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

Homework Assignment 1

Introduction To Design And Analysis Of Algorithms, Fall 2020

2020-11-03

Contents

1 Exercise 01

Proof the theorem:

$$f_1(n) \in O(g_1(n)) \land f_2 \in O(g_2(n)) \implies f_1(n) + f_2(n) \in O(max\{g_1(n), g_2(n)\})$$

From the definition of big-O notation:

$$f_1(n) \in O(g_1(n)) \iff f_1(n) \leq c_1 \cdot g_1(n), \forall n \geq n_1$$
$$f_2(n) \in O(g_2(n)) \iff f_2(n) \leq c_2 \cdot g_2(n), \forall n \geq n_2$$

Explicitly, we can see that:

$$f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n)$$
$$f_1(n) + f_2(n) \leq c_1 \cdot max(\{g_1(n), g_2(n)\}) + c_2 \cdot max(\{g_1(n), g_2(n)\}), \forall n \geq max(n_1, n_2)$$
$$f_1(n) + f_2(n) \leq (c_1 + c_2) \cdot max(\{g_1(n), g_2(n)\}), \forall n \geq max(n_1, n_2)$$
$$f_1(n) + f_2(n) \in O(max\{g_1(n), g_2(n)\})$$

The theorem is proved.

2 Exercise 02

Analyze the complexity of the improvement Bubble Sort:

```
void BubbleSort(int a[1..n], int n)
{
    bool flag = true;
    int m = 1;
    while (flag)
    {
        flag = false;
        m++;
        for (int j = n; j >= m; j++) {
            if (a[j-1] > a[j]) {
                swap(a[j-1], a[j]);
                flag = true;
            }
        }
    }
}
```

The size of input: n (the number of elements in array needs to be sorted).
Basis operation: the comparison between a[j-1] and a[j] in if condition.
This algorithm depends on the distribution of the data in array.

2.1 The best case

The best case happens when the array was sorted in desired order.
Let $B(n)$ is the time complexity of this algorithm in the best case, we can see that $B(n) = n - 1 \in \Theta(n)$.

2.2 The worst case

The worst case happens when the array was sorted in the inverse order than the desired order, or the max element appears in the begin of the array.

Let analyze the complexity of this case, look at the for loop in this case, when the if condition is true, the flag is true, so the while loop is true.

Basis operation is the comparison in the if condition, traverse every for loop and while loop, we can see that:

- $m = 2 : j \in [2 \ldots n] \implies$ loop $n - 1$ times.
- $m = 3 : j \in [3 \ldots n] \implies$ loop $n - 2$ times.
  $\ldots$
- $m = n : j = n \implies$ loop 1 times.

Let $W(n)$ is the time complexity:

$$W(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$

2.3 The average case

Suppose the possible which algorithm stop after each iteration is the same. So, the probability of this is $\frac{1}{n-1}$.

Let C_i is number of time operates the comparison (basis operation in this algorithm in for loop) after the i^{th} iteration (in while loop):

- $C_1 = n - 1$ time operates the comparison.
- $C_2 = n - 2 + n - 1 = 2 \cdot n - 3$ time operates the comparison.
  $\ldots$
- $C_i = n - i + \ldots + n - 1$
  $\ldots$

- $C_{n-1} = 1 + ... + n - 1 = \frac{n\cdot(n-1)}{2}$ time operates the comparison.

Let $A(n)$ is time complexity:

$$A(n) = \frac{1}{n-1} \cdot (n - 1 + 2 \cdot n - 3 + ... + \frac{n \cdot (n-1)}{2}) \in \Theta(n^2)$$

3 Exercise 03

Analyze the complexity of the Insertion Sort:

```
void InsertionSort(int a[1..n], int n)
{
    for(int i = 2; i <= n; i++)
    {
        v = a[i];
        j = i-1;
        while(j >= 1 && a[j] > v)
        {
            a[j+1] = a[j];
            j--;
        }
        a[j + 1] = v;
    }
}
```

The size of input: n (number of elements in array)
The basis operation: the comparison between $a[j]$ and v in while loop condition
This algorithm depends on the distribution of data in array.

3.1 The best case

The best case happens when the array was sorted in desired order.
Let $B(n)$ is the time complexity of this algorithm in the best case, we can see that $B(n) = n - 1 \in \Theta(n)$.

3.2 The worst case

The worst case happens when the array was sorted in the inverse order than the desired order.

Let analyze the complexity of this case, look at the for loop and while loop in this case, we can see that:

- $i = 2 : j = 1 \implies$ loop 1 times.
- $i = 3 : j \in [1 \dots 2] \implies$ loop 2 times.
- $i = 4 : j \in [1 \dots 3] \implies$ loop 3 times.
  $\dots$
- $i = n - 1 : j \in [1 \dots n - 2] \implies$ loop $n - 2$ times.
- $i = n : j \in [1 \dots n - 1] \implies$ loop $n - 1$ times.

Let $W(n)$ is the time complexity of this algorithm in the worst case:

$$W(n) = 1 + 2 + 3 + \dots (n - 2) + (n - 1) = \frac{n \cdot (n - 1)}{2} \in \Theta(n^2)$$

3.3 The average case

Suppose every number distinct each other, the probability of inserting to i position is the same, it is $\frac{1}{i}$.

