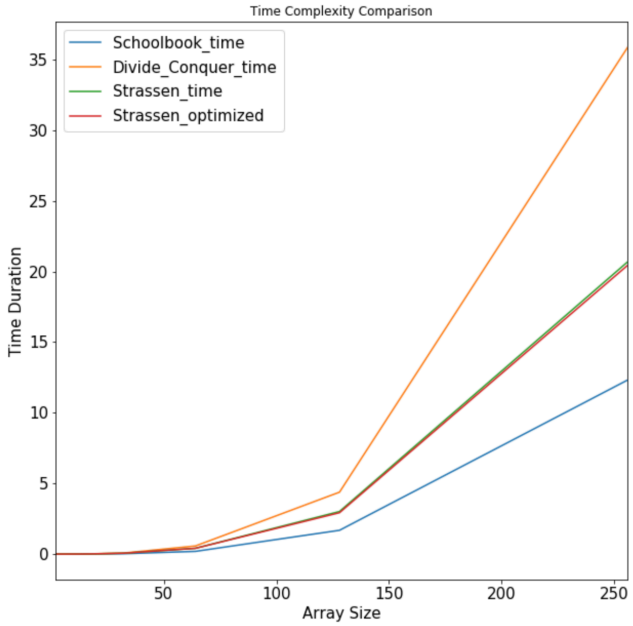
All the source code is in the appendix.

[1] https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

Observation

The implementation of the algorithm itself was not difficult as the code was not much different from the pseudo-code provided in the textbook. However, after implementation, I found out that the execution time of the Naïve divide and conquer and Strassen took so much more time compared to the standard matrix multiplication. It took 20 times longer for divide and conquer and 8 times more for Strassen compared to schoolbook implementation. Below is the time comparison when I first implemented the algorithm.

Array_size	Schoolbook_time	Divide_Conquer_time	Strassen_time
2	4.7206878662109375e-05	0.0002162456512451172	0.0001704692840576172
4	0.00010967254638671875	0.0013222694396972656	0.0012149810791015625
8	0.0006349086761474609	0.01047968864440918	0.005822181701660156
16	0.002545595169067383	0.05387568473815918	0.03977251052856445
32	0.019907236099243164	0.5146727561950684	0.33355069160461426
64	0.22497129440307617	4.117962598800659	2.360898733139038
128	1.554166555404663	33.267693758010864	16.539644956588745
256	13.431137800216675	263.7457625865936	116.66768956184387
512	108.36793613433838	2108.810677051544	808.7790343761444



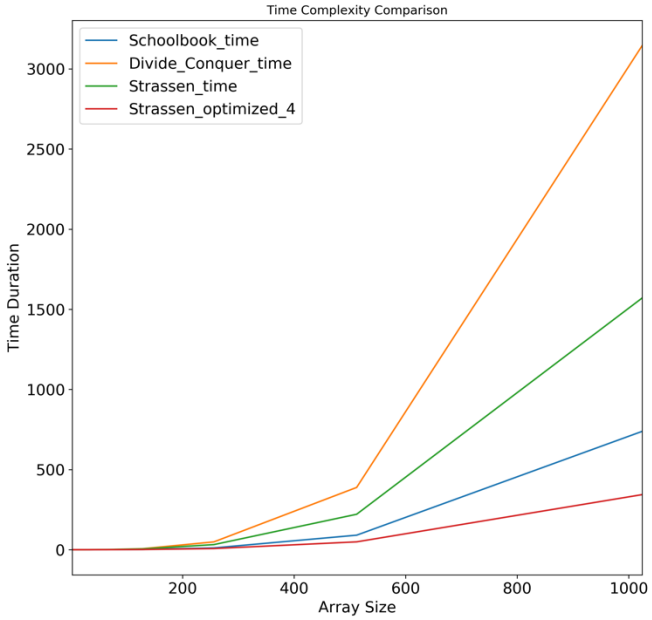
There were some comments in Google saying that Strassen will only perform better when the array size is big enough. However, even after increasing the size of the matrix to 1024, there wasn't much difference although the gap decreased slightly. I tried reducing the lines of code, removed Numpy functions like "np.vstack" and made the codes neater hoping that it would result the difference. However, the results weren't much different from what I first observed.

