Algorithm Lab [ICT-2263]
Fourth Semester B. Tech. (IT/CCE)
Dept. of Information and Communication Technology
MIT, Manipal

Introduction to Algorithms and Asymptotic analysis

1 Algorithm: Design

An algorithm is a finite sequence of logically related instructions to solve a computational problem. The following problems are well defined and are considered to be computational problems.

- Given an integer x, test whether x is prime or not.
- Given a program P, check whether P runs into an infinite loop.

Any algorithm must have an input provided by the user and must produce an output. Computational instructions in the algorithm must involve only basic algebraic (arithmetic) operations and it should terminate after a finite number of steps. Finally, any algorithm must involve unambiguous instructions and produce the desired output. Mathematically, an algorithm can be represented as a function, $F : I \rightarrow O$, where I is the set of inputs and O is the set of outputs generated by the algorithm. The word algorithm comes from the name of Persian author "*Abu Jafar Mohammad ibn Musa al Khawarizmi* ".

Note: Although, the phrase *algorithm* is associated with computer science, the notion *computation* (algorithm) did exist for many centuries.

The definition of algorithm sparks natural fundamental questions;

- How to design an algorithm for a given problem?
- Is every problem algorithmically solvable? If so, how many algorithms can a problem have and how to find the efficient one.

We shall address these questions in this lecture. Let us consider an example of finding a `maximum element` in an array of size n. An algorithm is typically described using pseudo code as follows:

```
Example : Finding a maximum element in an array

Algo Max-array(A,n)
{
       Max = A[1];
          for i = 2 to n do
           if ( A[i] > Max ) then Max = A[i];

       return Max;
}
```

Note: The maximum can also be computed by sorting the array in an increasing order (decreasing order) and picking the last element (first element). There are at least five different algorithms to find a maximum element and therefore, it is natural ask for an efficient algorithm. This calls for the study of analysis of algorithms.

1.1 Types of Algorithm

There are two ways to write an algorithm, namely, top-down approach (Iterative algorithm) and bottom-up approach (Recursive algorithm). For example, the iterative and recursive algorithm for finding a factorial of a given number n is given below:

1. **Iterative :**

```
Fact(n)
{
      for i = 1 to n
          fact = fact * i;
      return fact;
}
```

   Here the factorial is calculated as $1 \times 2 \times 3 \times \ldots \times n$.

2. **Recursive :**

```
 Fact(n)
{
      if n = 1
          return 1;
      else
          return n * fact(n-1);
}
```

   Here the factorial is calculated as $n \times (n-1) \times \ldots \times 1$.

2 Algorithm: Analysis

The analysis of algorithms involves the following measures and are listed in the order of priority.

1. **Correctness:** For any algorithm, a proof of correctness is important which will exhibit the fact that the algorithm indeed output the desired answer. Often, discovering the underlying combinatorics is quite challenging.
2. **Amount of work done (time complexity) :** For a problem, there may exist many algorithms. Comparison of two algorithms is inevitable to pick the best of two. By analysis we mean, the amount of time and space required to execute the algorithm. A computational problem can have many algorithms but the estimation of time and space complexity provide an insight into reasonable directions of search for finding the efficient algorithm. The time Complexity does not refer to the actual running time of an algorithm in terms of *millisec* (system time). The

actual running time depends on the system configuration. An algorithm taking 100μs on Intel machine may take 10μs on an AMD machine. So, the time complexity must be defined independent of the system configuration. For each algorithm, we focus on `step count: the number of times each statement in the algorithm is executed`, which in some sense reflects the time complexity. The step count focuses on primitive operations along with basic operations. Moreover, this number increases with the problem size. Therefore, we express the step count (time complexity) as a function of the input size. The notion *input size* and *primitive operations* vary from one problem to another. The popular ones are;

`Common Problems`	`Associated Primitive Operations`	`Input Size`
(Search Problem) Find x in an array A	Comparison of x with other elements of A	Array size.
Multiply 2 matrices A and B (Arithmetic on Matrices)	Multiplication and Addition	Dimension of the matrix.
Sorting (Arrangement of elements in some order)	Comparison	Array size.
Graph Traversal	Number of times an edge is traversed	the number of vertices and edges.
Any Recursive Procedure	Number of recursive calls + time spent on each recursive call	Array size (for array related problems).
Finding Factorial	Multiplication	the input number.
Finding LCM(a,b)	Basic arithmetic (division, subtraction)	the number of bits needed to represent a and b.

