Muhammed ÖZKAN 151044084

CSE-321 Introduction to Algorithm Design

Fall 2020 Homework 4

1.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	text
0	**0**	**1**	0																	
	0	**0**	**1**	0																
		0	**0**	**1**	0															
			0	**0**	**1**	0														
				0	**0**	**1**	0													0010
					0	**0**	**1**	0												Pattern size=4
						0	**0**	**1**	0											
							0	**0**	**1**	0										m=3
								0	**0**	**1**	0									
									0	**0**	**1**	0								
										0	**0**	**1**	0							
											0	**0**	**1**	0						
												0	**0**	**1**	0					
													0	**0**	**1**	0				
														0	**0**	**1**	0			
															0	**0**	**1**	0		
																0	**0**	**1**	0	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	......................n

Number of Comparisons																	
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	**O(m*(n-m))**

(in terms of n)

The algorithm repeatedly makes 3 comparison and shifts by 3.

Total number of comparisons: **3(n-3).**   **O(m*(n-m))**

If the length of the input pattern is 3 (3 bits), there may be two options.

a. **001** if the algorithm starts comparing from left.
b. **100** if the algorithm starts comparing from right.

2.

```
Start node: A
['A', 'B', 'C', 'D', 'E', 'A'] path length= 22
['A', 'B', 'C', 'E', 'D', 'A'] path length= 25
['A', 'B', 'D', 'C', 'E', 'A'] path length= 18
['A', 'B', 'D', 'E', 'C', 'A'] path length= 24
['A', 'B', 'E', 'C', 'D', 'A'] path length= 16
['A', 'B', 'E', 'D', 'C', 'A'] path length= 19
['A', 'C', 'B', 'D', 'E', 'A'] path length= 24
['A', 'C', 'B', 'E', 'D', 'A'] path length= 22
['A', 'C', 'D', 'B', 'E', 'A'] path length= 18
['A', 'C', 'D', 'E', 'B', 'A'] path length= 19
['A', 'C', 'E', 'B', 'D', 'A'] path length= 18
['A', 'C', 'E', 'D', 'B', 'A'] path length= 27
['A', 'D', 'B', 'C', 'E', 'A'] path length= 24
['A', 'D', 'B', 'E', 'C', 'A'] path length= 21
['A', 'D', 'C', 'B', 'E', 'A'] path length= 16
['A', 'D', 'C', 'E', 'B', 'A'] path length= 16
['A', 'D', 'E', 'B', 'C', 'A'] path length= 22
['A', 'D', 'E', 'C', 'B', 'A'] path length= 25
['A', 'E', 'B', 'C', 'D', 'A'] path length= 16
['A', 'E', 'B', 'D', 'C', 'A'] path length= 15
['A', 'E', 'C', 'B', 'D', 'A'] path length= 21
['A', 'E', 'C', 'D', 'B', 'A'] path length= 21
['A', 'E', 'D', 'B', 'C', 'A'] path length= 27
['A', 'E', 'D', 'C', 'B', 'A'] path length= 22
```

Since all nodes are connected to each other, it is okay to start from any node. Starting from any node will not change the shortest path. When we select the node A as the starting point, the brute force algorithm tries all possible ways and gives us the shortest path to **15**.

Shortest route= [15, ['A', 'E', 'B', 'D', 'C', 'A']]

3.

The value of *$\log_2 n$* for a given number *n* is equal to *x* , where *$2^x=n$* . To compute the value of *$\log_2 n$* , divide the number by 2 recursively, that is, decrease the number by half till a threshold value is obtained.

