UCS1302:
DATA STRUCTURES

Algorithm analysis

Session Meta Data

Author	**Dr. B. Bharathi**
Reviewer	
Version Number	1.2
Release Date	23 June 2019

Revision History

Revision Date	Details	Version no.
22 September 2017	1. New SSN template applied	1.2

Session Objectives

- Understanding the concepts of algorithm analysis
- Running time calculations

Session Outcomes

- At the end of this session, participants will be able to
  - Understand algorithm analysis
  - Calculate the running time of an algorithm

Agenda

- Introduction
- Algorithm analysis
- Growth rate
- Asymptotic notation
- Running time calculation

Introduction to Algorithm analysis

Dr. B. Bharathi

SSNCE

June 23, 2019

Introduction

What is Algorithm?

•a clearly specified set of simple instructions to be followed to solve a problem

- ➢Takes a set of values, as input and
- ➢ produces a value, or set of values, as output

•May be specified

- ➢In English
- ➢As a computer program
- ➢As a pseudo-code

• Data structures

Methods of organizing data

•Program = algorithms + data structures

Introduction

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

Order of growth

- Any algorithm is expected to work fast for any input size.
- For smaller input size our algorithm will work fine but for higher input size the execution time is much higher
- By increasing the size of n (input size) we can analyze how well our algorithm works.
- Let input size, n=5 and we have to sort the list of elements for e.g. 25,29,10,15,2

Order of growth

- So for n=5 our algorithm will work fine but what if n=5000?
- So our algorithm will take much longer time to sort the elements or cause small delays to give the result
- So how the behavior of algorithm changes with the no. of inputs will give the analysis of the algorithm and is called the ***Order of Growth***

Order of growth

- For calculating the order of growth we have to go for higher value of n, because

➢ as the input size grow higher algorithm makes delays.
➢ for all real time applications we have higher values for n.

Example: Selection Problem

- Given a list of N numbers, determine the *k*th largest, where k ≤ N.
- Algorithm 1:
  (1) Read N numbers into an array
  (2) Sort the array in decreasing order by some simple algorithm
  (3) Return the element in position k

Example: Selection Problem

- Algorithm 2:

  (1) Read the first k elements into an array and sort them in decreasing order

  (2) Each remaining element is read one by one

  - If smaller than the kth element, then it is ignored
  - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.

  (3) The element in the kth position is returned as the answer.

Example: Selection Problem

- Which algorithm is better when
  - N =100 and k = 100?
  - N =100 and k = 1?
- What happens when N = 1,000,000 and k = 500,000?
- There exist better algorithms

Algorithm analysis

- We only analyze *correct* algorithms
- An algorithm is correct
  - If, for every input instance, it halts with the correct output
- Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer
- Analyzing an algorithm
  - Predicting the resources that the algorithm requires
  - Resources include
    - Memory
    - Computational time (usually most important)

Algorithm analysis

- Factors affecting the running time
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - typically, the *input size* (number of items in the input) is the main consideration
      - E.g. sorting problem ⇒ the number of items to be sorted
      - E.g. multiply two matrices together ⇒ the total number of elements in the two matrices

Efficiency of the algorithm

- There are three cases
  - Best case
  - Worst case
  - Average case

Efficiency of the algorithm

- Worst-case running time of an algorithm
  - The longest running time for **any** input of size n
  - An upper bound on the running time for any input

    $\Rightarrow$ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
- Best-case running time
  - sort a set of numbers in increasing order; and the data is already in increasing order
- Average-case running time
  - May be difficult to define what “average” means

Asymptotic Notations

- Algorithms perform f(n) basic operations to accomplish task
  - Identify that function
  - Identify size of problem (n)
  - Count number of operations in terms of n

Execution time

- Time computer takes to execute f(n) operations is cf(n)
- where c
  - depends on speed of computer and
  - varies from computer to computer

Development of Notation

- Not concerned with small values of n
- Concerned with VERY LARGE values of n
- Asymptotic – refers to study of function f as n approaches infinity

