CSCI 2110 Data Structures and Algorithms

Module 2: Introduction to Algorithm Complexity

DALHOUSIE UNIVERSITY

Learning Objectives/Topics

- *What is algorithm time complexity? Why should we care?*
- *How do you express algorithm time complexity in terms of basic operations?*
- *Define the standard measure of algorithm complexity – the order of complexity or big O.*
- *Study practical examples of typical big O's and know simple rules for deriving big O.*
- *Know other types of complexity, namely, Big-Omega, Big-Theta and Little-O and their relation to Big-O.*
- *Distinguish between average case, worst-case and best-case running time complexities.*

What is algorithm time complexity?

- *Algorithm time complexity is just a measure of how fast your algorithm/program runs.*

- *Thus it is a measure of the efficiency of the algorithm.*

- *This efficiency can be expressed by the speed or the running time of the algorithm (that is, of the program implementing the algorithm).*

Why should we care?

- *Understanding algorithm time complexity can make a world of difference in software design.*
- *We can compare algorithms and choose the right algorithm for the right task.*
- *Some simple examples:*
  - *Bubble sort algorithm on a million records → 1 billion steps.*
  - *Quick sort algorithm on a million records → 20 million steps!*
  - *Linear search algorithm on a million records → 1 million steps.*
  - *Binary search algorithm on a million records → 20 steps!*

Time Complexity vs. Space Complexity

- *In the previous examples, we are concerned about the runtime efficiency or the time complexity of the algorithm.*
- *Another factor that can also determine the efficiency of the algorithm is the space complexity.*
- *Space complexity is a measure of the memory required by the algorithm.*
- *Principles behind concepts of understanding time and space complexity are similar.*
- *We will focus on <u>time complexity</u>.*

We need to measure the run time –
why not use the "wall clock" approach?

- *Suppose that we run a competition in this class to test who has built the fastest spell checker algorithm.*
- *Manvi says:" My spell checker took 2.5 minutes to complete its task".*
- *Derek announces:" My spell checker tookonly 1.5 minutes"*
- *Mohamad shouts:" My spell checker took 50 seconds".*
- *Yiyang pipes in: "My spell checker took 40 seconds!"*
- *Megan quietly texts: "My spell checker just took 10 seconds."*
- *This is called the" wall clock" approach to measuring time complexity→ looking at the absolute time the program took to run.*

The "wall clock" approach is not reliable …

- *This is not a reliable measure because …*
- *…many factors cloud the actual efficiency of the algorithm, for example,*
  - *CPU Speed and OS*
  - *System environment (how many other processes were running simultaneously?)*
  - *Programming language and platform*
  - ***Differences in the document size (the size of the input)***

What is a better approach?

- *We need to compare algorithms independent of the peripheral issues such as CPU speed, etc.*
- *Hence a better approach is to count the number of <u>basic operations</u> in the algorithm for a specific input size.*
- *The running time will be proportional to this count.*
- *What are the basic operations of an algorithm?*
  - *Additions/subtractions*
  - *Multiplications/divisions*
  - *Comparisons*
  - *Assignment (copy) operations, etc.*

We will sneak in an approximation….

- *We will say that every basic operation such as*
  - *Addition/ Subtraction*
  - *Multiplication/Division/Modulus*
  - *Comparison*
  - *Assignment (copy)*
  - *etc.*

  takes the same amount of time.
- *We'll see later that in the long run, this approximation is valid.*
- *In some cases, we may not even count all the basic operations, but just some dominant operations.*
- *Again, in the long run this approximation will be valid.*

What about the input data size?

- *This is perhaps the most important parameter in comparing algorithms.*
- *If we say that an algorithm A runs faster than algorithm B, we need to make sure that they are running the same input data size.*
- *The input data could be, for example, the size of the array to be processed, number of characters in a file, number of records in a database, etc.*
- *If we increase the input size, will algorithm A still be faster than B?*
- *Therefore, we need to express the number of basic operations in terms of the input data size.*

Let’s work through some examples to determine the running times in terms of basic operations.

Example 1: Algorithm to find the largest integer in an array of n integers.

```
public static int findLargest(int[] arr)
{
    int largest = arr[0];
    int index = 1;
    while (index < arr.length)
    {
        if (arr[index] > largest)
            largest = arr[index];
        index++;
    }
    return largest;
}
```

Example 2: Algorithm to evaluate a polynomial

Example 3: Algorithm to find the weighted average of n items

Determine the average response time for visiting n websites, given the response time for each website and the number of times that website is visited.

