CST 370 – Spring A 2020
Homework 4

Name: Sara Kazemi
Class ID: 4444

1. (10 points) Consider the Master Theorem as we covered in the lecture.

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$
If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

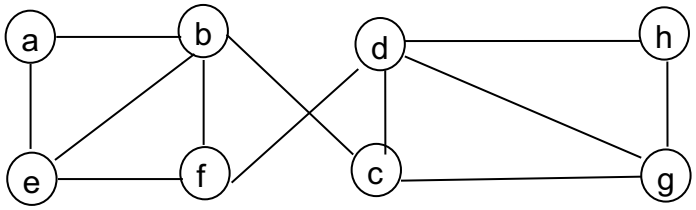
Based on the theorem, determine the time efficiency of the following formula T(n).

(a) $T(n) = 2 * T(n/2) + n^4$
Since, a = 2, d = 4, and $b^d = 2^4 = 8$, then $a < b^d$
Therefore, the time efficiency of T(n) $\in$ **$\Theta(n^4)$**

(b) $T(n) = 16 * T(n/4) + n^2$
Since, a = 16, d = 2, and $b^d = 4^2 = 16$, then $a = b^d$
Therefore, the time efficiency of T(n) $\in$ **$\Theta(n^2 \log n)$**

You need to indicate the values for a, b, and d. Also, indicate which case of the Master Theorem applies.

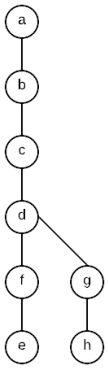
2. (10 points) Consider the following graph.



Starting at vertex ***a***, traverse the graph using the **depth-first search** (DFS) algorithm. After that, you should present a **mark array** and a **depth-first search tree** with only tree edges. For the algorithm, you have to use **our convention** of the class (= **ascending order for the alphabetical characters**).

Mark Array **DFS Tree**

a	1
b	2
c	3
d	4
e	6
f	5
g	7
h	8



3. (20 points) (a) Assume that you are going to develop an algorithm to find the position of the largest number in an array with *n* integer numbers using **a divide-and-conquer technique**. For example, if your algorithm has an input array such as **1, 3, 11, 7, 5, 6, 4, 9,** your algorithm should return 2 because the index 2 has the largest number (= 11) in the array. Describe the basic idea of your divide-and-conquer algorithm **in English**.

To find the index of the largest number in an array of *n* numbers $a_n \ldots a_{n-1}$ with a recursive divide-and-conquer technique, we firs must consider the base case where the size of the array is **n=1**. In this case, we would return index 0 as the position of the largest number in the array because it is the *only* number in the array.

If **n >1**, we divide the problem into two subproblems. The first subproblem will determine the position of the largest number in the first n/2 numbers in the input array. The second subproblem will determine the position of the largest number in the remaining n/2 numbers. The two subproblems will be solved using the same divide-and-conquer technique so that the sub arrays continue to be divided into smaller and smaller subarrays recursively until the base case is reached in both subproblems. The result of the base case will return a single integer position, which will be stored to a variable.

Since each recursive call to the divide-and-conquer algorithm divides the problem into two subproblems, we will get two integer positions back (one for the first n/2 numbers in the array and another for the second n/2 numbers in the array). Then we can compare the values at the two positions of the original array and return the position of the largest of the two. This process would continue for each recursive call of the algorithm that had been added to the call stack until we get to the original call, whereupon we should get the position of the largest number in the entire array.

(b) Based on the basic idea of (a), **write a pseudocode** of your divide-and-conquer algorithm. Note that the array index of your algorithm should start from zero.

