CSE373 Assignment 1

Aditya Balwani
SBUID: 109353920

April 5, 2016

1 Rotating Images

1.1 Part 1 : n is power of 2

```
# A: input matrix, n: width of matrix, power of 2
def rotateSquarePow2(A, n):
    if(n<=1):
        return A
    else:
        rotatedQuad1 = rotateSquarePow2(A[0:(n/2)-1][0:(n/2)-1], n/2)
        rotatedQuad2 = rotateSquarePow2(A[0:(n/2)-1][n/2:n-1], n/2)
        rotatedQuad3 = rotateSquarePow2(A[n/2:n-1][0:(n/2)-1], n/2)
        rotatedQuad4 = rotateSquarePow2(A[n/2:n-1][n/2:n-1], n/2)

        A[0:(n/2)-1][0:(n/2)-1] = rectangularCopy(rotatedQuad4)
        A[0:(n/2)-1][n/2:n-1] = rectangularCopy(rotatedQuad1)
        A[n/2:n-1][0:(n/2)-1] = rectangularCopy(rotatedQuad2)
        A[n/2:n-1][n/2:n-1] = rectangularCopy(rotatedQuad3)

        return A
```

1.2 Part 2 : n not is power of 2

```
# A: input matrix, m: height of A, n: width of matrix
def rotateArbitrary(A, m, n):
    if(n<=1):
        return A
    else:
        rotatedQuad1 = rotateArbitrary(A[0:(m/2)-1][0:(n/2)-1], m/2, n/2)
        rotatedQuad2 = rotateArbitrary(A[0:(m/2)-1][n/2:n-1], m/2, n/2)
        rotatedQuad3 = rotateArbitrary(A[n/2:m-1][0:(n/2)-1], m/2, n/2)
        rotatedQuad4 = rotateArbitrary(A[n/2:m-1][n/2:n-1], m/2, n/2)

        B[0:(m/2)-1][0:(n/2)-1] = rectangularCopy(rotatedQuad4)
        B[0:(m/2)-1][n/2:n-1] = rectangularCopy(rotatedQuad1)
        B[n/2:m-1][0:(n/2)-1] = rectangularCopy(rotatedQuad2)
        B[n/2:m-1][n/2:n-1] = rectangularCopy(rotatedQuad3)

        return A
```

1.3 Part 3: find T(n) if rectangular copy is $O(n^2)$

Assuming rc(a) is the running time of Rectangular Copy on a (a x a) matrix

$$\begin{aligned} T(n) &= 4T(n/2) + 4rc(n/2) + c' \\ &= 4^2T(n/2^2) + 4^2rc(n/2^2) + 4rc(n/2) + c' + c' \\ &= ... \\ &= 4^iT(n/2^i) + \sum_{j=1}^{i} 4^j rc(n/2^j) + ic' \end{aligned}$$

The base case is $T(1) \leq c$, in which $n/2^i = 1$. We have $2^i = n$ and $i = logn$. Since we know that $rc(a) = O(a^2)$, it means there exists $c_0 strc(a) \leq c_a(a^2)$. Then we have

$$4^j rc(n/2^j) \leq 4^j(c_0(n/2^j)^2) = c_0(4^j(n/2^j)^2 = c_0 n^2)$$

Hence we have :

$$\begin{aligned} T(n) &= 4^i T(n/2^i) + \sum_{j=1}^{i} 4^j rc(n/2^j) + ic' \\ &\leq cn^2 + \sum_{j=1}^{i} c_0 n^2 + ic' \\ &= cn^2 + logn(c_0 n^2) + ic' \\ &= O(n^2 logn) \end{aligned}$$

