Data Structure & Algorithms

Lec 07: Algorithms Analysis

Part-01

Fall 2017 - University of Windsor

Dr. Sherif Saad

Agenda

1. Algorithms
2. Asymptotic Analysis
3. Complexity of Algorithms
4. Guidelines Pseudocode

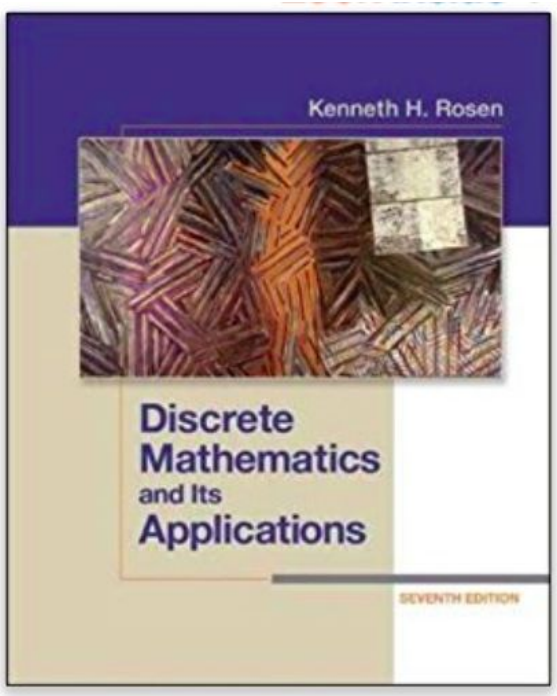
Learning Outcome

By the end of this class, you should be able to

- Decide if a given algorithm is a good or bad algorithm.
- Explain the meaning of Big-O notation and other asymptotic analysis notation.
- Estimate the runtime complexity of a given algorithm.
- Express algorithm in a pseudocode format.

References

The slides follow Rosen "Discrete mathematics and Its Applications ", 7th edition



Algorithm

An algorithm is a finite sequence of precise instructions for performing a computation or for solving a problem.

Algorithm Analysis or computational complexity analysis focuses on measuring the processing time and computer memory required by the algorithm to solve problems of a particular size.

How do we compare algorithms?

Properties of a Good Algorithm

Precision: an algorithm should have clear and well-defined instructions

Uniqueness: each step taken in the algorithm should give a definite result

Feasibility: the algorithm should be practicable in real life.

Finiteness: the algorithm stops after a finite number of instructions are executed

Generality: the algorithm applies to a set of inputs

Correctness: the algorithm should produce the correct output.

Effectiveness: the algorithm should execute each step in finite amount of time and memory

Properties of a Good Algorithm (cont...)

Show that the algorithm for finding the maximum element in a finite sequence of integers has all the properties of a good algorithm.

procedure $max(a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ **to** n
    if $max < a_i$ **then** $max := a_i$
return max {max is the largest element}

[Precision - Uniqueness - Feasibility - Finiteness - Generality - Correctness - Effectiveness]

Properties of a Good Algorithm (cont...)

procedure *double*(n: positive integer)
while $n > 0$
$n := 2n$

Which properties of a good algorithm properties the above procedure has and which it lack.

[Precision - Uniqueness - Feasibility - Finiteness - Generality - Correctness - Effectiveness]

Properties of a Good Algorithm (cont...)

procedure *divide*(n: positive integer)
while $n \geq 0$
  $m := 1/n$
  $n := n - 1$

Which properties of a good algorithm properties the above procedure has and which it lack.

[Precision - Uniqueness - Feasibility - Finiteness - Generality - Correctness - Effectiveness]

Properties of a Good Algorithm (cont...)

procedure *sum*(*n*: positive integer)
sum := 0
while *i* < 10
    sum := *sum* + *i*

Which properties of a good algorithm properties the above procedure has and which it lack.

[Precision - Uniqueness - Feasibility - Finiteness - Generality - Correctness - Effectiveness]

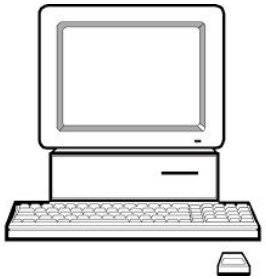
Properties of a Good Algorithm (cont...)

procedure *choose*(a, b: integers)
$x :=$ either a or b

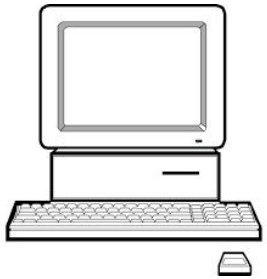
Which properties of a good algorithm properties the above procedure has and which it lack.