- Example: $f(n) = n^2+4n + 20$

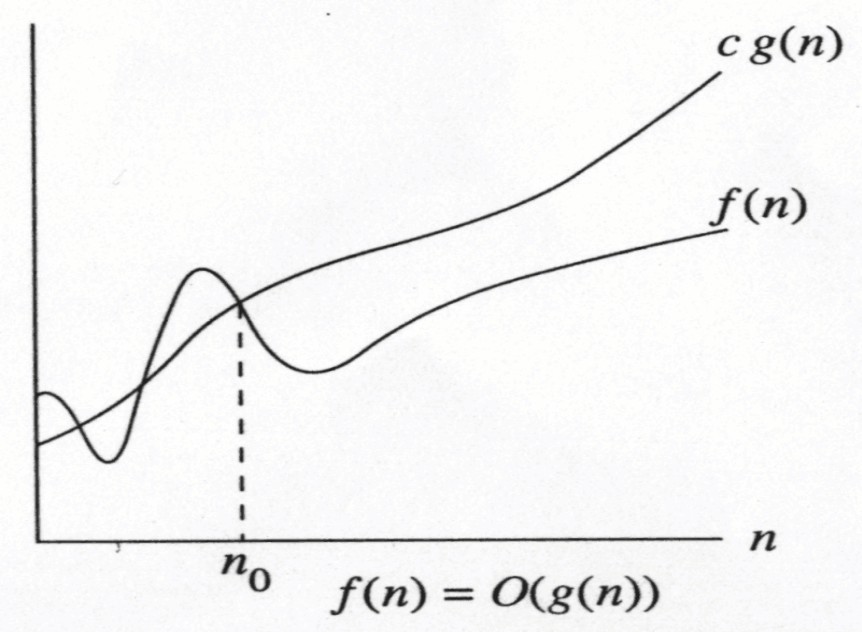
  n^2 is the dominant term and the term 4n + 20 becomes insignificant as n grows larger

Development of Notation

- Drop insignificant terms and constants
- Say function is of $O(n^2)$ called Big-O of n^2

- Common Big-O functions in algorithm analysis
  - g(n) = 1 (growth is constant)
  - g(n) = $\log_2 n$ (growth is logarithmic)
  - g(n) = n (growth is linear)
  - g(n) = $n \log_2 n$ (growth is faster than linear)
  - g(n) = n^2 (growth is quadratic)
  - g(n) = 2^n (growth is exponential)

Growth Rate



- The idea is to establish a relative order among functions for large n
- $\exists$ c , $n_0 > 0$ such that $f(N) \leq c\, g(N)$ when $N \geq n_0$
- f(N) grows no faster than g(N) for "large" N

Asymptotic notation: Big-Oh

- f(N) = O(g(N))
- There are positive constants c and n_0 such that

  $f(N) \leq c\ g(N)$ when $N \geq n_0$

- The growth rate of f(N) is *less than or equal to* the growth rate of g(N)
- g(N) is an upper bound on f(N)

Big-Oh: example

- Let $f(N) = 2N^2$. Then
  - $f(N) = O(N^4)$
  - $f(N) = O(N^3)$
  - $f(N) = O(N^2)$ (best answer, asymptotically tight)

- $O(N^2)$: reads "order N-squared" or "Big-Oh N-squared"

Big Oh: more examples

- $N^2 / 2 - 3N = O(N^2)$
- $1 + 4N = O(N)$
- $7N^2 + 10N + 3 = O(N^2) = O(N^3)$
- $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
- sin N = O(1); 10 = O(1), $10^{10} = O(1)$
- $$\sum_{i=1}^{N} i \leq N \cdot N = O(N^2)$$

  $$\sum_{i=1}^{N} i^2 \leq N \cdot N^2 = O(N^3)$$
- log N + N = O(N)
- $\log^k N = O(N)$ for any constant k
- $N = O(2^N)$, but 2^N is not O(N)
- 2^{10N} is not $O(2^N)$