1.4 Part 4: find T(n) if rectangular copy is $O(n)$

Assuming rc(a) is the running time of Rectangular Copy on a (a x a) matrix

$$\begin{aligned} T(n) &= 4T(n/2) + 4rc(n/2) + c' \\ &= 4^2 T(n/2^2) + 4^2 rc(n/2^2) + 4rc(n/2) + c' + c' \\ &= ... \\ &= 4^i T(n/2^i) + \sum_{j=1}^{i} 4^j rc(n/2^j) + ic' \end{aligned}$$

The base case is $T(1) \leq c$, in which $n/2^i = 1$. We have $2^i = n$ and $i = logn$. Since we know that $rc(a) = O(a^2)$, it means there exists $c_0 strc(a) \leq c_a(a)$. Then we have

$$4^j rc(n/2^j) \leq 4^j(c_0(n/2^j)) = c_0(4^j(n/2^j) = c_0 n 2^j)$$

Hence we have :

$$\begin{aligned} T(n) &= 4^i T(n/2^i) + \sum_{j=1}^{i} 4^j rc(n/2^j) + ic' \\ &= cn^2 c_0 \sum_{j=1}^{i} 2^j n + ic' \\ &= cn^2 + c_0(2^{i+1} - 2)n + ic' \\ &= cn^2 + c_0(2n - 2)n + ic' \\ &= (c + 2c_0)n^2 - 2c_0 n + c' logn \\ &= O(n^2) \end{aligned}$$

2 Applications of findRankKElt

Assuming findRankKElt returns the kth smallest element of the array

2.1 K1th to K2th Smallest Elements

```
def findK1thToK2thSmallestElement(A, k1, k2):
    k1thSmallest = findRankKElt(k1)
    k2thSmallest = findRankKElt(k2)
    arrayOfElements = []
    for element in A:
        if(element >= k1thSmallest && element =< k2thSmallest):
            arrayOfElements.append(element)
    return arrayOfElements
```

2.2 Check if an element occurs more than n/2 times

We can assume that if the element occurs in the array more than n/2 times, than that element is the median of the array after it is sorted

```
# A: Array, n: Length of array
def checkIfElementOccursMoreThanHalfOfLenght(A, n):
    mid1 = findRankKElt(n/2)
    mid2 = findRankKElt(n/2 + 1)
    count1 = 0
    count2 = 0
    for element in A:
        if(mid1 == element)
            count1++
        if(mid2 == element)
            count2++

    if(count1 > n/2 || count2 > n/2):
        return True
```

2.3 Find Weighted Median

```
# A: Array, W: Weights, k1, k2
def findK1thToK2thSmallestElementWithWeights(A, W, k1, k2):
    k1thSmallest = findRankKElt(k1)
    k2thSmallest = findRankKElt(k2)
    elements = []
    weights = []
    for index, element in A:
        if(element >= k1thSmallest && element =< k2thSmallest):
            arrayOfElements.append(element)
            weights.append(W[index])
    return (arrayOfElements, weights)

# A: Array, W: Weights, k1, k2
def sumOfWeightsOfElementsBetweenk1andk2(A, W, k1, k2):
    (elements, weights) = findK1thToK2thSmallestElementWithWeights(A, W, k1, k2)
    sum = 0
    for index, element in elements:
        sum += sum + weights[index]
    return sum

#A: Array, W: weights, n: length, leftWantedSum: sum of left side needed, rightWantedSum:
def findWeightSplitElement(A, W, n, leftWantedSum, rightWantedSum, totalSum):
    k = n/2
    elementX = findRankKElt(A, k)
    weightX = weight of elementX
    leftSum = sumOfWeightsOfElementsBetweenk1andk2(A, W, 0, k)
    rightSum = totalSum - leftSum - weightX
    if(leftSum < leftWantedSum and rightSum <= rightWantedSum):
        return k
    elif(leftSum < leftWantedSum and rightSum > rightWantedSum):
        leftWantedSum = leftWantedSum - leftSum
        (A, W) = findK1thToK2thSmallestElementWithWeights(A, W, n/2, n-1)
        length = len(A)
        totalSum = rightSum
        element = findWeightSplitElement(A, W, length, leftWantedSum, rightWantedSum, total
        return element
    else:
        rightWantedSum = rightWantedSum - rightSum
        (A, W) = findK1thToK2thSmallestElementWithWeights(A, W, 0, n/2 - 1)
```

```
        length = len(A)
        totalSum = leftSum
        element = findWeightSplitElement(A, W, length, leftWantedSum, rightWantedSum, total
        return element
```