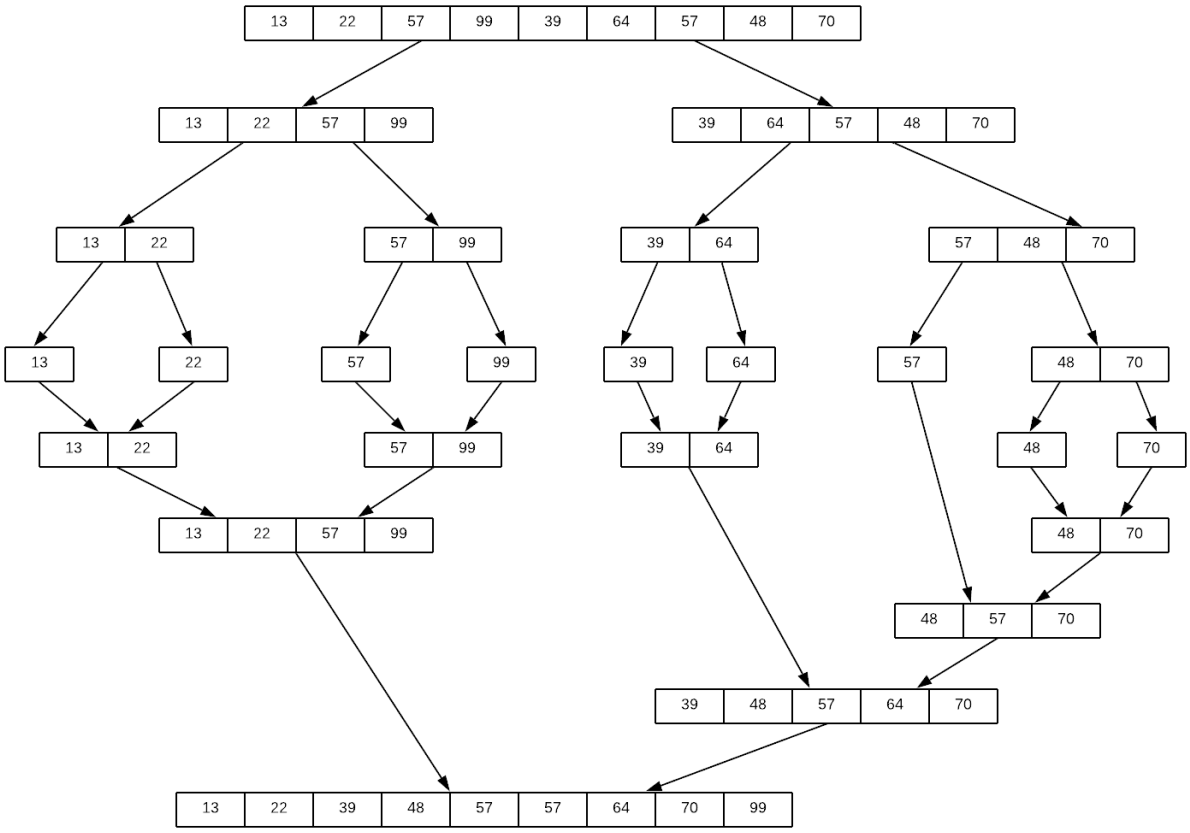
```
ALGORITHM getLargestDNC(A[0…n-1])
// Returns index (an integer) of the largest value in an array via recursive divide-and-conquer technique
// Input: An array A[0…n-1] of integers
// Output: An integer representing the index of the largest number in the input array
  if n = 1 return 0 // base case, if the array size is 1 return position 0
  else
     copy A[0…⌊n/2⌋ - 1] to B[0…⌊n/2⌋  -1]
     copy A[⌊n/2⌋...n - 1] to C[0… ⌈n/2⌉  -1]
     largestB ← getLargestDNC(B[0…⌊n/2⌋  -1])
     largestC ← getLargestDNC(C[0… ⌈n/2⌉  -1])
     if A[largestB] > A[largestC] return largestB
     else return largestC
```

(c) Determine the time complexity of your algorithm. Set up a recurrence relation and apply the Master Theorem to calculate the time complexity. So, you should provide the values of symbols "a", "b", and "d". And then provide the time efficiency of your algorithm

$$T(n) = aT(n/b) + f(n) \quad \text{where } f(n) \in \Theta(n^d), \quad d \geq 0$$

T(n) = 2 * T(n/2) + 1
Since, a = 2, d = 0, and $b^d = 2^0 = 1$, then $a > b^d$
Therefore, the time efficiency of $T(n) \in \Theta(n^{\log_2 2}) \in \Theta(n)$

4. (5 points) Using the Mergesort pseudocode in our textbook, sort the numbers *13, 22, 57, 99, 39, 64, 57, 48, 70* in the ascending order. You should describe your answer as Figure 5.2 of our textbook. Note that our textbook's example has 8 numbers. However, **this problem has 9 numbers**. Thus, you should read the pseudocode carefully and identify how it sorts the 9 numbers. *Note that the second "copy" statement of the Mergesort algorithm in the textbooks has a typo: for odd number of elements, the second copy should have one more element than the first copy.*



5. (5 points) Consider the quicksort algorithm covered in the class. Present the **first partitioning operation** for the list "**6 10 13 5 8 3 2 11** ", You should use the first number, 6, as a pivot for the partitioning. In your answer, you have to present the indexes i and j clearly.

0	1	2	3	4	5	6	7	Indices
6	10^{i}	13	5	8	3	2	11^{j}	Initial configuration. 0^{th} element is the pivot and index i is at position 1 (value is 10) and index j is at position 7 (value is 11).
6	10^{i}	13	5	8	3	2^{j}	11	The value at index i 10 >= the pivot value 6, so we proceed to check the value at index j. The value at index j 11 !<= the pivot value 6, so we decrement the index j (shift it to the left).
6	2^{i}	13	5	8	3	10^{j}	11	Now that we have shifted index j to the 6^{th} index, the value at index j 2 <= the pivot value 6. Now we **swap** the values at index i and index j.
6	2	13^{i}	5	8	3	10^{j}	11	Since i is not >= j, we go back to evaluating the value at index i and must increment i.
6	2	13^{i}	5	8	3	10^{j}	11	The value at index i is 13 >= the pivot value 6, so we proceed to check the value at index j. The value at index j 10 is not <= the pivot value 6, so we decrement j.
6	2	13^{i}	5	8	3^{j}	10	11	Now the value at index j is 3 which is <= the pivot value 6, so we will swap the values at indices i and j.
6	2	3^{i}	5	8	13^{j}	10	11	Since i is not >= j, we go back to evaluating the value at index i and must increment i.
6	2	3	5^{i}	8	13^{j}	10	11	The value at index i 5 is <= the pivot value 6, so we proceed to evaluating the value at index j. Since the value at index j 13 is not <= the pivot value 6, we decrement the index j.
6	2	3	5^{i}	8^{j}	13	10	11	The value at index j 8 is not <= the pivot value 6 so we decrement the index j.
6	2	3	5^{ij}	8	13	10	11	The value at index j 5 <= the pivot value 6 so we swap the values indices i and j. This has no effect since the indices are the same. We'd also re-swap the values since i >= j. But, again, this would have no effect. After this, we swap the pivot with the value at index j.
5	2	3	$\mathbf{6}^{ij}$	8	13	10	11	The next step would be to repeat the process on the subarray **5, 2, 3** with 5 as the pivot. and the subarray **8, 13, 10, 11** with 8 as the pivot. The value 6 is in its final sorted position.

