VO TRAN QUANG TUAN
18127248
18127248@student.hcmus.edu.vn

Homework Assignment 2
Introduction To Design And Analysis Of Algorithms, Fall 2020

2020-11-12

Contents

1 Problem 1

Analyze the complexity of the function $Permutation()$ presented in Chapter 3: Brute-Force technique (Exhaustive Search section).
Pseudo code:

Algorithm Generate Permutations of Set A

```
1: procedure PERMUTATIONS(pivot, a[1...n])        ▷ Generate the permutations of a, with a pivot
2:     if pivot == n then
3:         PRINT(a);
4:     else
5:         for i = pivot, ..., n do
6:             SWAP(a[pivot], a[i])
7:             PERMUTATIONS(pivot + 1, a)
8:             SWAP(a[pivot], a[i])
9:         end for
10:    end if
11: end procedure
```

Before analyzing the time complexity of $Permutations$ algorithm, we must determine some details:

- The size of input: an pivot (don't care about that, it is an integer represents the position of each array's element to call the recursion), array input of n numbers. So the size of input is n.
- Basic operations: the swap function between $a[pivot]$ and $a[i]$, also $a[i]$ and $a[pivot]$.
- The arrangement of input is not important because it doesn't affect to the time executes algorithm.

Let's denote T(n) is the number of times the basic operation is executed when running this algorithm.
Clearly, with $pivot = n$, the algorithm doesn't execute any swapping, so $T(1) = 0$.
And what happen when $pivot! = n$, slightly, we can see that this algorithm executes the for loop. In the for loop, we have two swapping and a recursion function involved. So with the for loop from $i = 1$ to $i = n$, we have $2 \cdot n$ swapping and $2 \cdot n \in \Theta(n)$, and $n \cdot T(n-1)$ times for involving the recursion:

$$T(n) = n \cdot T(n-1) + \Theta(n)$$
$$T(1) = 0$$

To simplifying the $2 \cdot n$, we change it to n, because they belong to $\Theta(n)$.

$$\begin{aligned} T(n) &= n \cdot T(n-1) + n \\ &= n \cdot (n-1) \cdot T(n-2) + n \cdot (n-1) + n \\ &= n \cdot (n-1) \cdot (n-2) \cdot T(n-3) + n \cdot (n-1) \cdot (n-2) + n \cdot (n-1) + n \\ &\dots \\ &= \frac{n!}{(n-i)!} \cdot T(n-i) + \frac{n!}{(n-1)!} + \frac{n!}{(n-2)!} + \dots + \frac{n!}{(n-i)!} \end{aligned}$$

With the initial condition: $T(1) = 0$, let $n - i = 1$:

$$\begin{aligned} T(n) &= \frac{n!}{(n-i)!} \cdot T(n-i) + \frac{n!}{(n-1)!} + \frac{n!}{(n-2)!} + \dots + \frac{n!}{(n-i)!} \\ &= \frac{n!}{1!} \cdot 0 + \frac{n!}{(n-1)!} + \frac{n!}{(n-2)!} + \dots + \frac{n!}{(1)!} \\ &= n! \cdot [\frac{1}{(1)!} + \frac{1}{(2!)} + \dots + \frac{1}{(n-1)!}] \end{aligned}$$

Taylor's Theorem: with $x \to 0$

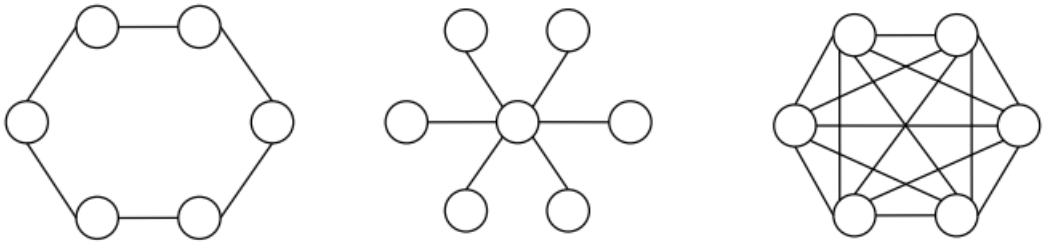
$$e^x \approx 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!}$$

So:

$$T(n) \geq n! \cdot e \implies T(n) \in \Omega(n!)$$

2 Problem 2

A given boolean matrix $G[1..n, 1..n]$, where $n > 3$, which is supposed to be adjacency matrix for a graph modeling a network with one of three topologies: Ring, Star and Fully connected mesh.



