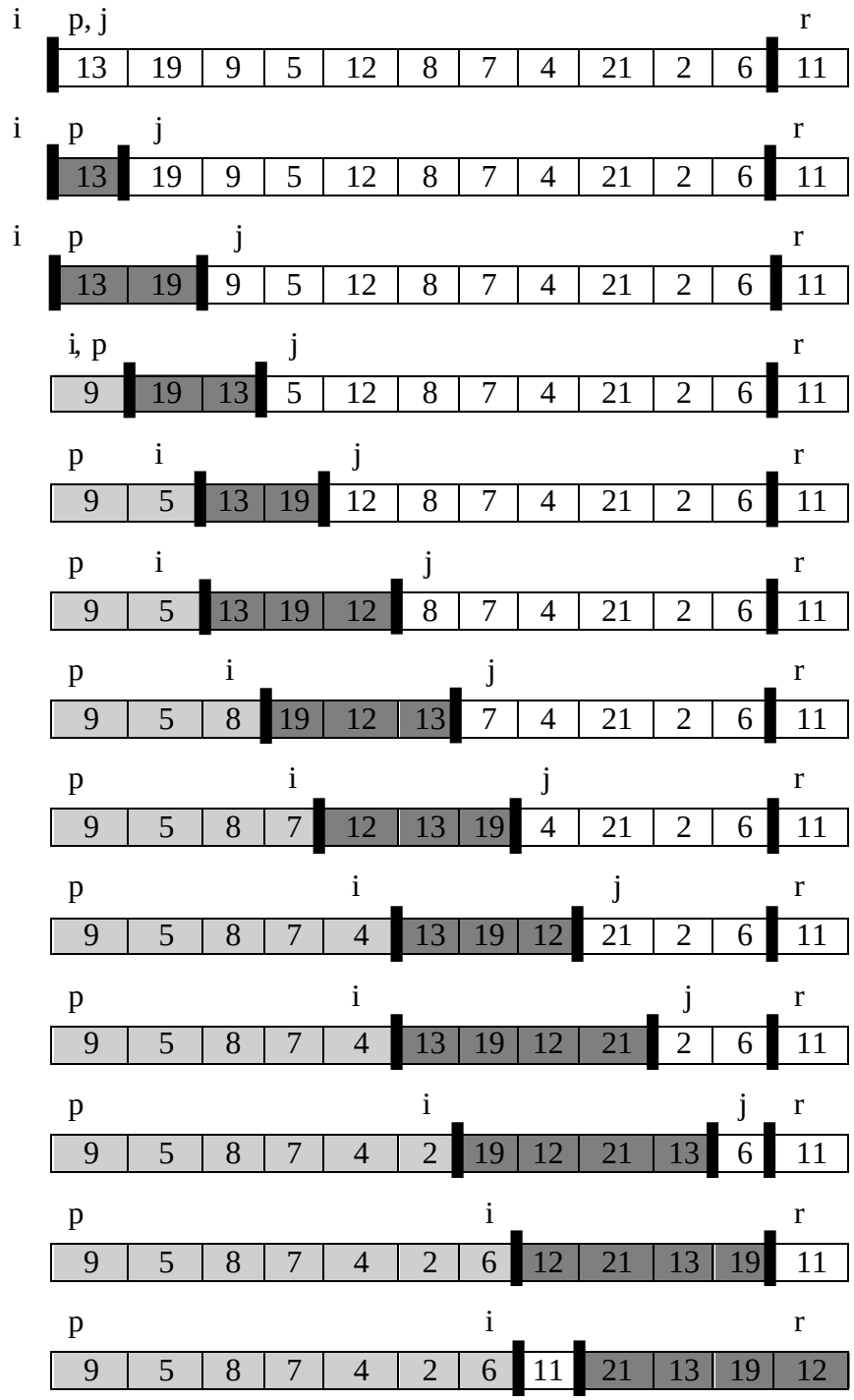
Homework 4, CPSC 4100-01, Winter 2017

1) Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array A[13,19,9,5,12,8,7,4,21,2,6,11]. [CLRS 7.1-1] 5 points



2) Assume that we have an array with length n containing integer numbers. Develop an efficient algorithm that put all the negative numbers on the right side of the array and all the positive numbers on the left side. As an example:
A = [5 , -2 , 3 , 4 , -1 , 0 , -2 , 2]
Answer = [5 , 3 , 4 , 0 , 2 , -2 , -1 , -2]
Order of the items in the answer are not important.
15 points

```
void pos_neg_Partition(int* A[], int p) {
        int x = A[r];
        int i = p - 1;
        int temp = null;
        for(int j = p, j < r, j++){
                if (A[j] >= 0){
                        i++:
                        temp = A[i];
                        A[i] = A[j];
                        A[j] = temp;
                }
        }
        temp = A[i + 1];
        A[i + 1] = A[r];
        A[r] = A[temp];
}
```