The result should be the floor value of the *$\log_2 n$* .

```
function logBaseTwo(int n):
    if n = 1
        return 0
    else
        return 1 + logBaseTwo(n/2)
```

$$T(n) = 0 \quad for\ n = 1$$

$$T(n) = 1 + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) \quad for\ n > 1$$

$$T(n) = \begin{cases} 0 & for\ n = 1 \\ 1 + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) & for\ n > 1 \end{cases}$$

Use master theorem to solve the above problem. Here a = 1 and b = 2. As per the master theorem

$$\begin{aligned} n^{\log_b a} &= n^{\log_2 1} \\ &= n^0 \qquad \because \ \log_2 1 = 0 \\ &= 1 \end{aligned}$$

Since, $T(n) = \Theta(n^{\log_b a})$, the case2 of master's theorem is applicable i.e., $T(n) = \Theta(\log_2 n)$

Therefore, the total efficiency of the algorithm is $\boldsymbol{T(n) = \Theta(\log_2 n)}$

4.

Incorrectly manufactured bottle problem. We can solve this problem by using decrease-by-factor-2 algorithm. Because we can use the factory scale to find the incorrectly bottle. Therefore, we can find the bottle or bottle groups that are different from the factory scale by measuring, and focus on them in the next step and find the incorrectly produced bottle. Here is the algorithm

```
procedure findIncorrectlyBottle(list,scale)
        if list has 1 bottle then.
                return The bottle of different weight.
        end if

        if list has 2 bottles then.
                return One of the two bottles has a different weight.
        end if

        divide list to 2 list: A, B

        if sum(A) > (lenght(A) * scale) then. Measurement is done here.
                findIncorrectlyBottle(A,scale).  Find the different one among them.
        else then
                findIncorrectlyBottle(B,scale).
end procedure
```

Implementation didn't specified as required for this problem. In this implementation; when it finds the coin it just shows that it's found. It's not given any indexes or etc. for the sake of user.

Analysis of complexities.

Best Case: In the best case scenario, Until it’s found incorrectly bottle always remains in the B. We are saved from only more check. Which makes our recurrence relation as: $T(n) = T(n/2) + 1$. Using the master theorem: $T_{best}(n) \in \boldsymbol{\theta(\log n)}$.

Worst Case: In the worst case scenario, Until it’s found incorrectly bottle always remains in the A. Which makes our recurrence relation as: $T(n) = T(n/2) + 2$. Using the master theorem: $T_{worst}(n) \in \boldsymbol{\theta(\log n)}$.

Average Case: $T_{best}(n) = T_{worst}(n) \Longrightarrow \boldsymbol{T_{avg} \in \theta(\log n)}$.

5.

From given two unsorted arrays A and B, we will design a divide-and-conquer algorithm that finds xth element. Here is the implementation of the algorithm:

```
def xth_element(A, B, m, n, xth,flag):
    if flag==1:
        A.sort()
        B.sort()
    if m > n:  # switch arrays to make m <= n
        return xth_element(B, A, n, m, xth,0)
    if m == 0:
        return B[xth - 1]
    if xth == 1:
        return min(A[0], B[0])

    i = min(m, xth // 2)
    j = min(n, xth // 2)
    if A[i - 1] > B[j - 1]:
        return xth_element(A, B[j:], m, n - j, xth - j,0)
    else:
        return xth_element(A[i:], B, m - i, n, xth - i,0)
```

This algorithm assumes the given x is valid for the arrays and sorts the arrays given unsorted when the flag is 1 and continues to process. This sorting process is done only once. And x is between 1 and m + n. Meaning that first element of merged arrays is x = 1. In this algorithm, in each recursion we decide which array to go on for search and then divide our problem regarding to that. If the index of A has reached to 0, we just return the x^{th} element from B. Or if the x^{th} became 1, we return the minimum element among the first elements of A and B. Other than base cases, we get the i and j as the minimum one among x^{th} and their m and n indices respectively. After finding the i and j, we check which item is bigger on that indices in A and B. Regarding to solution we either decrease the A or B, then continue to searching.

Worst Case: Our worst case happens when union of A and B are not homogeneous. Since the complexity of sequencing algorithms is below the complexity of these operations, the effect on the result is minimal.

This will cause, divide operation to continue until both of the arrays become empty. It will cause A to be divided log m times, and B to be divided log n times. Hence, T(m, n) ∈ **O(log m + log n)**