3 Find the lonely element

```
#A: Array, n: length
def findLonelyElement(A, n):
    if(n==0):
        return None;
    if(n==1):
        return A[0]

    mid = n/2
    if(mid%2 == 0):
        if(A[mid] == A[mid+1]):
            A = A[mid+2:n]
            n = len(A)
            return findLonelyElement(A, n)
        else:
            A = A[0:mid]
            n = len(A)
            return findLonelyElement(A, n)
    else:
        if(A[mid] == A[mid-1]):
            A = A[mid+1:n]
            n = len(A)
            return findLonelyElement(A, n)
        else:
            A = A[0:mid-1]
            n = len(A)
            return findLonelyElement(A, n)
```

4 Find Frequent Element

4.1 O(nlogn) Algorithm

```
#A: array, n: length
def findFrequentElement(A, n):
    if n == 1:
        return A[0]
    leftSplit = A[0:n/2]
    rightSplit = A[n/2: n-1]
    leftFrequent = findFrequentElement(leftSplit, len(leftSplit))
    rightFrequent = findFrequentElement(rightSplit, len(rightSplit))

    if(leftFrequent is None and rightFrequent is None):
        return None
    if(leftFrequent == rightFrequent):
        return leftFrequent
    if(leftFrequent is not None and rightFrequent is not None):
        leftFrequentCount = 0
        for element in A:
            if leftFrequent == element:
                leftFrequentCount+=1

        rightFrequentCount = 0
        for element in A:
            if rightFrequent == element:
                rightFrequentCount+=1
```

```
        if(leftFrequentCount > rightFrequentCount and leftFrequentCount > n/2):
            return leftFrequent
        elif(rightFrequentCount > leftFrequentCount and rightFrequentCount > n/2):
            return rightFrequent
        else:
            return None
    if(rightFrequent is not None):
        rightFrequentCount = 0
        for element in A:
            if rightFrequent == element:
                rightFrequentCount+=1
        if(rightFrequentCount > n/2):
            return rightFrequent
        else:
            return None
    if(leftFrequent is not None):
        leftFrequentCount = 0
        for element in A:
            if leftFrequent == element:
                leftFrequentCount+=1
        if(leftFrequentCount > n/2):
            return leftFrequent
        else:
            return None
```

4.2 O(n) Algorithm

```
# A: array, n: length
def findPossibleFrequentElement(A, n):
    possibleIndex = 0, count = 1
    for index, element in A:
        if(A[possibleIndex] == element):
            count+=1
        else:
            count-=1
        if(count == 0):
            possibleIndex = index
            count = 1

    return a[possibleIndex]

# A: array, n: length, frq: element to be tested
def isFrequentElement(A, n, frq):
    count = 0
    for element in A:
        if element == frq:
            count+=1

    if count > n/2:
        return True
    else:
        return False

# A: array, n: length
def findFrequentElement(A, n):
    possibleFrequent = findPossibleFrequentElement(A, n)
    if(isFrequentElement(A, n, possibleFrequent)):
        return possibleFrequent
    else:
        return None
```

5 Tetromino Cover

A chessboard has an even number of white and black squares. A T-tetronimo covers 3 of the same color and 1 other color, ie. either 3 black and 1 white or 3 white and one black. A square tetronimo covers 2 of each. As a result, after we place 15 T-tetrominoes on the board, an odd number of whites and blacks are covered. After we add the additional square, the number of whites and blacks covered are still odd which means a complete cover is not possible