[Precision - Uniqueness - Feasibility - Finiteness - Generality - Correctness - Effectiveness]

Asymptotic Analysis



Machine 1 executes algorithm A



Machine 2 executes algorithm B

How do we compare algorithms?

- We could use execution time (how many seconds the algorithm takes to terminate)?
- We could use the size of the algorithm (e.g how many statements in the algorithms?)

Asymptotic Analysis (cont...)

How do we compare algorithms?

We need a method to compare algorithms that is independent of the machine hardware and software.

We could express the running time of a given algorithm as a function of input size n. Then, for any number of algorithms for a given problem we can compare these different functions.

Expressing the algorithm run-time as a function of its input allow use to estimate the rate of growth without worry about the hardware or the software.

What is Rate of Growth?

The rate at which the running time increases as a function of the input is called rate of growth.

The growth of functions is often described using a special notation **Big-O.**

With Big-O notation we can determine whether it is practical to use a particular algorithm to solve a problem as the size of the input increases.

We can compare two algorithms to determine which is more efficient as the size of the input grows.

Growth of Functions

Let f(n) be a function of a positive integer n, the dominant term of f(n) determines the behavior of *f(n)* as $\boldsymbol{n \to \infty}$

Example $f(n) = 100n + 3n^3 + 2n^2 + 7$

The dominant term of f(n) is $3n^3$, this means as n becomes large $\mathbf{n \to \infty}$ $3n^3$ dominates the behavior of f(n). The effect other terms $100n$, $2n^2$ etc become less significant.

Constants and lower order terms are ignored to simplify the algorithm runtime estimation.

Big-O Notation

This notation gives the tight upper bound of a given function. We represent as $f(n) = O(g(n))$.

This means as n becomes large the upper bound of f(n) is g(n)

For example: assume the runtime of an algorithm is

$$f(n) = n^3 + 2\log(n) + 100n + 10$$

Then n^3 is g(n) (dominant term of f(n)) which mean g(n) gives the maximum rate of growth for f(n) $n \to \infty$

Big-O Notation (cont…)

What is Big-O represent? Is it worst case, best case or average case?

Big-O Formal Definition

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

Big-O Notation (cont...)

The functions need to satisfy the following condition in order to be in Big-O class. f(n) is O(g(n)) if and if

0 <= f(n) <= cg(n) for all **n> n_0** ,where ***c and n_0*** are positive constants

Example

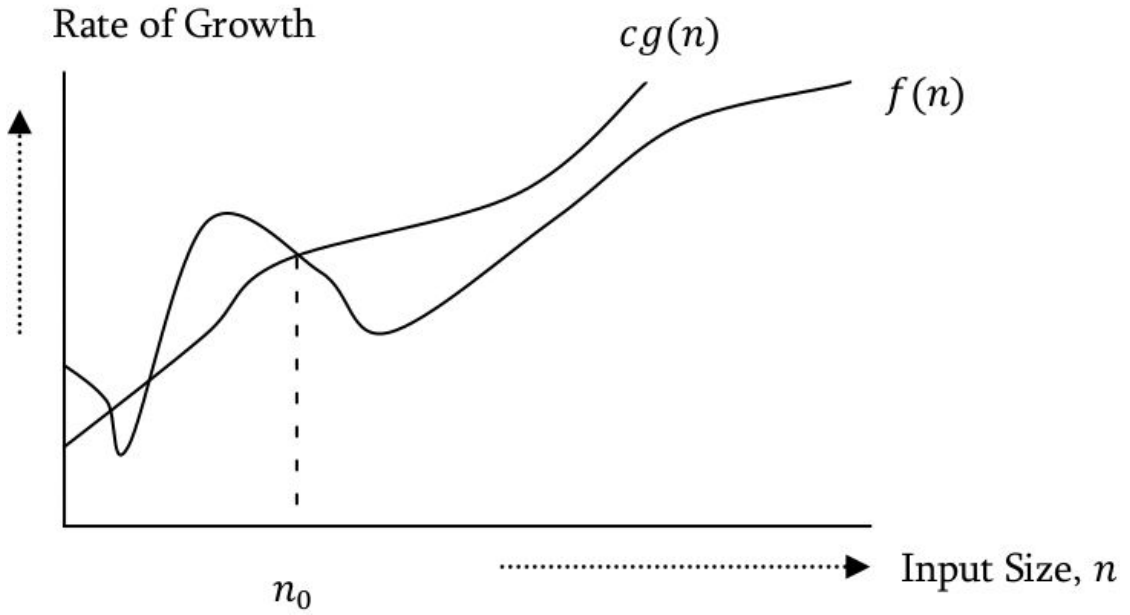
n^3 does not belong to O(n) as it fails satisfy the inequality f(n) <= cg(n)

Big-O Notation (cont...)

f(n) is O(g(n)): f(n) grows slower than some fixed multiple of g(n) as n grows without bound.