3. **Note:** The number of primitive operations increases with the problem size. Therefore, we express the step count (time complexity) as a function of the input size. *Space complexity* is a related complexity measure that refers to the amount of space used by an algorithm. Here again, the analysis focuses on the number of words required leaving the system details. It is good to highlight that this measure is considered less significant compared to the time complexity analysis.
4. **Optimality:** For a given combinatorial problem, there may exist many algorithms. A natural question is to find the best algorithm (efficient), i.e., one might be interested in discovering best ever possible. This study also helps us to understand the inherent complexity of the problem.

2.1 Step-count Method and Asymptotic Notation

In this section, we shall look at analysis of algorithms using step count method. Time and space complexity are calculated using step count method. Some basic assumptions are;

- There is no count for { and } .
- Each basic statement like 'assignment' and 'return' have a count of 1.
- If a basic statement is iterated, then multiply by the number of times the loop is run.
- The loop statement is iterated n times, it has a count of $(n + 1)$. Here the loop runs n times for the true case and a check is performed for the loop exit (the false condition), hence the additional 1 in the count.

Examples for Step-Count Calculation:

1. Sum of elements in an array

```
                                        Step-count (T.C)      Step-count (Space)
     Algorithm Sum(a,n)
     {                                        0
          sum = 0;                            1                 1 word for sum
          for i = 1 to n do                  n+1                1 word each for i and n
              sum = sum + a[i];               n                 n words for the array a[]
          return sum;                         1
     }                                        0

                                  Total:    2n+3                (n+3) words
```

2. Adding two matrices of order m and n

```
Algorithm  Add(a, b, c, m, n)            Step Count
{
  for i = 1 to m do                       ----  m + 1
    for j = 1 to n do                     ----  m(n + 1)
      c[i,j] = a[i,j] + b[i,j]            ----  m.n
}                                              -------------
                              Total no of steps=  2mn + 2m + 2
```

Note that the first 'for loop' is executed $m + 1$ times, i.e., the first m calls are true calls during which the inner loop is executed and the last call $(m + 1)^{th}$ call is a false call.

3. Fibonacci series

```
algorithm Fibonacci(n)                   Step Count
{
  if n <= 1 then                           ---- 1
    output 'n'
  else
    f2 = 0;                                ---- 1
    f1 = 1;                                ---- 1

    for i = 2 to n do                      ---- n
    {
      f = f1 + f2;                         ---- n - 1
      f2 = f1;                             ---- n - 1
      f1 = f;                              ---- n - 1
    }
    output 'f'                             ---- 1
}                                              --------
                              Total no of steps=  4n + 1
```

Note that if $n \leq 1$ then the step count is just two and $4n + 1$ otherwise.

4. Recursive sum of elements in an array

```
algorithm RecursiveSum(a, n)                Step Count
{
  if n <= 0 then                            ---- 1
    return 0;                               ---- 1
  else
    return RecursiveSum(a, n-1) + a[n];     ---- 2 + Step Count of recursive call
}
```

Note that step count of two is added at each recursive call. One count for making a 'recursive call' with $(n-1)$ size input and the other count is for addition when the recursion bottoms out. Let the Step count of array of size n be denoted as *T(n)*, then the step-count for the above algorithm is

- *T(n)* = 3 + *T(n-1)*, $n > 0$ `/* one count for the 'if' statement and two counts at 'return statement' + count on recursive call */`
- *T(n)* = 2, $n \leq 0$

Solving, the above equation yields $T(n) = 3n + 2$.

2.2 Order of Growth

Performing step count calculation for large algorithms is a time consuming task. A natural way is to upper bound the time complexity instead of finding the exact step count. **Order of Growth** or **Rate of Growth** of an algorithm gives a simple characterization of the algorithm's efficiency by identifying relatively significant term in the step count. (e.g.) For an algorithm with a step count $2n^2+3n+1$, the order of growth depends on $2n^2$ for large n. Other terms are relatively insignificant as n increases. *Asymptotic analysis* is a technique that focuses analysis on the 'significant term'. In the next section, we shall look at some of the commonly used asymptotic notations in the literature. The popular ones are;

1. Big-oh Notation (O) - To express an upper bound on the time complexity as a function of the input size.
2. Omega (Ω) - To express a lower bound on the time complexity as a function of the input size.
3. Theta (Θ) - To express the tight bound on the time complexity as a function of the input size.