Ok, let's compare algorithms….

- ***Suppose that we have three algorithms with the following run times (or run times proportional to):***
  - ***Algorithm A: 5000 n + 1000***
  - ***Algorithm B: 200 n^2 + 500 n***
  - ***Algorithm C: 1.1^n***
- ***Which algorithm is the best?***
- ***From the calculations for n=10, n=100, n=1000, and n=10,000 we conclude that Algorithm A performs the best in the "long run" or for "large enough values of n".***
- ***This is refered to as the asymptotic complexity***
- ***WE COMPARE ALGORITHMS FOR LARGE VALUES OF THE INPUT SIZE OR IN OTHER WORDS, BASED ON THEIR ASYMPTOTIC COMPLEXITIES***

Now comes the Big O notation….

- ***Compare the following algorithms:***

- ***Algorithm A: $5\ n + 10$***
- ***Algorithm B: $4\ n + 250$***
- ***Algorithm C: $3n^2 + 5n$***
- ***Algorithm D: $2\ n^2 - 500$***

- ***We say that Algorithms A and B are in the "same league" for large values of n.***
- ***Similarly Algorithms C and D are in the "same league" for large values of n.***
- ***The Big O notation classifies algorithms into the "same league".***
- ***In other words, we say that the complexity of algorithms A and B is $O(n)$ and the complexity of algorithms C and D is $O(n^2)$.***

NOW A LITTLE MATH….

FORMAL DEFINITION OF BIG O OR THE O( ) NOTATION

Deriving the Big O is easy!

- ***Replace all additive constants in the run time with the constant 1.***
- ***Retain only the highest order term in this modified run time.***
- ***If the highest order term is 1, then the order is O(1).***
- ***If the highest order term is not 1, remove the constant (if any) that multiplies the term.***
- ***You are left with the order.***

Examples

Derive the order of complexities (Big O) for the following functions (that is, run times expressed in terms of the input size n)

EXAMPLES FOR DERIVING THE BIG O (cont'd.)

PRACTICAL EXAMPLES

O(1): Constant Time Complexity

(Means that the algorithm requires the same fixed number of steps irrespective of the size of the task)

