Written Questions

Q1.

a) Well documented pseudocode of a recursive algorithm that solves this problem and prints a protocol of all disc movements, e.g., “disc 7 moved from A to B”.

Algorithm MoveDisk(n, start, spare, end)

Input : n is size of tower of disk (positive integer), start is where it starts from, spare is where it store, end is the last destination (Note that user will put start as A, spare as B, end as C).

Output: Void

if n == 1 then

    Print (disk 1 moved from start to spare) (a)

    Print (disk 1 moved from spare to end) (b)

else

    MoveDisk(n-1, start, spare, end) (c)

    Print (disk n moved from start to spare) (d)

    MoveDisk(n-1, end, spare, start) (e)

    Print (disk n moved from spare to start) (f)

    MoveDisk(n-1, start, spare, end) (g)

b) A hand-run of your algorithm for n=3

Check small notation for each algorithm on peusdo code. Note that I denote feg A as start, B as spare, C as end. Additionally, I am not able to move directly A to C. For simplicity, I will simply say disk n, A -> B format.

1) It goes to (c) since n is not 1, once n = 2, MoveDisk(2-1,A,B,C) is running and it will print

(a) and (b), disk 1, A -> B and disk 1, B->C. (c) is done so print (d), disk 2, A->B. And then run (e), print disk 1, C->B and B->A ((a) and (b)). (f) print, disk 2, B->C and (g) runs, disk 1, A->B and B->C.

(To sum up, disk 1, A -> B disk 1, B -> C disk 2, A -> B disk 1, C -> B disk 1, B -> A disk 2, B -> C disk 1, A -> B disk 1, B -> C)

2) parameter n=3 for (c) (which is MoveDisk(3-1,A,B,C) is done. (d) print, disk 3, A->B. (e) for 2 will run. It changes MoveDisk (2, C, B, A). for (c), it becomes MoveDisk (1, C, B, A), print disk 1, C->B, and disk 1, B->A (note that disk 1 stayed in C). Print (d) disk 2, C->B and MoveDisk (1, A,B,C) will print disk 1, A->B, and disk 1, B->C. print (f) disk 2, B->A and (g) runs, print disk 1, C->B and disk B->A.

(To sum up, disk 3, A -> B disk 1, C -> B disk 1, B -> A disk 2, C -> B disk 1, A -> B disk 1, B -> C disk 2, B -> A disk 1, C -> B disk 1, B -> A)

3) parameter n=3 for (e) is done (which is MoveDisk(3-1,C,B,A), (f) print, disk 3, B->C. (g) for 2 is running. This is exactly the same result as 1) (disk 1, A -> B disk 1, B -> C disk 2, A -> B disk 1, C -> B disk 1, B -> A disk 2, B -> C disk 1, A -> B disk 1, B -> C).

(To sum up, disk 3, B -> C disk 1, A -> B disk 1, B -> C disk 2, A -> B disk 1, C -> B disk 1, B -> A disk 2, B -> C disk 1, A -> B disk 1, B -> C)

In this way, the rule is well followed, and All the components are moved to A to C, with largest number of disk in the bottom and smallest number of disk in the top.

c) Formulation and calculation of its complexity using the big-Oh notation. Justify your answers.

$T(n) = k + 3T(n-1) = k + 3k + 9T(n-2) = k + 3k + 9k + ... + 3^{n-1}*k$

Therefore, by geometric law $(1-r^n)/(1-r)$, it becomes $k*(3^n-1)/2$ and Big-Oh is $O(3^n)$, $k*(3^n-1)/2 <= c3^n$ where $c > C$ and $n>=1$, $n_0 = 1$.

Q2. Answer the following questions:

a) In class, we discussed a recursive algorithm for generating the power set of a set. In this assignment, you will develop a well-documented pseudocode that generates all possible subsets of a given set T (i.e. power set of T) containing n elements with the following requirements: your solution must be non-recursive, and must use a stack and a queue to solve the problem. For example: if T = {2, 4, 7, 9} then your algorithm would generate: {}, {2}, {4}, {7}, {9}, {2,4}, {2,7}, {2,9}, {4,7}, {4,9}, {7,9}, …etc. (Note: your algorithm's output need not be in this order).

```
Algorithm PowerSet(set T):

Input: set of T.

Output : power set's elements of set T

if (T is empty) Then

        P[0] = T

        return P // if it's empty set, the powerset is only empty itself so return the same thing.

Else

        Create new Queue Q, which has element set of T // to use queue

        Create empty Stack S // to use stack

        Create empty set P // power set (Which will be returned)

        Pushing S from dequeue of Q

        P[0] <- {}

        P[1] <- S.pop() // because previous if condition, we know that Q is not empty so there is something in stack  .

        for i = 1 to size of Q // note that size of Q decreased from dequeuing.
```

```
            Pushing S from dequeue of Q // push stack from queue
            temp <- S.pop()
            for j = 0 to size of P-1
                  P[j+size of P] = P[j] ∪ temp // it will keep accumulating previous
power set with union of new element
      return P
```

Explanation:

Since the question asks to use stack and queue, I use both of them but actually, in my algorithm, I can use either stack or queue. If given set consists of empty set, as commented, it will go to if condition.