Optimization

With some more digging into Google, I found this comment[2] where it states *"Strassen's algorithm has a better Big O complexity, but constants do matter in reality, which means in reality you're better off with a standard n^3 matrix multiplication for smaller problem sizes."*

To decrease the constant time I figured out Both Naïve divide and conquer and Strassen were making a recursive call until when $n == 1$. After trying out with different n, Strassen shows way much better performance than the standard matrix multiplication. The experimental results are shown below.

	Array_size	Schoolbook_time	Divide_Conquer_time	Strassen_time	Strassen_optimized_4
0	2	1.8358230590820312e-05	4.220008850097656e-05	7.653236389160156e-05	1.1444091796875e-05
1	4	9.5367431640625e-05	0.00027322769165039057	0.0005576610565185547	5.507469177246094e-05
2	8	0.0003728865771484375	0.0018463134765625	0.002040386199951172	0.00043487548828125
3	16	0.002859354019165039	0.012175559997558594	0.013447046279907228	0.003189563751220703
4	32	0.022125244140625	0.09489798545837402	0.09223532676696776	0.02053999900817871
5	64	0.1759471893310547	0.7579932212829591	0.6472883224487305	0.14138078689575195
6	128	1.3935551643371582	6.051497459411621	4.538825750350952	1.0051209926605225
7	256	11.085448026657104	48.66181397438049	31.73547124862671	7.021599531173706
8	512	90.54794383049013	388.38110184669495	221.06101369857788	48.91199851036072
9	1024	738.2962219715117	3145.115173816681	1570.400215625763	343.60464906692505



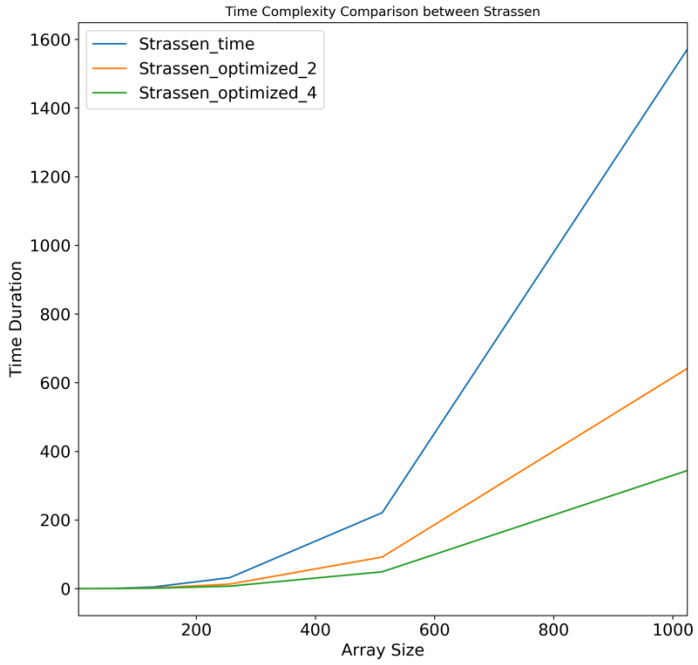
We can see that Strassen now performs better than any other algorithms.

[2] https://stackoverflow.com/questions/11495723/why-is-strassen-matrix-multiplication-so-much-slower-than-standard-matrix-multip

Strassen Comparison

	Array_size	Strassen_time	Strassen_optimized_2	Strassen_optimized_4
0	2	7.653236389160156e-05	1.33514404296875e-05	1.1444091796875e-05
1	4	0.0005576610565185547	0.00012946128845214844	5.507469177246094e-05
2	8	0.002040386199951172	0.0008156299591064452	0.00043487548828125
3	16	0.013447046279907228	0.005401372909545898	0.003189563751220703
4	32	0.09223532676696776	0.03788375854492188	0.02053999900817871
5	64	0.6472883224487305	0.2641913890838623	0.14138078689575195
6	128	4.538825750350952	1.859738111495972	1.0051209926605225
7	256	31.73547124862671	12.974822998046875	7.021599531173706
8	512	221.06101369857788	91.91951942443848	48.91199851036072
9	1024	1570.400215625763	641.0124650001526	343.60464906692505

Strassen Time : Ordinary Strassen with base recursion as 1
Strassen_optimized_2 : Strassen with base recursion as 2
Strassen_optimized_4 : Strassen with base recursion as 4



As we can see by increasing the base case size, the Strassen algorithm takes shorter time.