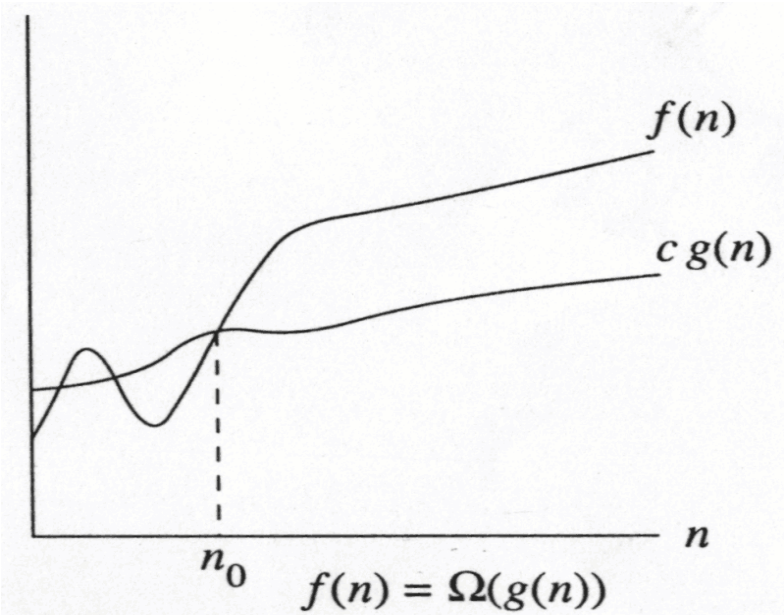
Some rules

When considering the growth rate of a function using Big-Oh

- Ignore the lower order terms and the coefficients of the highest-order term
- No need to specify the base of logarithm
  - Changing the base from one constant to another changes the value of the logarithm by only a constant factor

- If $T_1(N) = O(f(N)$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$

Big-Omega



- $\exists$ c , $n_0 > 0$ such that $f(N) \geq c\ g(N)$ when $N \geq n_0$
- f(N) grows no slower than g(N) for “large” N

Big-Omega

- $f(N) = \Omega(g(N))$
- There are positive constants c and n_0 such that

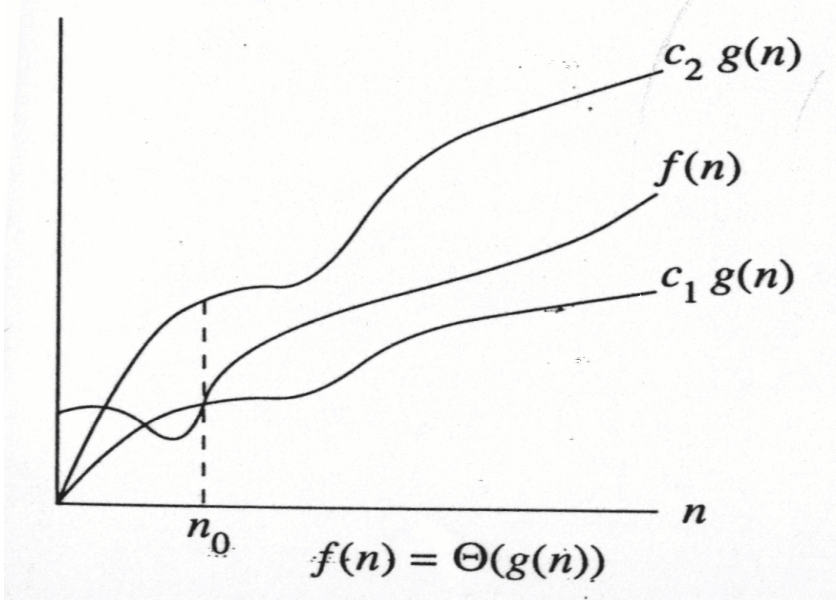
  $f(N) \geq c\ g(N)$ when $N \geq n_0$

- The growth rate of f(N) is *greater than or equal to* the growth rate of g(N).

Big-Omega: examples

- Let $f(N) = 2N^2$. Then
  - $f(N) = \Omega(N)$
  - $f(N) = \Omega(N^2)$ (best answer)

f(N) = Θ(g(N))



- the growth rate of f(N) *is the same as* the growth rate of g(N)

Big-Theta

- $f(N) = \Theta(g(N))$ iff
  $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$
- The growth rate of f(N) *equals* the growth rate of g(N)
- Example: Let $f(N)=N^2$, $g(N)=2N^2$
  - Since $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$,
    thus $f(N) = \Theta(g(N))$.
- Big-Theta means the bound is the tightest possible.