Ring *Star* *Fully connected mesh*

Obiviously, the adjacency matrix illustrate the connection between all vertices in a graph. each cell in this matrix represents number of edges between two particular vertex in graph correspond with the index of row and index of col in adjacency matrix.
Let's concentrate in the adjacency matrices which correspond with three kinds of graph in the picture above. Clearly, we can infer some comments from it:

- With the Ring topology, degree of each vertex is exactly 2. So the sum of all value in each row or each column of the adjacency matrix is 2.
- With the Star topology, degree of each vertex is exactly 1 except an center vertex has degree $n-1$ with n is numbers of vertex in this graph. So the sum of each row or each column ($n-1$ row or $n-1$ column except 1 column/1 row) in the adjacency matrix is 1, except 1 row/1 column has sum is $n-1$.
- With the fully connected mesh, clearly, we can see that each row/each column in the adjacency matrix has sum is $n-1$, because each vertex fully connects to the other vertices.

Following the analyzing above, we can design an brute-force algorithm for determine which kinds of topologies base on the adjacency matrix.
Pseudo code:

Algorithm Which Topologies?

```
1:  procedure SUM(A[1...n])
2:      S ← 0
3:      for i = 1, ..., n do
4:          S+ = A[i]
5:      end for
6:      return S
7:  end procedure
8:
9:  procedure SPECIFYTOPOLOGY(G[1...n, 1...n])      ▷ Specify the kind of topology corresponds with the
    adjacency matrix G
10:     i ← 1
11:     if SUM(G[i]) == 2 then                                                    ▷ G[i] is row
12:         return Ring
13:     end if
14:     if SUM(G[i]) == n − 1  and SUM(G[i + 1])== n − 1 then
15:         return Fully connected mesh
16:     else
17:         return Star
18:     end if
19: end procedure
```

Let's analyzing the time complexity of this algorithm. To begin with the Sum function, it calculates the sum of an array, so the time complexity of this function at each call step is $f(n) = n \in \Theta(n)$.
Next, we consider to the $SpecifyTopology$ function. The basic operation is in if conditions. Let $g(n)$ is the time complexity, with a specific adjacency matrix, all the if conditions in this algorithm were executed. So $g(n) = n + n + n = 3 \cdot n \in \Theta(n)$. So the time complexity is linear.

3 Problem 3

Partition Problem: In number theory and computer science, the partition of problem is the task of deciding whether a given set S of n positive integers can be partitioned into two subset S_1 and S_2 such that the sum of the number in S_1 equals the sum of the numbers in S_2. With brute-force approach, a simple solution is that we can generate all the subset of the given set.
Pseudo code:

Algorithm 1 Generate all Subset of Set A

```
1:  procedure SUBSET(a[1...n])
2:      Set ← {∅}
3:      for i = 1, ..., n do
4:          len ← LENGTH(Set)
5:          for j = 1, ..., len do
6:              temp ← Set[j]
7:              temp ← temp ∪ a[i]
8:              Set ∪ temp
9:          end for
10:     end for
11:     return Set
12: end procedure
```

After that, we calculate the sum of each subset and compare with the remaining set when use the except operator between the given set and current subset.
Pseudo code:

Algorithm 2 Which set satisfies Partition Problem?

```
procedure PARTITIONPROBLEM(a[1...n])
    Set ← INITIALSET(a)
    ListSubSet ← SUBSET(a)
    for SubSet in ListSubSet do
        if SUM(SubSet) == SUM(Set \ SubSet) then
            return true
        end if
    end for
    return false
end procedure
```

Let's analyzing the time complexity of this algorithm. Firstly, looking at the $SubSet(a)$ function, we can see that:

- The basic operation is the second assign operation in the for loop. Particularly, it is $temp \leftarrow temp \cup a[i]$.
- Let $f(n)$ is the time complexity of this function:
  - $i = 1, j = 1 \implies$ loop $1 = 2^0$ time.
  - $i = 2, j \in [1, \ldots, 2^1] \implies$ loop 2^1 times.
  - $i = 3, j \in [1, \ldots, 2^2] \implies$ loop 2^2 times.