Appendix

```
import matplotlib.pyplot as plt
import numpy as np
import time
import pandas as pd

def create_test_case():
    input_size = [2 ** i for i in range(1,12)]
    log= pd.DataFrame(index=[],columns=
['Array_size','Schoolbook_time','Divide_Conquer_time','Divide_Conquer_time2','Stras
sen_time','Strassen_optimized_2','Strassen_optimized_4'])
    print(input_size)
    for size in input_size:
        print(size)
        #Initialize random array with size corresponding to the power of 2
        array_1 = np.random.randint(low=-100,high=100,size=(size,size),dtype=int)
        array_2 = np.random.randint(low=-100,high=100,size=(size,size),dtype=int)

        # Test school book algorithm
        print("Testing School Book size {}".format(size))
        start_time = 0
        start_time = time.time()
        result = schoolbook2(array_1,array_2)
        schoolbook_duration = time.time()-start_time
        #print(result)

        # Divide and conquer
        print("Testing Divide and conquer size {}".format(size))
        start_time = 0
        start_time = time.time()
        result = divide_and_conquer(array_1,array_2)
        divide_and_conquer_duration = time.time()-start_time
        #print(result)

        # Divide and conquer2
        print("Testing Divide and conquer 2 size {}".format(size))
        start_time = 0
        start_time = time.time()
        result = divide_and_conquer2(array_1,array_2)
        divide_and_conquer_duration2 = time.time()-start_time

        # Strassen
        print("Testing Strassen size {}".format(size))
        start_time = 0
        start_time = time.time()
        result = strassen(array_1,array_2)
        strassen_duration = time.time()-start_time

        # Strassen_optimized_leaf2
```

```
        print("Testing Strassen optimized 2 size {}".format(size))
        start_time = 0
        start_time = time.time()
        result = strassen_optimized_leaf2(array_1,array_2)
        strassen_duration_optimized_2 = time.time()-start_time

        # Strassen_optimized_leaf4
        print("Testing Strassen optimized 4 size {}".format(size))
        start_time = 0
        start_time = time.time()
        result = strassen_optimized_leaf_4(array_1,array_2)
        strassen_duration_optimized_4 = time.time()-start_time

        print("SCHOOL BOOK:{}, DIVIDE AND CONQUER:{}, DIVIDE AND CONQUER2:{},
STRASSEN:{}, STRASSEN OPTIMIZED_2:{}, STRASSEN
OPTIMIZED_4:{}".format(schoolbook_duration,

divide_and_conquer_duration, divide_and_conquer_duration2,
strassen_duration,strassen_duration_optimized_2,strassen_duration_optimized_4))

        tmp = pd.Series([
            size,
            schoolbook_duration,
            divide_and_conquer_duration,
            divide_and_conquer_duration2,
            strassen_duration,
            strassen_duration_optimized_2,
            strassen_duration_optimized_4
        ],
index=['Array_size','Schoolbook_time','Divide_Conquer_time','Divide_Conquer_time2',
'Strassen_time','Strassen_optimized_2','Strassen_optimized_4'])
        log = log.append(tmp,ignore_index=True)
        log.to_csv('new_time_complexity_python.csv',index=False)
    return log


def read_input(input):
    array = np.loadtxt(input,dtype='i',delimiter=' ')
    #Seperate array
    array_first,array_second = np.split(array,2,axis=0)
    return array_first, array_second

def schoolbook(array_first,array_second):
    #print("SCHOOL BOOK Implementation")
    #Create empty matrix to use it as return matrix
    result = np.zeros_like(array_first)
    for row_idx, row in enumerate(array_first):
        for i in range(len(row)):
            for cell_idx, cell in enumerate(row):
                result[row_idx,i]+= cell * array_second[cell_idx,i]
```

```
    return result

def schoolbook2(array_first,array_second):
    n = len(array_first)
    result = np.zeros((n,n))

