Question #7

```
function findMaxDiscount(stores, currentSet):
    if store is empty:
        return calc_discount(currentSet)

    maxDiscount = 0
    bestSet = None

    for each store in stores:
        newSet = current + [store]
        remainingStore = stores - store
        discount = findMaxDiscount(remainingStores, newSet)

        if(discount) maxDiscount:
            maxDiscount = discount
            bestSet = newSet

    return bestSet

initial stores = [list of all stores]
maxDiscountSet = findMaxDiscount(initial stores, [])
```

Base case: when n=0 (no stores left) the function simply calculates and return the discount for the current set. we can consider this a constant time operation $O(1)$, so $T(0)=1$

Recursive case: when there are n stores to consider the function iterates over each store removes it from the list and makes a recursive call with $n-1$ stores. For each recursive call it also performs some constants-time operations (like adding a store to current set calculating the discount and comparing discounts).

The recursive relation can be formulated as:

$T(n) = n \cdot T(n-1) + O(1)$

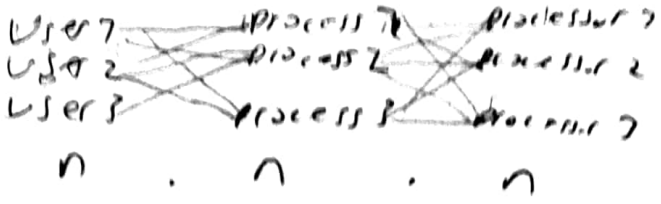
$F(2) = 2 \cdot F(1)$
$F(1) = 1 \cdot F(0)$
$F(0) = 1$

$F(n) = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \cdot F(0)$

$F(n) = n! \, F(0)$

$F(n) = n!$

Question #2




```
Function findminimal-cost (Users[], Processes[], Processors[]):

    For user each in Users
      For processor each in processors
       For process each in process
          tmp_cost= Calculate_totalcost(user, process, processor)
          if(tmp-cost < min cost)
              min_cost = tmp_cost
              min_cost_user = user
              min_cost_process = process
              min_cost_processor = processor

    return min_cost_user, min_cost Process, min_cost_Processor
```

Timing Analysis:

Each for require n times. all for is time n^3

There aren't average, best, worst case. Only Timing analysis $O(n^3)$

//

Question #3

n! All Point permutations

```
Function findMinimumEnergySequence(ports):
    min_energy = Infinity
    min_sequence = []
    For each sequence in allPermutations(ports): // return list of n! items
        energy = calculateEnergy(sequence)
        If energy < min_energy:
            min_energy = energy
            min_sequence = sequence

    return min_sequence, min_energy

Function allPermutations(ports): // Generate all possible permutations of the ports
    return list using recursive approach
Function calculateEnergy(sequence): // Calculate the total energy cost
For a given sequence

Ports = [List of n ports]

optimal_sequence, energy_cost = findMinimumEnergySequence(ports)
```

Worst-case complexity: The worst-case scenario occurs when the algorithm has to explore each possible sequence. Since we are generating all permutations of the ports the time complexity is O(n.n!) where n is number of ports. This is because there are n! permutations for n ports

Best-case complexity: The best-case scenario would still require generating all permutations so the best-case complexity remains O(n.n!)

Average-case complexity: On average the complexity would still revolve around the permutations of the ports, so it also stands at O(n.n!)

```
function all_permutation(ports)

if len(ports) <= 1:
    return [ports]

else
    all_sequences = []
    for i in range(len(ports)):
     current_port = ports[i:i] + ports[i+1:]
     for sequence in generate_all_possible_sequences(remaining_ports)
         all_sequences.append([current_port] + sequence)

    return all_sequences
```

Question #4

```
Function minCoins(coins, target):
  minCoins[target] = INF
  minCoins[0] = 0

  for i from 1 to target:
    for j from 0 to size(coins) - 1:
      if coins[j] <= i:
        subResult = minCoins[i - coins[j]]
        if subResult != INF and subResult + 1 < minCoins[i]:
          minCoins[i] = subResult + 1

  return minCoins[target]
```

$T(n) = n^2 \cdot T(n-1)$

1) Initialize an Array 'minCoins' to keep track of minimum number of coins required for each value up to the target amount. Set 'minCoins[target]' to infinity and minCoins[0] to 0 since zero coins are needed to make zero change

2) Using two nested loops iterate through the values from 1 to the target amount and iterate through each coin denomination.

3) For each value 'i' and each coin denomination 'j', check if the the coin denomination 'coins[j]' is less than or equal to current value 'i', if it is calculate the subresult by accessing 'minCoins[i - coins[j]]'.

4) if the sub-result is not infinity (meaning it's a valid value) and adding one coin to it would result in smaller number of coins than the current minimum for 'minCoins[i]' update 'minCoins[i]' to the new smaller value (subResult+1)

5) Finally return 'minCoins[target]' which contains the minimum number of coins need to make up exact target amount of change.

Time complexity :

1. The outer loop from 1 to the target value.
2. Inner loop iterates through the avaible coin denominations.

The outer loop runs for up to the target value. so it executes 'target' times. meanwhile, the inner loop operates based on the number of available coin denominations, which is represented by size(coins).

Average best and worst case

$T(n) = n^2 \cdot T(n-1)$

Question #5

In function findMinMax we have two recursive call for each call to itself, each time with half of the array. This gives us the following reccurence relation.

$T(n) = 2T(\frac{n}{2}) + T(n)$

- $T(n)$ is the time complexity of the function when called with an array of length n.
- The term $2T(\frac{n}{2})$ represent the time time complexity of the two recursive calls on half of the array.
- $T(n)$ represent the time for all other operations within the function call, excluding the recursive calls.

Simplify Recursion Relation

$T(n) = 2T(\frac{n}{2}) + C$

$T(n) = aT(\frac{n}{b}) + f(n)$

$a \geq 1$ $b \geq 1$ $f(n)$ asymtotically positive function

in our case $a = 2$ $b = 2$ $f(n) = C$ (C is $O(1)$)

$O(n^{\log_b a}) = O(n^0) = O(1)$ we fall in to case 2 of the master theorem where $f(n)$ is $\Theta(n^{\log_b a})$

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

$T(n) \begin{cases} c_1 & \text{if } n = 1, \\ 2T(\frac{n}{2}) + c_2 & \text{if } n > 1, \end{cases}$