This is very similar to the PARTITION algorithm. Instead of checking A[j] is greater than or equal to the pivot, it checks to see if it is less than or equal to 0. It swaps the last value that was less than 0 with the current value that is greater than 0. At the end the last value gets swapped with the last value less than 0. It does not matter if this last value, r, is less than, greater than, or equal to 0 because it will be placed in the middle of the sections.

3) We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation on the BUILD-MAX-HEAP procedure:

```
BUILD-MAX-HEAP(A)
1       A.heap-size = A.length
2       for i = ⌊A.length/2⌋ downto 1
3               MAX-HEAPIFY (A, i)

BUILD-MAX-HEAP'(A)
1       A.heap-size = 1
2       for i = 2 to A.length
3               MAX-HEAP-INSERT (A, A[i])
```

a) Do the procedures $BUILD\_MAX\_HEAP$ and $BUILD\_MAX\_HEAP'$ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

No, when building a heap using the array A[4, 1, 3, 2, 16, 9, 10, 14, 8, 7] the two methods build heaps that have the correct structure but are in slightly different orders. This is due to the fact that there is no defined relationship between the children of a parent and the manner in which the two methods build the heap. BUILD_MAX_HEAP sorts the array as if it is already a heap and looks at half the values. BUILD_MAX_HEAP' adds each element from the array into a heap structure one by one.

b) Show that in the worst case, $BUILD\_MAX\_HEAP'$ requires $\Theta(n\log n)$ time to build an n-element heap.
[CLRS 6-1]
15 points

If the array is in such an order, ascending order for example, then each time that an element is added to the heap, it will be placed in the last leaf node on the lowest level of the heap. This means that each new element will always be the new max for the heap and it will have to be moved all the way up to the top. This is cause the worst-case number of comparisons and result in a $\Theta(n\log n)$ run time.

4) Implement the Strassen's method for matrix multiplication, to multiply two $n\times n$ matrices. Test your code for three example: a 2×2 case, a 4×4 case, and an 8×8 case. Report your code as well as your test cases.
15 points

```
# Nils Fenske
# HW 4 Q 4

import random

random.seed()

n = 4

A = []
for i in range(0, n):
    A.append([])
    for j in range(0, n):
        A[i].append(random.randint(-10, 10))

B = []
for l in range(0, n):
    B.append([])
    for k in range(0, n):
        B[l].append(random.randint(-10, 10))

#print(A)
#print(B)

def SQUARE_MARTIX_MULTIPLY_RECURSIVE(current_A, current_B):
    r = len(current_A)
```

```
    C = []
    for f in range(0, r):
        C.append([])
        for v in range(0, r):
            C[f].append(0)
    if len(current_A) == 1: C[0][0] = current_A[0][0] * current_B[0][0]
    else:
        new_A = []
        new_B = []
        for x in range(0, int(r/2)):
            new_A.append([])
            new_B.append([])
            for y in range(0, int(r/2)):
                new_A[x].append([])
                new_B[x].append([])
                for z in range(0, int(r/2)):
                    new A[y][z].append(current A[y][z])
                    new_B[y][z].append(current_B[y][z])


        C[0][0] = SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[0][0], new_B[0][0]) +
SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[0][1], new_B[1][0])
        C[0][1] = SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[0][0], new_B[0][1]) +
SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[0][1], new_B[1][1])
        C[1][0] = SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[1][0], new_B[0][0]) +
SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[1][1], new_B[1][0])
        C[1][1] = SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[1][0], new_B[0][1]) +
SQUARE_MARTIX_MULTIPLY_RECURSIVE(new_A[1][1], new_B[1][1])
    return(C)

print(SQUARE_MARTIX_MULTIPLY_RECURSIVE(A,B))
```

I used randomly filled matrices to test my code. N can be set to a number such as 2, 4, or 8 and a matrix is filled in of size n x n. I was unable to figure out exactly how to partition the matrix. I understand fully what needs to be done. The matrix is supposed to be partitioned into n/2 x n2/ quadrants and the function is called on each quadrant until the quadrants all consist of just one element. I simply ran out of time and couldn't get the code to work exactly.