    for i in range(n):
        for j in range(n):
            for k in range(n):
                result[i,j] += array_first[i,k] *array_second[k,j]
    return result



def divide_and_conquer(array_first,array_second):
    n = len(array_first)
    if n == 1:
        return array_first * array_second
    else:
        a11 = array_first[:int(len(array_first)/2),:int(len(array_first)/2)]
        a12 = array_first[:int(len(array_first)/2),int(len(array_first)/2):]
        a21 = array_first[int(len(array_first)/2):,:int(len(array_first)/2)]
        a22 = array_first[int(len(array_first)/2):,int(len(array_first)/2):]

        b11 = array_second[:int(len(array_second)/2),:int(len(array_second)/2)]
        b12 = array_second[:int(len(array_second)/2),int(len(array_second)/2):]
        b21 = array_second[int(len(array_second)/2):,:int(len(array_second)/2)]
        b22 = array_second[int(len(array_second)/2):,int(len(array_second)/2):]

        c11 = divide_and_conquer(a11,b11) + divide_and_conquer(a12,b21)
        c12 = divide_and_conquer(a11,b12) + divide_and_conquer(a12,b22)
        c21 = divide_and_conquer(a21,b11) + divide_and_conquer(a22,b21)
        c22 = divide_and_conquer(a21,b12) + divide_and_conquer(a22,b22)

        result = np.zeros((n,n))
        result[:int(len(result)/2),:int(len(result)/2)] = c11
        result[:int(len(result)/2),int(len(result)/2):] = c12
        result[int(len(result)/2):,:int(len(result)/2)] = c21
        result[int(len(result)/2):,int(len(result)/2):] = c22
        #print("Divide and Conquer")
    return result

def divide_and_conquer2(array_first,array_second):
    n = len(array_first)
    if n <= 4:
        return schoolbook2(array_first,array_second)
    else:
        a11 = array_first[:int(len(array_first)/2),:int(len(array_first)/2)]
        a12 = array_first[:int(len(array_first)/2),int(len(array_first)/2):]
        a21 = array_first[int(len(array_first)/2):,:int(len(array_first)/2)]
```

```
        a22 = array_first[int(len(array_first)/2):,int(len(array_first)/2):]

        b11 = array_second[:int(len(array_second)/2),:int(len(array_second)/2)]
        b12 = array_second[:int(len(array_second)/2),int(len(array_second)/2):]
        b21 = array_second[int(len(array_second)/2):,:int(len(array_second)/2)]
        b22 = array_second[int(len(array_second)/2):,int(len(array_second)/2):]

        c11 = divide_and_conquer2(a11,b11) + divide_and_conquer2(a12,b21)
        c12 = divide_and_conquer2(a11,b12) + divide_and_conquer2(a12,b22)
        c21 = divide_and_conquer2(a21,b11) + divide_and_conquer2(a22,b21)
        c22 = divide_and_conquer2(a21,b12) + divide_and_conquer2(a22,b22)

        result = np.zeros((n,n))
        result[:int(len(result)/2),:int(len(result)/2)] = c11
        result[:int(len(result)/2),int(len(result)/2):] = c12
        result[int(len(result)/2):,:int(len(result)/2)] = c21
        result[int(len(result)/2):,int(len(result)/2):] = c22
        #print("Divide and Conquer")
    return result

def strassen(array_first,array_second):
    n = len(array_first)
    if n == 1:
        return array_first * array_second
    else:
        a11 = array_first[:int(len(array_first)/2),:int(len(array_first)/2)]
        a12 = array_first[:int(len(array_first)/2),int(len(array_first)/2):]
        a21 = array_first[int(len(array_first)/2):,:int(len(array_first)/2)]
        a22 = array_first[int(len(array_first)/2):,int(len(array_first)/2):]

        b11 = array_second[:int(len(array_second)/2),:int(len(array_second)/2)]
        b12 = array_second[:int(len(array_second)/2),int(len(array_second)/2):]
        b21 = array_second[int(len(array_second)/2):,:int(len(array_second)/2)]
        b22 = array_second[int(len(array_second)/2):,int(len(array_second)/2):]

        S1 = b12 - b22
        S2 = a11 + a12
        S3 = a21 + a22
        S4 = b21 - b11