2.3 Asymptotic upper bounds

Big-oh notation
The function $f(n) = O(g(n))$ if and only if there exist positive constants c, n_0 such that $f(n) \leq cg(n), \forall n \geq n_0$. Big-oh can be used to denote all upper bounds on the time complexity of an algorithm. Big-oh also captures the worst case analysis of an algorithm.
Examples:

1. $3n + 2$
   $3n + 2 \leq 4n$, $c = 4$ $\forall n \geq 2$
   $\Rightarrow 3n + 2 = O(n)$
   Note that $3n + 2$ is $O(n^2), O(n^3), O(2^n), O(10^n)$ as per the definition. i.e., it captures all upper bounds. For the above example, $O(n)$ is a tight upper bound whereas the rest are loose upper bounds. Is there any notation which captures all the loose upper bounds?

2. $100n + 6 = O(n)$
   $100n + 6 \leq 101.n$, $c = 101$ $\forall n \geq 6$. One can also write $100n + 6 = O(n^3)$.
3. $10n^2 + 4n + 2 = O(n^2)$
   $10n^2 + 4n + 2 \leq 11.n^2$, $c = 11$ $\forall n \geq 5$
4. $6.2^n + n^2 = O(2^n)$
   $6.2^n + n^2 \leq 7.2^n$, $c = 7$ $\forall n \geq 7$
5. $3n + 3 = O(2^n)$
   $3n + 3 \leq 10.2^n$, $c = 10$ $\forall n \geq 5$
6. $n^3 + n + 5 = O(n^3)$
   $n^3 + n + 5 \leq 7.n^3, c = 7, \forall n \geq 0$

Remark:

- Note that $3n + 2 \neq O(1)$. i.e., there does not exist c and n_0 such that, $3n + 2 \leq c.1$ for all n beyond n_0. However large the c is, there exist n beyond which $3n + 2 > c.1$.
- $10n^2 + 4n + 2 \neq O(n)$.
- $3^n \neq O(2^n)$. We shall prove this by contradiction. Suppose $3^n = O(2^n)$, then by definition, $3^n \leq c2^n$. This implies that $1.5^n \leq c$ where c is a positive constant, which is a contradiction.
- $n! \leq c.n^n$ for $c = 1$ and $\forall n \geq 2$ i.e., $n! = O(n^n)$. Moreover, from Stirling's approximation it follows that $n! \approx \sqrt{2n\pi}.n^n e^{-n}$

Next we present the notation which precisely captures the loose upper bounds.

o-notation
The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2$=$O(n^2)$ is asymptotically tight, but the bound $2n$=$O(n^2)$ is not. We use o-notation (Little oh) to denote an upper bound that is not asymptotically tight. We formally define as $f(n) = o(g(n))$ if for any positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that $0 \leq f(n) < c.g(n)$ for all $n \geq n_0$.
Note that in the definition the inequality works for any positive constant $c > 0$. This is true because $g(n)$ is a loose upper bound, and hence $g(n)$ is polynomially larger than $f(n)$ by $n^\epsilon, \epsilon > 0$. Due to this n^ϵ, the contribution of c to the inequality is minimal which is why the quantifier in o notation is universal whereas in O is existential.
For example,

1. $2n = o(n^2)$, but $2n^2 \neq o(n^2)$. Note that here n^2 is polynomially larger than $2n$ by $n^\epsilon, \epsilon = 1$.
2. $100n+6 = o(n^{1.2})$ Here $n^{1.2}$ is polynomially larger than $100n+6$ by $n^\epsilon, \epsilon = 0.2$. For any positive constant c, there exist n_0 such that $\forall n \geq n_0, 100n + 6 \leq c.n^{1.2}$
3. $10n^2 + 4n + 2 = o(n^3)$ Here n^3 is polynomially larger than $10n^2 + 4n + 2$ by $n^\epsilon, \epsilon = 1$
4. $6.2^n + n^2 = o(3^n)$ Note that 3^n is $1.5^n \times 2^n$. So for any $c > 0$, $2^n \leq c.3^n$. The value of c is insignificant as 1.5^n dominates any $c > 0$.
5. $3n + 3 = o(n^{1.00001})$ Here $n^{1.00001}$ is polynomially larger than $3n + 3$ by $n^\epsilon, \epsilon = 0.00001$
6. $n^3 + n + 5 = o(n^{3.1})$ Here $n^{3.1}$ is polynomially larger than $n^3 + n + 5$ by $n^\epsilon, \epsilon = 0.1$

Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
Having looked at the upper bounds, in the similar way we shall now see asymptotic lower bounds.

