Daniel Oliveros A02093272
February 8, 2018

Assignment 2: Divide and Conquer

Problem #1 Inversions in an array

(a) The inversions of the array $\{4, 2, 9, 1, 7\}$ are:

$$(4, 2), (4, 1), (2, 1), (9, 1), (9, 7)$$

(b) The array with elements from the set $\{1, 2, ..., n\}$ with the most inversions is an array sorted from the biggest to the smallest number. The number of inversions for the whole array could be calculated by adding the number of inversions each number in it has. Every element in the array has as many inversions as there are elements after it, which means we can calculate it like this:

$$\begin{aligned} I(A) &= (n-1) + (n-2) + ... + (2) + (1) + (0) \\ I(A) &= (1) + (2) + ... + (n-2) + (n-1) \\ I(A) &= \textstyle\sum_{i=1}^{n-1} i \\ I(A) &= \frac{n(n-1)}{2} \end{aligned}$$

(c) Divide-and-Conquer inversions.

1. **Algorithm Description:** This algorithm is a mild modification of Merge Sort. An inversion can be spotted in mergesort when elements from the right subarray are added before elements on the left one in the merging phase. depending on how many elements are left on the left subarray at the time of this happening, we can determine how many inversions occurred at this stage. These inversions are appended into a list, once the process is completed the values in the list are added up and returned as the total number of inversions.
2. **Pseudocode:**

```
arr=[4,3,2,1]
def Invert(A):
    c = []
    Inversion(arr, 0, len(arr)-1, c)
    val =0
    for ind in range(0, len(c)):
        val+=c[ind]
    print(val)

def Inversion(A, i, j, l):
    if(i==j):
        return
    mid = int((i+j)/2)
    Inversion(A, i, mid, l)
    Inversion(A, mid+1, j, l)
    newL = []
    a=i
    b=mid+1
    while(a<=mid and b<=j):
        if(A[a]<=A[b]):
            newL.append(A[a])
            a+=1
        else:
            newL.append(A[b])
            b+=1
            l.append(b-a-len(newL))
    if(a<mid):
```

```
        for index in range(a, mid):
            newL.append(A[index])
    else:
        for index in range(b, j):
            newL.append(A[index])
            l.append(b-a-len(newL))
    for index in range(0, len(newL)):
        A[index+i] = newL[index]

Invert(arr)
```

3. **Correctness:** In an array $a = [9, 7, 3, 5]$ there are 5 inversions. The way this algorithm can count them is based on how it tracks merge sort. First, divide into two lists $a_1 = [9, 7]$ and $a_2 = [3, 5]$. Then, each of these lists are divided into $a_{11} = [9]$, $a_{12} = [7]$, $a_{21} = [3]$, $a_{11} = [5]$. When a_{11} and a_{12} are merged, 7 is placed before 9, in this scenario it moves back one index, so we add 1 to the list of inversions. When merging a_{21} and a_{22} there are no inversions.
At this moment, $a_1 = [7, 9]$ and $a_2 = [3, 5]$, when these are merged we take both elements from a_2 first, each of these moves 2 indexes from a_1 forward. So, the inversions added in this step are 4. This method works for any sized array
4. **Time Analysis:** The operations performed in this array do not add any meaningful extra operations to Merge Sort. It therefore has the same runtime as Merge Sort. $O(Inv) = O(n \log n)$

Problem #2 Smaller of A in B.