The constants C and n_0 are called witnesses to the relationship f(n) is O(g(n)).

To establish that f(n) is O(g(n)) we only need one pair of witnesses



Big-O Notation (cont...)

The runtime complexity of an algorithm A is given by T(n) = 3n +2, show that algorithm A has $O(n^2)$ runtime

$f(n) = 3n + 2$

$g(n) = n$

$f(n) <= cg(n)$ for all $n > n_0$

$3n + 2 <= cn$

$c = 4$ and $n_0 = 2$

Big-O Notation (cont...)

The runtime complexity of an algorithm A is given by $T(n) = 3n^2+5$, show that algorithm A has $O(n^2)$ runtime

$f(n) = 3n^2 + 5$

$g(n) = n^2$

$f(n) <= cg(n)$ for all $n > n_0$

$3n^2 + 5 <= cn^2$

$c = 4$ and $n_0 = 3$

Big-Omega and Big-Theta Notation

Big-Omega : gives the tighter lower bound of a given algorithm.

The function need to satisfy the following condition in order to be in Bio-O class. f(n) is Ω(g(n)) if and if

$$0 < cg(n) < f(n) \text{ for all } n > n_0$$ where c and n_0 are positive constant

Big-Theta: this notation decide whether the upper and lower bound of a given algorithm are the same or not. If the rate of growth in the worst case and best case is the same then the average case will be also the same

$$0 < c_1 g(n) < f(n) < c_2 g(n) \text{ for all } n > n_0$$ where c_1, c_2 and n_0 are positive constant

Why it is called Asymptotic Analysis

f(n) is a function and we are trying to find another function g(n) that approximates f(n) at higher values of n.

This means g(n) is a curve for f(n). In mathematics we call this curve an asymptotic curve.

This is why we call algorithm analysis asymptotic analysis.

Asymptotic Analysis General Rules

O(1) : Describes an algorithm that will always execute in a fixed/constant time regardless of the input size.

Asymptotic Analysis General Rules (cont...)

O(n) : Describes an algorithm with a runtime that will grow linearly and in direct proportion to the input size, for example linear search

```
14 int LinearSearch(int key, int data[], int size){
15
16     int index =-1;
17
18     for(int i=0; i<size; i++){
19
20         if(data[i]==key){
21             index = i;
22             return index;
23         }
24     }
```

Asymptotic Analysis General Rules (cont...)

$O(n^2)$: Describes an algorithm with a runtime that will grow quadratically and directly proportional to the square of the input size. Example bubble sort

```
 7 void bubbleSort(int arr[], int n)
 8 {
 9     int i, j;
10     for (i = 0; i < n-1; i++)
11
12         // Last i elements are already in place
13         for (j = 0; j < n-i-1; j++)
14             if (arr[j] > arr[j+1])
15                 swap(&arr[j], &arr[j+1]);
16 }
```

Asymptotic Analysis General Rules (cont…)

$O(2^n)$: Describes an algorithm with with a runtime that doubles with each addition to the input size

```
 7 int fibonacci(int num){
 8     if (num == 1) {
 9         return 1;
10     }
11     else if (num == 0){
12         return 0;
13     }
14     else {
15         return fibonacci(num - 1) + fibonacci(num - 2);
16     }
17 }
```

Asymptotic Analysis General Rules (cont…)

O(log n) : Describes an algorithm that takes a constant time to cut the problem size by a fraction (usually by ½) as in binary search. Note it is base 2 log

```
35 int BinarySearch(int key, int data[], int size){
36
37     int begin =0;
38     int end = size-1;
39     int mid;
40
41     while(begin<=end){
42         mid = (begin+end)/2;
43
44         if(data[mid]==key)
45             return mid;
46         if(data[mid] < key)
47             begin = mid + 1;
48         else end = mid -1;
49     }
50     return -1;
51 }
```

How to Calculate the runtime of an Algorithm?

Consecutive Statements:

- Add the time complexities of each statement

Loop:

- in general, the running time of a loop is at most the runtime of the statements inside the loop multiplied by the number of iterations.
- In nested loops the total running time is the product f the sizes of all the loops.

IF-Then-Else:

- The runtime of the test plus either the then part of the else part (which is the larger)

How to Calculate the runtime of an Algorithm?

