Algorithm Design and Analysis

วิชาบังคับก่อน: 204251 และ 206281

ผู้สอน: ตอน 1 อ. เบญจมาศ ปัญญางาม เรียน ห้อง 201
ตอน 2 อ. ดร. จักริน ชวชาติ เรียน ห้อง 209

บทที่ 3
สัญลักษณ์แสดงขีดจำกัด (Asymptotic Notation)
Part II

Asymptotic notation :Properties

Properties

- Transitivity : example
  - f(n)∈Θ(g(*n*)) and g(n) ∈ Θ(h(*n*)) imply f(n) ∈ Θ(h(*n*))
- Reflexivity
  - f(n) ∈ Θ(f(*n*)) , f(n) ∈ O(f(*n*)), f(n) ∈ Ω(f(*n*))
- Symmetry
  - f(n) ∈ Θ(g(*n*)) iff g(n) ∈ Θ(f(*n*))
- Transpose symmetry
  - f(n) ∈ O(g(*n*)) iff g(n) ∈ Ω(f(*n*))
  - f(n) ∈ o(g(*n*)) iff g(n) ∈ ω(f(*n*))

Important Notes

- For analysis we try to give upper bound(O) and lower bound(Ω) and average running time(Θ).
- From the above example, it should also be clear that, for a given function(algorithm) getting upper bound(O) and lower bound(Ω) and average running time(Θ) may not be possible always.
- We concentrate on upper bound (O) because knowing lower bound of an algorithm is of no practical importance and we use (Θ) notation if upper bound and lower bound are same.

Why is it called Asymptotic analysis

- From the above discussion, we can easily understand that, in every case for a given function f(*n*) we are trying to find other function g(*n*) which approximates f(n) at higher values of *n*.
- That means g(*n*) is also a curve which approximates f(*n*) at higher values of *n*.
- In mathematics we call such curve as asymptotic curve. In other terms, g(*n*) is asymptotic curve for f(*n*).
- For this reason, we call analysis as asymptotic analysis.

Analyzing the time complexity

- Counting Primitive Operations
- Assumed to take a constant amount of time : $\mathbf{O}(1)$
- Sequential Instruction
- Use smallest possible class of function
  - 2n is $\mathbf{O}(n)$ instead of 2n is $\mathbf{O}(n^2)$

- $\Theta(n)+ \Theta(n)= \Theta(n)$,
- $\Theta(n)+ \mathbf{O}(n) = \Theta(n)$
- $\Theta(n)+ \mathbf{O}(n^2)= \mathbf{O}(n^2)$

Analyzing the time complexity

- If ...(t_1)... then ...(t_2)... else ...(t_3)... : t_1, t_2, t_3
  - $t = t_1 + \min(t_2,t_3) \leq t \leq t_1 + \max(t_2,t_3)$
- Loop : for , while
  - Depends on input size and/or number of repetition
- Recursive call : $t = t_1 + t_2$
  - t_1= Time consuming for recursive call tasks
  - t_2= other tasks

Analyzing the time complexity

To reduce expression

Define: f1(n)= O(g1(n)) and f2(n) ∈ O(g2(n))

1. $cf1(n) = c\mathbf{O}(g1(n)) = \mathbf{O}(g1(n))$
2. $f1(n) + f2(n) = \mathbf{O}(\max(g1(n),(g2(n)))$
3. $f1(n) \cdot f2(n) = \mathbf{O}(g1(n) \cdot (g2(n))$
4. $f1(n)\ k = \mathbf{O}(g1(n)k)$
5. $\sum_{k=1}^{n} O(f(k)) = O(\sum_{k=1}^{n} f(k))$

Analyzing the time complexity

- Example 1: An algorithm that finds the maximum element in a finite sequence of array A
- Input: data array A[1..n]
  Output: the largest element