2.4 Asymptotic lower bounds

Omega notation
The function $f(n) = \Omega(g(n))$ if and only if there exist positive constants c, n_0 such that $f(n) \geq c.g(n), \forall n \geq n_0$. Omega can be used to denote all lower bounds of an algorithm. Omega notation also denotes the best case analysis of an algorithm.
Examples:

1. $3n + 2$
   $3n + 2 \geq n, \forall n \geq 1$
   $\Rightarrow 3n + 2 = \Omega(n)$
2. $10n^2 + 4n + 2 = \Omega(n^2)$
   $10n^2 + 4n + 2 \geq n^2$, $c = 1$ $\forall n \geq 1$
3. $n^3 + n + 5 = \Omega(n^3)$
   $n^3 + n + 5 \geq n^3, c = 1, \forall n \geq 0$
4. $2n^2 + nlogn + 1 = \Omega(n^2)$
   $2n^2 + nlogn + 1 \geq 2.n^2, c = 2, \forall n \geq 1$
5. $6.2^n + n^2 = \Omega(2^n) = \Omega(n^2) = \Omega(n) = \Omega(1)$
   $6.2^n + n^2 \geq 2^n$, $c = 1$ $\forall n \geq 1$
   Of the above, $\Omega(2^n)$ is a tight lower bound, while all others are loose lower bounds.

Remark:

- $3n^2 + 2 \neq \Omega(n^3)$. Reason: There does not exist a positive constant c such that $3n^2 + 2 \geq c.n^3$ for every $n \geq n_0$ as we cannot bound n by a constant. That is, on the contrary if $3n^2 + 2 \geq c.n^3$, then $n \leq \frac{1}{c}$ is a contradiction.
- $3.2^n \neq \Omega(n!)$.
- $5 \neq \Omega(n)$.

ω-notation
We use ω-notation to denote a lower bound that is not asymptotically tight.
We define $\omega(g(n))$ (little-omega) as
$f(n) = \omega(g(n))$ if for any positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that $0 \leq c.g(n) < f(n)$ for all $n \geq n_0$.
For example,

1. $n^2 = \omega(n)$ but $n^2 \neq \omega(n^2)$.
2. $3n + 2 = \omega(log(n))$
3. $10n^3 + 4n + 2 = \omega(n^2)$
4. $5n^6 + 7n + 9 = \omega(n^3)$
5. $2n^2 + nlogn + 1 = \omega(n^{1.9999999})$
6. $15 \times 3^n + n^2 = \omega(2^n) = \omega(n^2) = \omega(n) = \omega(1)$

The relation $f(n) = \omega(g(n))$ implies that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

2.5 Asymptotic tight bound

Theta notation
The function $f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$. Theta can be used to denote tight bounds of an algorithm. i.e., $g(n)$ is a lower bound as well as an upper bound for $f(n)$. Note that $f(n) = \Theta(g(n))$ if and only if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.
Examples

1. $3n + 10^{10}$
   $3n \leq 3n + 10^{10} \leq 4n, \forall n \geq 10^{10}$
   $\Rightarrow 3n + 10^{10} = \Theta(n)$
   Note that the first inequality captures $3n + 10^{10} = \Omega(n)$ and the later one captures $3n + 10^{10} = O(n)$.
2. $10n^2 + 4n + 2 = \Theta(n^2)$
   $10n^2 \leq 10n^2 + 4n + 2 \leq 20n^2, \forall n \geq 1, c_1 = 10, c_2 = 20$
3. $6(2^n) + n^2 = \Theta(2^n)$
   $6(2^n) \leq 6(2^n) + n^2 \leq 12(2^n), \forall n \geq 1, c_1 = 6, c_2 = 12$
4. $2n^2 + nlogn + 1 = \Theta(n^2)$
   $2n^2 \leq 2n^2 + nlog_2 n + 1 \leq 5.n^2, \forall n \geq 2, c_1 = 2, c_2 = 5$
5. $n\sqrt{n} + nlog_2(n) + 2 = \Theta(n\sqrt{n})$
   $n\sqrt{n} \leq n\sqrt{n} + nlog_2(n) + 2 \leq 5.n\sqrt{n},\ \forall n \geq 2, c_1 = 1, c_2 = 5$

Remark:

- $3n + 2 \neq \Theta(1)$. Reason: $3n + 2 \neq O(1)$
- $3n + 3 \neq \Theta(n^2)$. Reason: $3n + 3 \neq \Omega(n^2)$
- $n^2 \neq \Theta(2^n)$. Reason: $n^2 \neq \Omega(2^n)$ *Proof:* Note that $f(n) \leq g(n)$ if and only if $log(f(n)) \leq log(g(n))$. Suppose $n^2 = \Omega(2^n)$, then by definition $n^2 \geq c.2^n$ where c is a positive constant. Then, $log(n^2) \geq log(c.2^n)$, and $2log(n) \geq nlog(2)$, which is a contradiction.