(c) $O(nlogm)$ approach:

1. **Algorithm Description:** This algorithm works by sorting array B and performing a binary search of each element in array A into array B. The locations in which elements from array A would be put into array B are stored in an auxiliary array. This array ends up storing how many elements are directly lower than each of these elements. By adding up the array as we go through it, we can determine how many elements are smaller than each of the elements in B
2. **Pseudocode:**

```
from bisect import bisect_right
A = [30,20,100,60,90,10,40,50,80,70]
B = [60,35,73]

def aInB(A, B):

    B.sort()#mlogm complexity
    bp = [0]*len(B) #const

    for element in A:#n times. nlogm
        var = bisect_right(B,element)#logm binary search in sorted array
        if var<len(B):
            bp[var]+=1 #const

    for x in range(1, len(bp)):#m
        bp[x] +=bp[x-1]

    return bp

print(aInB(A,B))
```

3. **Correctness:** This algorithm correctly determines how many elements of array A are smaller than each element in array B. The elements originally stored in the auxiliary array are the number of elements that can be placed in that particular location in the array. Once they're all added up, we can calculate the amount of elements that are smaller than a given element.

4. **Time Analysis:**

$$\begin{aligned} O(aInB) = &\quad O(m\log m + n\log m + m + 2c) \\ O(aInB) = &\quad O(n\log m) \end{aligned}$$

Problem #3 Solving recurrences

(a) $T(n) = 2 \cdot T(\frac{n}{2}) + n^3$. Case 3 of Master Theorem was used:

$$\begin{aligned} T(n) = &\quad 2T(\tfrac{n}{2}) + n^3 \\ &\quad a = 2, b = 2, f(n) = n^3 \end{aligned}$$

Check if it meets conditions 1 and 2:

1. $f(n) = \Omega(n^{\log_b a+\epsilon})$ where $\epsilon > 0$

$$\begin{aligned} n = &\quad \Omega(n^{\log_2 2+\epsilon}) \\ n = &\quad \Omega(n^{1+\epsilon}) \end{aligned}$$

   True for any ϵ in between 0 and 2.

2. There exists a c and an n such that $af(\frac{n}{b}) \leq cf(n)$

$$\begin{aligned} 2(\frac{n}{2})^3 \leq &\quad cn^3 \\ \frac{1}{4} \leq &\quad c \end{aligned}$$

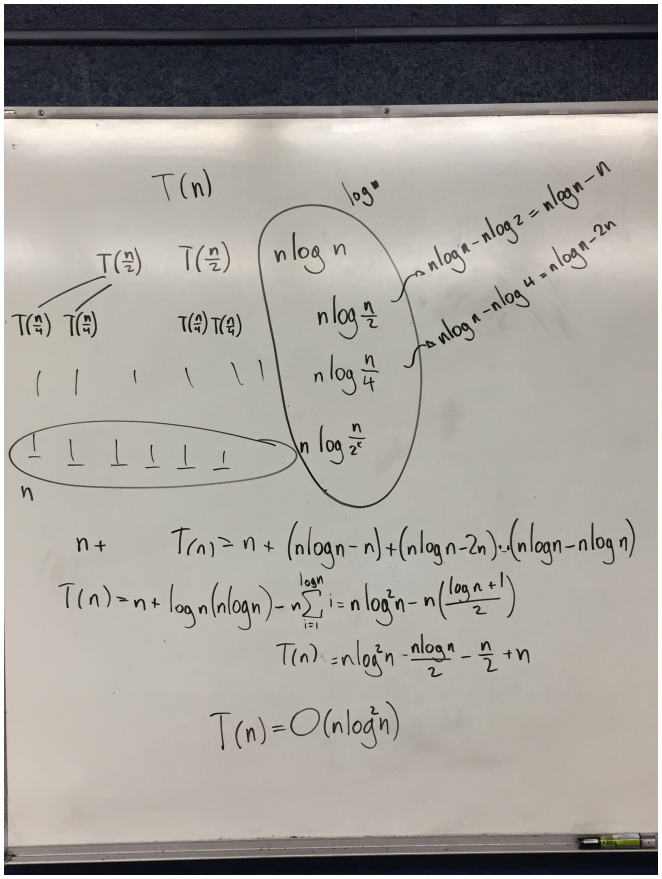
   This is true for c = 0.5 and n=1.