Suppose we have $i = 4$:

- Insert to the 4^{th} position: 1 comparison $\cdot \frac{1}{4}$
- Insert to the 3^{th} position: 2 comparison $\cdot \frac{1}{4}$
- Insert to the 2^{th} position: 3 comparison $\cdot \frac{1}{4}$
- Insert to the 1^{th} position: 3 comparison $\cdot \frac{1}{4}$

General:

- Near the last position: $\frac{i-1}{i}$

- The last position: $\frac{i-1}{i}$

Let $C(i)$ is the average of number of comparison to insert a_i:

$$C_i = \frac{i-1}{i} + \sum_{j=1}^{i-1} \frac{j}{i}$$

Let $A(n)$ is the time complexity of this algorithm in the average case:

$$A(n) = \sum_{i=2}^{n} C_i \approx \frac{n^2 - n}{4} + n - ln(n) - \gamma \approx \frac{n^2}{4} \in \Theta(n^2)$$

4 Exercise 04

Given an array of n integers. Design an algorithm with the time complexity $\Theta(n^2)$ to find a subsequence (or a contiguous subarray) with largest sum of numbers in the array. If all integer numbers in array is negative, we must return 0

4.1 Design the algorithm

Using two for loop to traverse the array. The first loop indicates the start of subsequence, the second loop indicates the length and compares sum of this subsequence (subsequence is being considered) with the current max sum and update the start point (using for tracing the subsequence after find max sum of array), update the length (using for tracing), and update the max sum.

Tracing the subsequence: using a start position and length of subsequence for traverse from the original array.

The result of subsequence which has max sum in original array has a format:

$$RES[0..N+1]$$

With $RES[0]$ is length of this subsequence, $RES[N+1]$ is sum of this subsequence, $RES[1..N]$ is the content of this subsequence.

4.2 Pseudo code

```
PROCEDURE Tracing(A[0..n-1], LengthOfResult, TraceIndex)
    if LengthOfResult = 0
        return NULL
    declare RES[0..LengthOfResult+1]
    RES[0] <- LengthOfResult
    for i <- TraceIndex to (TraceIndex + Length):
        RES[i-TraceIndex + 1] <- A[i]
    return RES

ALGORITHM FindSubsequenceWithMaxSum(A[0..n-1])
    maxSum <- 0, tempSum <- 0
    lengthOfResult <-0, traceIndex <- 0;
    for i <- 0 to n:
        tempLengthOfResult <- 0
        tempTraceIndex <- i
        for j <- 0 to n:
            tempSum <- (tempSum + A[j]);
            INCREASE lengthOfResult;
            if tempSum > maxSum:
                UPDATE maxSum <- tempSum
                UPDATE lengthOfResult
                UPDATE traceIndex
    RES <- Tracing(A, lengthOfResult, traceIndex)
    //Update maxSum to RES
    RES[lengthOfResult + 1] <- maxSum;
    return RES
```

4.3 Analyze the complexity

The size of input: n (number of elements in array).
With tracing, time complexity: $g(n) \in \Theta(n)$
Basis operation: Comparison between tempSum and MaxSum in the inner loop.
We have two for loop, and: $i : j \in [i, n-1]$

- $i = 0 : j \in [0, n-1] \implies$ loop n times
- $i = 1 : j \in [1, n-1] \implies$ loop $n-1$ times
  $\ldots$
- $i = n-1 : j = n-1 \implies$ loop 1 times

The time complexity:

$$f(n) = n + (n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n+1)}{2} \in \Theta(n^2)$$

So:

$$f(n) + g(n) \in \Theta(n^2) + \Theta(n) \iff f(n) + g(n) \in \Theta(n^2)$$

This algorithm doesn't depend on the distribution of data in array.

4.4 Implement in C++

Tracing the subsequence from input array, trace index and length of the subsequence:

```cpp
// tracing the subsequence.
int* tracing(const int* a, const int& resLength, const int& traceIndex) {
    if (resLength == 0) return nullptr;
    int* resArray = new int[resLength + 2];
    // the first element of resArray is length of array
    resArray[0] = resLength;
    // tracing the subsequence from the testArray
    for (size_t i = traceIndex; i < traceIndex + resLength; i++)
    {
        // subsequence traced from a and trace index
        resArray[i-traceIndex+1] = a[i];
    }
    return resArray;
}
```

Finding the subsequence in array, which have max sum of all elements in it:

```cpp
// find the max sum and subsequence.
int* findMaxSum(const int* a, const int& n) {
    int maxSum = 0;
    // temporatory sum
    int tmpSum = 0;
    // temporary trace index
    int tmpTraceIndex = 0;
    // trace index;
    int traceIndex = 0;
    // length of subsequence
    int resLength = 0;
    for (size_t i = 0; i < n; i++)
    {
        // array's length
        int tmpResLength = 0;
        tmpTraceIndex = i;
        tmpSum = 0;
        for (size_t j = i; j < n; j++)
        {
            tmpSum += a[j];
            tmpResLength += 1;
            if (tmpSum > maxSum) {
                // update max sum
                maxSum = tmpSum;
                //update length of subsequence
                resLength = tmpResLength;
                //update trace index
                traceIndex = tmpTraceIndex;
            }
        }
    }

    int* resArray = nullptr;
    resArray = tracing(a, resLength, traceIndex);
    if (resArray != nullptr) {
        resArray[resLength + 1] = maxSum;
    }
    return resArray;
}
```