If given set consists of 1 element of set, for loop is not running so the result will be empty set and element itself (by P[0] is {} and P[1] is S.pop()).

If elements of set are more than 1, for loop is working. To show it properly, let's say input is {A,B,C}. Queue becomes = {A,B,C} and P becomes {{},{A}} while A is dequeued. And then for loop is working i = 1 to 2 (size of Q became 2 since we dequeued one element.). Q = {C} and S = {B}. Another for loop make a combination of S. Therefore, P = {{},{A},{B},{A,B}} by union of temporary value (which pop from S).

Lastly, Q becomes empty and S = {C}. another for loop is running and P becomes {{},{A},{B},{A,B},{C},{A,C},{B,C},{A,B,C}}. In question, powerset does not need to be in order so the result is good.

b) Calculate the time complexity of your algorithm using the big-Oh notation. Show all calculations.

Until while loop, all operations are constant. I denote it as C_1. For the first for loop (for i = 0 to size of Q -1), it takes n times since size of Q is same as size of input (set). There is another while loop inside of that, there is for loop. The others are all running constant time

multiply n times. I denote this as C_2n.

Finally, for i = 0 to size of P-1, it depends on the size of P and P is changeable. However, start size of P is 2 and keep going 2*2, 2*2*2… etc. Since previous loop runs n times, this loop will run 2 + 2*2 + 2*2*2+ … + 2^{n-1} = $2(2^{n-1}-1)$. I denote this part as $C_3(2^n-2)$. For example, for case of {A,B,C}, since I denote P[0] and P[1] as constant time, while loop runs 2 + 4 times which is 6 = 2(2^(3-2)-1).

In total, $C_1+C_2n+ C_3(2^n-2) \leq c*2^n$ where $c = C_1+C_2+C_3$ and $1 \leq n$ ($n_0 = 1$). Therefore Big-Oh is $O(2^n)$.

Q3. Rank the following functions in non-decreasing order (≤) according to their big-Oh complexities and justify your ranking:

n^3+logn, log(log(n)), $\sqrt{n}$, n!, n/2, $\binom{n}{2}$, 2^n, nlogn, n^n, $2^{\log(n)}$, $2^{n!}$, 2^(2^n)

$n^3+\log n \leq cn^3$ where $2 \leq c$, and $n \geq 1$ or $n_0=1$. Big-Oh is $O(n^3)$

log(log(n)), Big-Oh is simply O(log(log(n))

$\sqrt{n}$ is $n^{1/2}$ and we can simply notate $O(n^{1/2})$ (or other form, $O(n^c)$ where $0<c<1$)

n! is n*(n-1)*…2*1. We know that it is smaller than n^n but it is greater than 2^n. Simply, n*n*n*…*n > n*(n-1)*…2*1 > 2*2*2*…*2*2. (All multiplying n times)

n/2 is O(n) but very precisely, it is less complexity than n (since n/2 < n and even though it has same big-Oh)

$\binom{n}{2}$ is nC2 which means n(n-1)/2. It is $O(n^2)$ where $n(n-1)/2 \leq cn^2$ $c \geq 1$, $n \geq 1$.

2^n is $O(2^n)$ and priority is discussed above.

nlogn is O(nlog(n))

n^n is $O(n^n)$ and priority is discussed above.

$2^{\log(n)}$=n. As mentioned, n/2>n but for big-Oh notation, they are the same

$2^{n!}$ is $2^{n*(n-1)*\ldots*2*1}$ on the other hand, $2^{(2^n)} = 2^{2*2*2*\ldots*2}$. As we discussed above, $n! > 2^n$.

Comparing n^n and $2^{(2^n)}$, $2^{(2^n)} > n^n$ regardless of value n because triple exponential is larger than exponential of exponential (where its derivate increases). We can show that even if base n is larger than 2, comparing n and 2^n, 2^n is much more drastically increasing. Simply, n = 1, it's 1<4 but just let n = 4, $4^4 < 2^{16} = 4^8$. When n = 8, $8^8 = 2^{24} < 2^{256}$ which is a really huge difference.

$O(\log(\log(n)) \leq O(n^{1/2})$ since logarithmic time takes less than fractional power time.

$O(n^{1/2}) \leq O(n)$ since $n^{1/2} \leq n$ where n is integer.

$O(n) \leq O(n\log n) \leq O(n^2)$, finding its derivation, $1 < \log(n) + 1 < 2n$

$O(n^3) \leq O(2^n)$, quadratic time takes less than exponential time.

Therefore, the ranking is following,

$\log(\log(n)) \leq \sqrt{n} \leq n/2 \leq 2^{\log(n)} \leq n\log n \leq \binom{n}{2} \leq n^3+\log n \leq 2^n \leq n! \leq n^n \leq 2^{(2^n)} \leq 2^{n!}$

In terms of big-oh notation, in a same order,

$O(\log(\log(n)) \leq O(n^{1/2}) \leq O(n) \leq O(n) \leq O(n\log n) \leq O(n^2) \leq O(n^3) \leq O(2^n) \leq O(n!) \leq O(n^n) \leq O(2^{(2^n)} \leq O(2^{n!})$.