        S5 = a11 + a22
        S6 = b11 + b22
        S7 = a12 - a22
        S8 = b21 + b22
        S9 = a11 - a21
        S10 = b11 + b12

        P1 = strassen(a11,S1)
        P2 = strassen(S2,b22)
        P3 = strassen(S3,b11)
        P4 = strassen(a22,S4)
```

```
        P5 = strassen(S5,S6)
        P6 = strassen(S7,S8)
        P7 = strassen(S9,S10)

        c11 = P5 +P4 -P2 +P6
        c12 = P1 +P2
        c21 = P3 +P4
        c22 = P5 +P1 -P3 -P7

        result = np.zeros((n,n))
        result[:int(len(result)/2),:int(len(result)/2)] = c11
        result[:int(len(result)/2),int(len(result)/2):] = c12
        result[int(len(result)/2):,:int(len(result)/2)] = c21
        result[int(len(result)/2):,int(len(result)/2):] = c22
        return result


def strassen_optimized_leaf2(array_first,array_second):
    n = len(array_first)
    if n <= 2:
        return schoolbook2(array_first,array_second)
    else:
        a11 = array_first[:int(len(array_first)/2),:int(len(array_first)/2)]
        a12 = array_first[:int(len(array_first)/2),int(len(array_first)/2):]
        a21 = array_first[int(len(array_first)/2):,:int(len(array_first)/2)]
        a22 = array_first[int(len(array_first)/2):,int(len(array_first)/2):]

        b11 = array_second[:int(len(array_second)/2),:int(len(array_second)/2)]
        b12 = array_second[:int(len(array_second)/2),int(len(array_second)/2):]
        b21 = array_second[int(len(array_second)/2):,:int(len(array_second)/2)]
        b22 = array_second[int(len(array_second)/2):,int(len(array_second)/2):]

        P1 = strassen_optimized_leaf2(a11,b12 - b22)
        P2 = strassen_optimized_leaf2(a11 + a12,b22)
        P3 = strassen_optimized_leaf2(a21 + a22,b11)
        P4 = strassen_optimized_leaf2(a22,b21 - b11)
        P5 = strassen_optimized_leaf2(a11 + a22,b11 + b22)
        P6 = strassen_optimized_leaf2(a12 - a22,b21 + b22)
        P7 = strassen_optimized_leaf2(a11 - a21,b11 + b12)

        result = np.zeros((n,n))
        result[:int(len(result)/2),:int(len(result)/2)] = P5 +P4 -P2 +P6
        result[:int(len(result)/2),int(len(result)/2):] = P1 +P2
        result[int(len(result)/2):,:int(len(result)/2)] = P3 +P4
        result[int(len(result)/2):,int(len(result)/2):] = P5 +P1 -P3 -P7
        return result

def strassen_optimized_leaf_4(array_first,array_second):
    n = len(array_first)
    if n <= 4:
```

```
        return schoolbook2(array_first,array_second)
    else:
        a11 = array_first[:int(len(array_first)/2),:int(len(array_first)/2)]
        a12 = array_first[:int(len(array_first)/2),int(len(array_first)/2):]
        a21 = array_first[int(len(array_first)/2):,:int(len(array_first)/2)]
        a22 = array_first[int(len(array_first)/2):,int(len(array_first)/2):]

        b11 = array_second[:int(len(array_second)/2),:int(len(array_second)/2)]
        b12 = array_second[:int(len(array_second)/2),int(len(array_second)/2):]
        b21 = array_second[int(len(array_second)/2):,:int(len(array_second)/2)]
        b22 = array_second[int(len(array_second)/2):,int(len(array_second)/2):]

        P1 = strassen_optimized_leaf_4(a11,b12 - b22)
        P2 = strassen_optimized_leaf_4(a11 + a12,b22)
        P3 = strassen_optimized_leaf_4(a21 + a22,b11)
        P4 = strassen_optimized_leaf_4(a22,b21 - b11)
        P5 = strassen_optimized_leaf_4(a11 + a22,b11 + b22)
        P6 = strassen_optimized_leaf_4(a12 - a22,b21 + b22)
        P7 = strassen_optimized_leaf_4(a11 - a21,b11 + b12)

        result = np.zeros((n,n))
        result[:int(len(result)/2),:int(len(result)/2)] = P5 +P4 -P2 +P6
        result[:int(len(result)/2),int(len(result)/2):] = P1 +P2
        result[int(len(result)/2):,:int(len(result)/2)] = P3 +P4
        result[int(len(result)/2):,int(len(result)/2):] = P5 +P1 -P3 -P7
        return result


if __name__ == "__main__":
    log = create_test_case()
```