So, $T(n) = \Theta(n^3)$

(b) $T(n) = 4 \cdot T(\frac{n}{2}) + n\sqrt{n}$. This recurrence can be solved with the Master Theorem. It's a case 1 type recurrence. So:

$$\begin{aligned} T(n) = &\quad 4T(\tfrac{n}{2}) + n\sqrt{n} \\ &\quad a = 4, b = 2, f(n) = n^{3/2} \\ T(n) = &\quad n^{\log_2 4-\epsilon} = n^{2-\epsilon} \qquad \epsilon > 0 \\ &\qquad\qquad\qquad\qquad\quad \epsilon = 0.1 \\ T(n) = &\quad \Theta(n^{\log_2 4}) \\ T(n) = &\quad \Theta(n^2) \end{aligned}$$

(c) $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$.



(d) $T(n) = T(\frac{3}{4} \cdot n) + n$. Case 3 of Master Theorem was used:

$$\begin{aligned} T(n) = \quad & T(\tfrac{3}{4} \cdot n) + n \\ & a = 1, b = \tfrac{4}{3}, f(n) = n \end{aligned}$$

Check if it meets conditions 1 and 2:

1. $f(n) = \Omega(n^{\log_b a + \epsilon})$ where $\epsilon > 0$

$$\begin{aligned} n = \quad & \Omega(n^{\log_{3/4} 1 + \epsilon}) \\ n = \quad & \Omega(n^{0+\epsilon}) \\ n = \quad & \Omega(n^{\epsilon}) \end{aligned}$$

   True for any ϵ in between 0 and 1.

2. There exists a c and an n such that $af(\frac{n}{b}) \leq cf(n)$

$$\begin{aligned} \frac{n}{4/3} \leq \quad & cn \\ \frac{3}{4}n \leq \quad & cn \end{aligned}$$

   This is true for c = 0.8 and n=1.

So, $T(n) = \Theta(n)$

Problem #4 Buy and Sell using Divide and Conquer:

1. **Algorithm Description:** This algorithm works by first creating a difference array based on the original. It then finds the maximum subarray in this difference array using a divide and conquer approach. The value of this maximum subarray as well as the indexes its comprised within are returned and used for our output

2. **Pseudocode:**

```
#This function should take in two arrays, A and results.
#It returns the sum of the max subarray of A and stores the indexes of it in r
#using a divide and conquer approach
def max_subarray(A, r, i, j): #O(maxSub)
    if(i==j):
        return A[i]
    mid = (i+j)//2
    left = max_subarray(A, results, i, mid) #O(maxSub/2)
    right = max_subarray(A, results, mid+1, j) #O(maxSub/2)
    middle = getMidSubarr(A, results, i, j) #O(n)

    #get max of left, right, and middle. store indexes into r

def BuySell(arr):
    diff = []
    r = [None]*2
    for i in range (1, len(arr)): #O(n)
        diff.append(arr[i]-arr[i-1])
    var = max_subarray(diff, r, 0, len(diff))
    print ("Buy on day ", r[0]+1, " and sell on day ", r[1]+1, "to get a profit of", v

# Driver program to test above functions
arr = [9, 3, 5, 4, 7]
BuySell(arr)
```

3. **Correctness:** In a scenario where we have a list $a = [-5, 4, 5, -2]$ the difference array would have the values $d = [9, 1, -7]$. By finding the maximum subarray in d we get the array $[9, 1]$ with indexes $0, 1$. Adding one to these indexes reflects these locations in the original array.

4. **Time Analysis:**

$$\begin{aligned} O(BuySell) &= O(n) + O(maxSub) \\ O(BuySell) &= O(n) + (2O(maxSub/2) + O(n)) \\ &\dots \\ O(BuySell) &= O(n) + O(n \log n) \\ O(BuySell) &= O(n \log n) \end{aligned}$$

People I worked with for this assignment:

- Chris Darais
- Olivia Garrido