What is the runtime of the function FOO?

$T(n) = c_0 + c_1 + \max(c_2, c_3) + c_4$

O(1)

```
7  int Foo(int n){
8
9      int  i= 21;
10     if ((n-i) >0){
11
12         n = 0;
13     }else{
14         n = n-i;
15     }
16
17     return n;
18 }
```

How to Calculate the runtime of an Algorithm?

What is the runtime of the function FOO?

$T(n) = c_0 + c_1 + c_2 n + c_4$

O(n)

```
7  int Foo(int n){
8
9      int  i = n;
10     int sum=0;
11
12     while (i >=1){
13         sum = sum + 1;
14     }
15
16     return sum;
17 }
```

How to Calculate the runtime of an Algorithm?

What is the runtime of the function FOO?

$T(n) = c_0 + c_1 + 2n + c_4$

$T(n) = O(n)$

something doesn't add up??

```
7  int Foo(int n){
8
9      int  i = n;
10     int count=0;
11     while (i >=1){
12         i = i/2;
13         count ++;
14     }
15
16     return count;
17 }
```

How to Calculate the runtime of an Algorithm?

What is the runtime of the function FOO?

T(n) = ??

O(n) =??

```
 7 int Foo(int n){
 8
 9     int i, count;
10     count = 0;
11     for(int i=1;i*i <= n; i++){
12         count++;
13     }
14     return count;
15 }
```

Commonly Terminology for Algorithms Complexity

Complexity	Terminology
O(1)	Constant Complexity
O(log n)	Logarithmic complexity
O(n)	Linear complexity
O(n log n)	Linearithmic complexity
$O(n^b)$	Polynomial complexity
$O(b^n)$	Exponential complexity
O(n!)	Factorial complexity

Commonly Terminology for Algorithms Complexity

Polynomial Complexity: an algorithm with runtime complexity of $O(n^b)$ where b is an integer >=1 is has a polynomial complexity for example the bubble sort has n^2 runtime complexity.

Exponential Complexity: an algorithm has an exponential complexity if it has a time complexity $O(b^n)$ where b is an integer > 1. The Tower of Hanoi problem has 2^n complexity

Factorial Complexity: an algorithm has an factorial complexity if it has a time complexity of $O(n!)$. Example find all possible paths for a travel salesman problem.

Pseudocode Guidelines

Pseudocode is an intermediate step between a natural language description of algorithm and actual implementation of the algorithm in a programming language.

The advantages of using pseudocode include the simplicity with which it can be written and understood and the ease of producing actual computer code.

There is no universal standard syntax on how we should write pseudocode. In fact, there are different styles, and specific programming languages influence some them.

Pseudocode Guidelines

A good pseudocode should:

1. Use vocabularies from the domain of the problem.
2. Allow anyone who understand the problem domain to understand the algorithm.
3. Use short but informative statements.
4. Use one word per concept.
5. Use verbs for procedure names and nouns for variable names

Pseudocode Guidelines

Procedure Statements a pseudocode for an algorithm begins with a procedure statement that gives the name of an algorithm, lists the input variables, and describes what kind of variable each input is.

```
Procedure maximum (L: list of integers)
```

Assignments Statement:

Variable := expression

max := first element in L

Pseudocode Guidelines

Assignments Statement:

1. max := first element in L
2. L := a_1, a_2, …, a_n list of integers
3. max := a_1
4. max := L[0]
5. max := largest integer in the list L
6. swap a and b
7. Interchange a and b

Pseudocode Guidelines

Conditional Statement:

```
IF condition Then
    statement or block of statements
Else
    statement or block of statements
```

You may use indentation or { } or BEGIN and END

Pseudocode Guidelines

Loop Constructions: In general, you should limits the keywords for loop to for and while

```
For variable := initial value to final value
        statement or block of statements
For all elements with a certain property in a collection
        statement or block of statements
For each element in a collection
        statement or block of statements
```

Pseudocode Guidelines

We can use a procedure from within another procedure (or within itself in a recursive program), for example

```
L := a1, a2, …, an list of integers
max = FindMax(L)
L := CALL Sort(L)
```

We use a return statement to show where a procedure produces output.

```
return max
```

Self-assessment

A algorithm with O(n) runtime complexity is not always better than algorithm with $O(n^2)$ runtime complexity. Do you agree or disagree explain your answer?

Show that $f(n) = 3n^2 + 2n + 4$ is $O(n^2)$

Write an algorithm to find the minimum and maximum value in a list of integers. What is the T(n) of your algorithm?