3 Properties of Asymptotic notation

1. Reflexivity :

$f(n) = O(f(n))$ $\qquad$ $f(n) = \Omega(f(n))$ $\qquad$ $f(n) = \theta(f(n))$

2. Symmetry :

$f(n) = \theta(g(n))$ if and only if $g(n) = \theta(f(n))$

Proof:

Necessary part: $f(n) = \theta(g(n)) \Rightarrow g(n) = \theta(f(n))$
By the definition of θ , there exists positive constants c_1, c_2, n_o such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_o$

$$\Rightarrow g(n) \leq \frac{1}{c_1}.f(n) \text{ and } g(n) \geq \frac{1}{c_2}.f(n)$$

$\Rightarrow \frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)$

Since c_1 and c_2 are positive constants, $\frac{1}{c_1}$ and $\frac{1}{c_2}$ are well defined. Therefore, by the definition of θ , $g(n) = \theta(f(n))$

Sufficiency part: $g(n) = \theta(f(n)) \Rightarrow f(n) = \theta(g(n))$

By the definition of θ , there exists positive constants c_1, c_2, n_o such that $c_1.f(n) \leq g(n) \leq c_2.f(n)$ for all $n \geq n_o$

$\Rightarrow f(n) \leq \frac{1}{c_1}.g(n)$ and $f(n) \geq \frac{1}{c_2}.g(n)$

$\Rightarrow \frac{1}{c_2}.g(n) \leq f(n) \leq \frac{1}{c_1}.g(n)$

By the definition of θ , $f(n) = \theta(g(n))$
This completes the proof of Symmetry property.

3. Transitivity :

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

Proof:

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

By the definition of Big-Oh(O) , there exists positive constants c, n_o such that $f(n) \leq c.g(n)$ for all $n \geq n_o$

$\Rightarrow f(n) \leq c_1.g(n)$

$\Rightarrow g(n) \leq c_2.h(n)$

$\Rightarrow f(n) \leq c_1.c_2 h(n)$

$\Rightarrow f(n) \leq c.h(n)$, where, $c = c_1.c_2$

By the definition, $f(n) = O(h(n))$ **Note :** Theta(Θ) and Omega(Ω) also satisfies Transitivity Property.

4. Transpose Symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
Proof:

Necessity: $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$

By the definition of Big-Oh (O)

$\Rightarrow f(n) \leq c.g(n)$ for some positive constant c

$\Rightarrow g(n) \geq \frac{1}{c} f(n)$

By the definition of Omega (Ω) , $g(n) = \Omega(f(n))$

Sufficiency: $g(n) = \Omega(f(n)) \Rightarrow f(n) = O(g(n))$

By the definition of Omega (Ω), for some positive constant c

$\Rightarrow g(n) \geq c.f(n)$

$\Rightarrow f(n) \leq \frac{1}{c} g(n)$

By the definition of Big-Oh(O) , $f(n) = O(g(n))$
Therefore, Transpose Symmetry is proved.

4 Some more Observations on Asymptotic Notation

Lemma 1. *Let $f(n)$ and $g(n)$ be two asymptotic non-negative functions.*
Then, $max(f(n), g(n)) = \theta(f(n) + g(n))$

Proof. Without loss of generality, assume $f(n) \leq g(n)$, $\Rightarrow max(f(n) + g(n)) = g(n)$

Consider, $g(n) \leq max(f(n) + g(n)) \leq g(n)$

$\Rightarrow g(n) \leq max(f(n) + g(n)) \leq f(n) + g(n)$

$\Rightarrow \frac{1}{2} g(n) + \frac{1}{2} g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

From what we assumed,we can write

$\Rightarrow \frac{1}{2} f(n) + \frac{1}{2} g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

$\Rightarrow \frac{1}{2}(f(n) + g(n)) \leq max(f(n), g(n)) \leq f(n) + g(n)$

By the definition of θ ,

$max(f(n), g(n)) = \theta(f(n) + g(n))$

Lemma 2. *For two asymptotic functions $f(n)$ and $g(n)$, $O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$*

Proof. Without loss of generality, assume $f(n) \leq g(n)$

$\Rightarrow O(f(n)) + O(g(n)) = c_1 f(n) + c_2 g(n)$

From what we assumed,we can write

$O(f(n)) + O(g(n)) \leq c_1 g(n) + c_2 g(n)$

$\leq (c_1 + c_2) g(n)$

$\leq$ c $g(n)$

$\leq$ c $max(f(n), g(n))$

By the definition of Big-Oh(O),

$O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$