Some rules

- If T(N) is a polynomial of degree k, then $T(N) = \Theta(N^k)$.
- For logarithmic functions,

  $T(\log_m N) = \Theta(\log N)$.

General Rules

- For loops
  - at most the running time of the statements inside the for-loop (including tests) times the number of iterations.
- Nested for loops

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        k++;
```

  - the running time of the statement multiplied by the product of the sizes of all the for-loops.
  - $O(N^2)$

General rules (cont'd)

- Consecutive statements

```
for (i=0;i<n;i++)
      a[i]=0;
for (i=0;i<n;i++)
      for (j=0;j<n;j++)
            a[i] += a[j]+i+j;
```

  - These just add
  - $O(N) + O(N^2) = O(N^2)$

- `If S1`

  `Else S2`

  - never more than the running time of the test plus the larger of the running times of S1 and S2.

Example

- Calculate

$$\sum_{i=1}^{N} i^3$$

```
int sum(int n)
{
    int partialSum;

1   partialSum=0;                  1
2   for (int i=1;i<=n;i++)         2N+2
3      partialSum += i*i*i;        4N
4   return partialSum;             1
}
```

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time four units
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 6N + 4 $\Rightarrow$ O(N)

Example

1. How many times count++ will be executed?

```
int count = 0;
 for (int i = 0; i < N; i++)
     for (int j = 0; j < i; j++)
         count++;
```

- When i=0, it will run 0 times.
  When i=1, it will run 1 times.
  When i=2, it will run 2 times and so on
- Total number of times **count++** will run is $0+1+2+...+(N-1)=N*(N-1)/2$.
- So the time complexity will be $O(N^2)$.

Example

2. How many times count++ will be executed?

```
int count = 0;
    for (int i = N; i > 0; i /= 2)
        for (int j = 0; j < i; j++)
            count++;
```

- When i=N, it will run N times.
  When i=N/2, it will run N/2 times.
  When i=N/4, it will run N/4 times and so on
- Total number of times **count++** will run is

  N + N/2 +N/4 +…..1 = 2* N
- So the time complexity will be O(N).

Another Example

- Maximum Subsequence Sum Problem
- Given (possibly negative) integers A_1, A_2, ..., A_n, find the maximum value of

  – For convenience, the maximum subsequence sum is 0 if all the integers are negative

$$\sum_{k=i}^{j} A_k$$

- E.g. for input –2, 11, -4, 13, -5, -2
  – Answer: 20 (A_2 through A_4)

Algorithm 1: Simple

- Exhaustively tries all possibilities (brute force)

```
int maxSubSum1 (const vector<int> &a)
{
        int maxSum=0;

        for (int i=0;i<a.size();i++)
                for (int j=i;j<a.size();j++)
                {
                        int thisSum=0;

                        for (int k=i;k<=j;k++)
                                thisSum += a[k];

                        if (thisSum > maxSum)
                                maxSum = thisSum;
                }
        return maxSum;
}
```

- $O(N^3)$

Binary search

```
int
binary_search( input_type a[ ], input_type x, unsigned int n )
{
int low, mid, high;       /* Can't be unsigned; why? */
/*1*/       low = 0; high = n - 1;
/*2*/       while( low <= high )
{
/*3*/            mid = (low + high)/2;
/*4*/            if( a[mid] < x )
/*5*/                 low = mid + 1;
else
/*6*/           if ( a[mid] < x )
/*7*/                 high = mid - 1;
else
/*8*/               return( mid );  /* found */
}
/*9*/       return( NOT_FOUND );
}
```

n	$\log_2 n$
1	0
2	1
4	2
8	4
16	8

O(log n)

Summary

- Algorithm
- Order of growth
- Need for algorithm analysis
- Asymptotic notation
- Running time calculation
- Examples