O(n): LInear Time Complexity

O(n^2): Quadratic Time Complexity

O(log n): Logarithmic Time Complexity

O(n log n): En Log En Time Complexity

O(n^3): Cubic Time Complexity

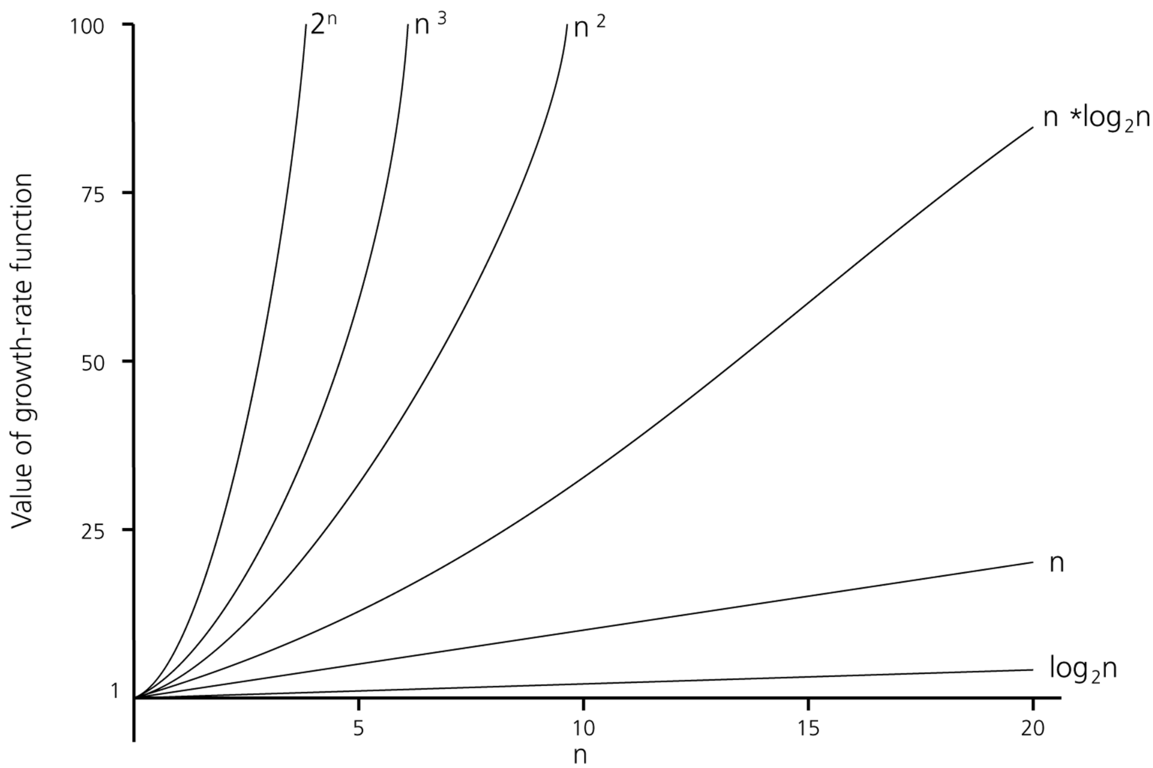
$O(k^n)$ Exponential Time Complexity

"Brute Force" or exhaustive search through all possible combinations.

Example: Breaking a secret key stored as a n-bit binary number.

COMPARISON OF GROWTH RATES

(b)



(a)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

ESTIMATION PROBLEMS WITH BIG O

1. An algorithm takes 1 ms to run when the input size is 1000. Approximately, how long will the algorithm take to process an input size of 5000 if it has the following orders of complexities:

   a. Linear
   b. Quadratic
   c. nlogn

2. Algorithm A takes 10 ms to solve a problem of size 1000. Algorithm B takes 100 ms to solve a problem of size 10,000. Algorithm A's complexity is quadratic while Algorithm B's complexity is cubic. How large a problem can each solve in 1 second?

3. Software packages A and B spend exactly $T_A = c_A n^2$ and $T_B = c_B n^3 + 500$ milliseconds to process n data items, respectively, where c_A and c_B are some constants. During a test, A takes 1000 milliseconds and B takes 600 milliseconds to process n=100 data items. Which package is better for processing 10 data items? Which package is better for processing 1000 data items? (Show steps).

OTHER ORDERS OR COMPLEXITY

Although we will use the Big O primarily as a measure of algorithm complexity in this course, there are three other types of complexity related to Big O.

We redefine Big-O to place it in context.

Big O: A growth function T(N) is O(F(N)) if there are positive constants c and N_0 such that T(N) <= cF(N) when N>=N_0 .

Big Omega: A growth function T(N) is Ω(F(N)) if there are positive constants c and N_0 such that T(N) >= cF(N) when N>=N_0 .

Big Theta: A growth function T(N) is Θ(F(N)) if and only if T(N) is O(F(N) and T(N) is Ω(F(N)).

Little-O: A growth function T(N) is o(F(N)) if and only if T(N) is O(F(N) and T(N) is is not Θ(F(N)).

ORDER ARITHMETIC

SOME SIMPLE RULES OF THUMB

Best case, worst case and average case complexity

- The best-case running time of an algorithm is the running time under ideal or best conditions.
- The worst-case running time of an algorithm offers a guarantee that the running time will never be worse than it.
- The average running time of an algorithm is the expected running time on the average.
- If there is no reference to worst-case or average complexity order, it usually means worst-case.

Find the largest integer in an array of integers (size of the array is n) - Revisited

```
public static int findLargest(int [] a){
  int largest = a[0];          → t
  int i = 1;                   → t
  while (i<a.length)           → nt
  {
      if (a[i] > largest)      → (n-1)t
          largest = a[i];      → (n-1)t
      i++;                     → (n-1)t
  }
  return largest;              → t
}
```

Summary

- The most reliable way to measure the run time of an algorithm is to count the number of basic operations it performs.
- Express the count of the basic operations of an algorithm as a function of the input size n.
- Then express the function in terms of the Big O.
- Big O refers to the asymptotic growth of the function for large values of n.
- This is the measure that we use to compare algorithms.

Summary

- Algorithm run time complexity can be best case, worst case or average case.
- Default algorithm time complexity refers to worst case.
- Deriving Big O is easy!
- Some typical run time orders are: O(1) (constant), O(log n) (logarithmic), O(n) (linear),

O(n log n), O(n^2) (quadratic), O(n^3) (cubic), O(k^n) (exponential).