6. (15 points) Write a program called ***select-sort-k.cpp*** that modifies the selection sort algorithm to sort the first *k* smallest elements of the array with *n* elements (k <= n, the value of k will be entered by the user). Your program should read the integer numbers from an input file, store them in an array (dynamic), and display the k smallest elements in sorted order. Your algorithm must run in O(n*k) time. When you write the program, don't forget to include "Title", "Abstract", "ID (A four-digit number)", "Name", and "Date".

Sample input file:

This is the content of **t1.txt**. Note that the **first line** (10) indicates the number of integers in the file.

```
10
4
6
8
15
20
22
10
3
9
2
```

Sample Run 1:

Enter a file name: **./t1.txt**

Enter the value k: **4**

Output: 2, 3, 4, 6

Sample Run 2:

Enter a file name: **./t1.txt**

Enter the value k: **6**

Output: 2, 3, 4, 6, 8, 9

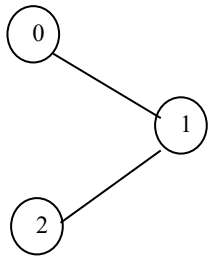
7. (20 points) Write a C++ program called **BFS.cpp** that implements the ***Breadth-First Search (BFS) algorithm***. Your program should read an input file name and a starting node from a user. After that, your program should display the list of vertices visited. In the problem, you can assume that the number of vertices in the input file is less than or equal to 25. When you write the program, don't forget to include "Title", "Abstract", "ID (A four-digit number)", "Name", and "Date".

[**Hint**: Since the maximum number of vertices is less than or equal to 25, you can use a simple array-based queue for the BFS algorithm. Or you can use a STL queue library. For more info, refer to http://www.cplusplus.com/reference/queue/queue/pop/]

Input file format: This is a sample input file called **`t1.txt`**.

```
3
0 1 0
1 0 1
0 1 0
```

The first line (= 3 in the example) indicates that there are three vertices in the graph. For the homework, we can assume that the first vertex starts from the number 0. Thus, **t1.txt** describes a graph like below:



One blank space is used to delimiter the data. Note that there's no blank space at the end of each line. **If your program does not read the file properly, your program will get no credit.**

This is a sample run:

```
Enter filename: ./t1.txt
Enter a start vertex: 0
BFS order: 0 -> 1 -> 2
```

In the program, your program has to **follow our convention (= ascending order)**.

This is another sample input file called **t2.txt**.

```
5
0 0 1 0 0
0 0 1 0 0
1 1 0 1 1
0 0 1 0 0
0 0 1 0 0
```

This is a sample run:

```
Enter filename: ./t2.txt
Enter a start vertex: 0
BFS order: 0 -> 2 -> 1 -> 3 -> 4
```

8. (15 points) Write a C++ program called **assignment.cpp** that solves the *Assignment Problem* in Chap 3.4. Your program should read the assignment costs of each person from a user and determine the best assignment. In the program, you can assume that the number of all jobs is less than 15. To get the full credits, your program should display all combinations of assignments. However, the sequence of combinations to be displayed on the screen is not important.

This is a sample run:

```
Enter number of jobs: 2
```

```
Enter assignment costs of 2 persons:
Person 1: 2 5
Person 2: 4 6
Output:
Permutation 1: <1, 2> => total cost: 8
Permutation 2: <2, 1> => total cost: 9

Solution: <1, 2> => total cost: 8
```

This is another sample run

```
Enter number of jobs: 3
Enter assignment costs of 3 persons:
Person 1: 2 5 3
Person 2: 4 6 9
Person 3: 7 2 4

Permutation 1: <1, 2, 3> => total cost: 12
Permutation 2: <1, 3, 2> => total cost: 13
Permutation 3: <2, 1, 3> => total cost: 13
Permutation 4: <2, 3, 1> => total cost: 21
Permutation 5: <3, 1, 2> => total cost: 9
Permutation 6: <3, 2, 1> => total cost: 16

Solution: <3, 1, 2> => total cost: 9
```