$T(n) = c_1 + c_2\ n + c_3\ (n-1) + c_4$
$= (c_2 + c_3)\ n + (c_1 + c_4 - c_3)$
$= O(n)$

Find_Max(A)

1. max := A[1] O(1)
2. for i := 2 to n
3. if max < a[i] then max := A[i]
4. return(max) O(1)

$\sum_{i=2}^{n} O(1) = O(\sum_{i=2}^{n} 1)$

- T(n) = O(max(1+ n + 1) = O(n)
- Using O(g1(n))+ O(g2(n))= O(max(g1(n),(g2(n)))

Analyzing the time complexity

- Example 2: Selection sort
- Input : data array A[1..n]
  Output : sorted data array A

T(n)= n +(n-1)+$\sum_{i=2}^{n} i$ +$\sum_{i=2}^{n}(i-1)$ +(n-1)
= n +(n-1)+ n(n+1)/2-1+n(n-1)/2+(n-1)
T(n)= $O(n^2)$

Selection_Sort(A)

```
1. for i = n downto 2
2.     j=1                                   O(1)
3.     for (k = 2; k <= i; k++)              O(i)
4.         if (A[j] < A[k] ) j = k;          O(1)
5.     swap(A,i,j)                           O(1)
```

max(O(1),O(i),O(1))= O(i)

T(n)= $\sum_{i=2}^{n} O(i) = O(\sum_{i=2}^{n} i)$ = O(n(n+1)/2-1) = $O(n^2)$

Analyzing the time complexity

- Example3

```
1. for i=1 to n
2.     A[i]=0
3. for i=1 to n
4.     for j=1 to n
5.         A[i]=A[i]+A[j]
```

$\sum_{i=1}^{n} O(1) = O(\sum_{i=1}^{n} 1)$ = O(n)

$\sum_{i=1}^{n} O(n) = O(\sum_{i=1}^{n} n)$ = $O(n^2)$

= O(n)

T(n)=max(O(n),$O(n^2)$)=$O(n^2)$

Analyzing the time complexity

1. O(k) = O(1), 10k = O(1) for any constant k
2. $N^2 / 2 - 3N = O(N^2)$
3. 1 + 4N = O(N)
4. $7N^2 + 10N + 3 = O(N2)$
5. $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
6. log N + N = O(N)
7. $\sum_{i=1}^{N} i = O(N^2) ==> \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$
8. $\sum_{i=1}^{N} i^2 = O(N^3) ==> \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$

Analyzing the time complexity : Home Work

ทบทวน Algorithm สำหรับปัญหา Sorting

- Selection Sort $O(n^2)$
- Insertion Sort $O(n^2)$
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort

Analyzing the time complexity : Practice & Solution

- 1. คำนวณหา Time Complexity

```
int Function2_1( const int A[ ], int N )
 {      int sum, MaxSum, i, j, k;
/* 1*/ MaxSum = 0;
/* 2*/ for( i = 1; i <= N; i++ )
/* 3*/         sum = 0;
/* 4*/         for( j = i; j <= N; j++ )
/* 5*/               sum += A[ j ];
/* 6*/               if (sum>MaxSum )
/* 7*/                     MaxSum = sum;
/* 8*/ return MaxSum;
}
```


Analyzing the time complexity : Practice & Solution

- 2. คำนวณหา Time Complexity

```
int  CALLEDFUNC2_2(int A [], int n)
{  unsigned int i,sum ;
/*1*/    sum = 0;
/*2*/    for (i = 1; i<= n; i = i*2)
/*3*/          A[i-1]= i*i*i;
/*4*/          Sum=sum+A[i-1];
/*5*/    return sum;
}
void  Function2_2(int A[], int n)
{  unsigned int i,sum ;
/*6*/    sum = CALLEDFUNC2_2(A, n);
/*7*/    for (i = 1; i< n; i++)
/*8/         printf("%d +",A[i-1]);
/*9*/    printf("%d = %d",A[n-1],sum);
}
```