    $\ldots$
  - $i = n, j \in [1, \ldots, 2^{n-1}] \implies$ loop 2^{n-1} times.
- So, the time complexity:

$$f(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \in \Theta(2^n)$$

Next, considering the $PartitionProblem(a)$ function, we can see that:

- The basic operation is in the if condition in the for loop.
- Let $g(n)$ is the time complexity of this function, clearly $g(n) \in \Theta(n^2)$.

In conclusion, the time complexity for this partition problem with brute-force approach is $\Theta(max\{n^2, 2^n\}) = \Theta(2^n)$

4 Problem 4

A magic square of order n is an arrangement of n^2 numbers (from 1 to n^2) in a square, such that the n numbers in all rows, all columns and both diagonals sum to the same constant t.

4.1 Problem 4.a

Prove that $t = \frac{n \cdot (n^2+1)}{2}$.
We have an arrangement of n^2 numbers (from 1 to n^2), so the sum all of them is $\frac{n^2 \cdot (n^2+1)}{2}$.
But with the property of magic square, we have n rows with the same sum, so:

$$t = \frac{n^2 \cdot (n^2+1)}{2 \cdot n} = \frac{n \cdot (n^2+1)}{2}$$

4.2 Problem 4.b

Generate all magic squares of order n with brute-force approach, we can generate all the permutations of the array $a[1 \dots n^2]$, and check all sum in all rows, columns and diagonals. Compare with the t in problem $4.a$. If existing one sum different from t, return false, else return true.
Firstly, Generate all the permutations of the array $a[1, \dots, n^2]$:
Pseudo code:

Algorithm 1 Generate Permutations of Set A

```
1:  procedure PERMUTATIONS(pivot, a[1...n], Set)        ▷ Generate the permutations of a, with a pivot
2:      if pivot == n then
3:          Set ∪ a                                     ▷ Set contains all permutaions of a
4:      else
5:          for i = pivot, ..., n do
6:              SWAP(a[pivot], a[i])
7:              PERMUTATIONS(pivot + 1, a)
8:              SWAP(a[pivot], a[i])
9:          end for
10:     end if
11: end procedure
```

Secondly, calculate sum of row, column, diagonal and compare with t:
Pseudo code:

Algorithm 2 Compare sum of each row, column, diagonal with t

```
procedure CompareSum(A[1 ... n^2])
    Matrix ← CreateMatrix(A)
    for i = 1, ..., n do                                  ▷ Compare sum of row with t
        if Sum(Matrix[i]) ≠ t then
            return false
        end if
    end for

    SumCol ← 0
    for i = 1, ..., n do                                  ▷ Compare sum of column with t
        for j = 1, ..., n do
            SumCol ← SumCol + Matrix[j][i]
        end for
        if SumCol ≠ t then
            return false
        end if
        SumCol ← 0
    end for

    Principal ← 0
    Secondary ← 0
    for i = 1, ..., n do                                  ▷ Compare sum of two diagonals with t
        for j = 0, ..., n do
            if i == j then
                Principal ← Principal + Matrix[i][j]
            end if
            if (i + j) == (n − 1) then
                Secondary ← Secondary + Matrix[i][j]
            end if
        end for
    end for
    if Principal ≠ t or Secondary ≠ t then
        return false
    end if
    return true
end procedure
```

Finally, checking each permutations of this array satisfy the property of magic square, if true, print the array. Pseudo code:

Algorithm 3 Generate all magic squares of order n

```
procedure GENERATEMAGICSQUARE(n)
    if n == 1 then
        PRINT(1)
    end if
    if n == 2 then
        PRINT(∅)
    end if
    A ← INITIALARRAY(n)                    ▷ Initial array A with element belong to 1 . . . n^2
    Set ← {}
    PERMUTATIONS(1, A, Set)
    for per in Set do
        if COMPARESUM(per) then            ▷ Satisfy the property of magic square?
            PRINT(per)
        end if
    end for
end procedure
```

Let's analyzing the time complexity of this algorithm for this problem, let $f(n)$ is the time complexity of $GenerateMagicSquare$. With $Permutations$ function, we have $(n^2)!$ array, the time complexity is $\Omega((n^2)!)$, with the $Comparesum$ function, the time complexity is $\Theta(n^2)$. Looking carefully to the for loop in $GenerateMagicSquare$, we can see that the for execute $(n^2)!$ times. So:

$$f(n) = n^2 \cdot (n^2)! \in \Theta(n^2 \cdot (n^2)!)$$