CIS 351-Data Structure-Algorithm Analysis
Feb 25, 2020

Dr. Farzana Rahman

Syracuse University

Example problem

- Assume we have a lab assignment where you are asked to:
  - Count the number of steps in this building
- You decide to keep track on a piece of paper.
- As you climb up each step, you make a mark on your page. You count 100 steps.

- How much work did this assignment require? Include:
  - Number of steps going up
  - Number of steps going down
  - Number of marks made on paper

Same Three Algorithms / Different Building

- Lab now takes place in a building at any other place.
- how much work is required for a single algorithm?
  - For a building with 1000 steps (10 times the steps)
  - For a building with 2000 steps (2 times the steps as before)
  - For a building with *n* steps (unknown number) ???

Algorithm analysis

- Correctness?
  - Important, but this class won't focus on proofs of correctness
- Clarity/simplicity?
  - Important, but not what we are taking about today
- Space efficiency?
  - Yes! This is something we want to understand
- **Time Efficiency?**
  - **Yes! This is usually the main concern**

Algorithm analysis

- Analysis must account for input size
  - How does the running time change as the input size increases?

Algorithm analysis

- Goal is to analyze algorithms, not programs.
- Running time of programs is subject to:
  - Programming language
  - Speed of the computer
  - Hardware
  - Compiler version….

If Not Time, Then What?

- Number of steps that the algorithm takes to complete
- **Goal:** develop a function that maps from input size to the number of steps

Basic Operations

- **Goal restated:** develop a function that maps from input size to the number of times the "basic operation" is performed

Guideline:

- Usually happens in inner-most loop
- If chosen well, count will be proportional to execution time

Growth/Complexity Functions

- Map our algorithm to a complexity function
- **Informal description:** Growth functions are categorized according to their **dominant (fastest growing) term**
- Constants and lower-order terms are discarded
- **Examples:**
  - $10n \sim O(n)$
  - $5n^2 + 2n + 3 \sim O(n^2)$
  - $n\log n + n \sim O(n\log n)$

Why Drop the Constants?

- Despite constants, functions from slower growing classes will always be faster eventually
- Real goal is to understand the relative impact of increasing input size
- Contribution of lower-order terms becomes insignificant as input size increases

Big-O Notation

- Simplify the math
- What's the largest term of the equation?
- Ignore constants that are multiplied to that term.

What is Big O

- Big O notation is used to describe the performance or complexity of an algorithm.
- Big O specifically describes the worst case scenario, and can be used to describe the execution time required by an algorithm.

O(1)

- O(1) describes an algorithm that will always execute in the same time regardless of the size of the input data set.

```
bool IsFirstElementNull(String[] strings)
{
    if(strings[0] == null)
    {
        return true;
    }
    return false;
}
```

O(N)

- O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of input data set.

```
bool ContainsValue(String[] strings, String value)
{
    for(int i = 0; i < strings.Length; i++)
    {
        if(strings[i] == value)
        {
            return true;
        }
     }
    return false;
}
```

$O(N^2)$

- $O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
- Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.

```
bool ContainsDuplicates(String[] strings) {
        for(int i = 0; i < strings.Length; i++)
        {
                for(int j = 0; j < strings.Length; j++) {
                        if(i == j) // Don't compare with self  {
                                continue;
                        }
                        if(strings[i] == strings[j]) {
                                return true;
                        }
                }
        }
        return false;
}
```

O(logN)

- The iterative halving of data sets in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase

- e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds.

- Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved.

- This makes algorithms like binary search extremely efficient when dealing with large data sets

Meaning

- Big O = Upper bound

  = The algo is at least this fast

- Big Omega = Lower bound

  = The algo is at least this slow

- Big Theta= Average bound

We are mostly interested in Upper and Lower bound

Example

```
bool contains (int key, int * array)
{
  for (int i = 0; i < n; i++ )
  {
     if ( key == array[i])
          return true;
  }
   return false;
}
```

Example

- Best Case: 1 comparison, $O(1)$
- Worst Case: n comparisons, $O(n)$
- Average Case: $\frac{n+1}{2}$ comparisons, $O(n)$

Source: OpenDSA Project

Formal Definition of Big-O

Big O

$f(n) \in O(g(n))$ iff there exist positive constants c and N such that for all $n > N$,

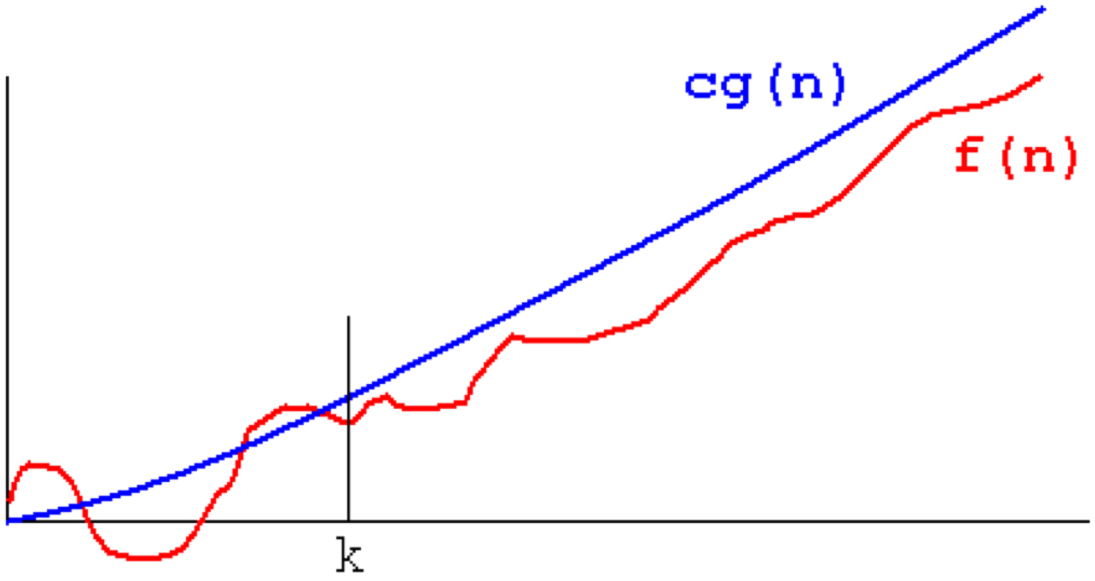
$$f(n) \leq cg(n)$$

- Informal rule of "dropping constants" follows immediately:
  - $50n \stackrel{?}{\in} O(n)$
  - Yes! choose c = 50, N = 0, clearly
  - $50n \leq 50n$ for all $n > 0$

Source: OpenDSA Project

- In the image below, f(n) = O(g(n)). In other words, there are positive constants c and k, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$.

- Importance of k is that after **'k'** this Big O will stay true, no matter what value of n.



What steps to follow

- STEP 1: Select a measure of input size and a basic operation
- STEP 2: Develop a function $T(n)$ that describes the number of times the basic operation occurs as a function of input size
- STEP 3: Describe $T(n)$ using order notation (Big-O)

Source: OpenDSA Project

Example

```
x = 0;
for (int i=1; i<=n; i=i*2)
{
     x = x + 1;
}
```

- For the above algorithm, outside the loop a **p** constant number of operations are performed.
- Also each iteration of the loop performs a constant number **k** of operations.
- Counting the total number of iterations of the loop is more complicated. Let us look at the value of i in each iteration:

Iteration number	Value of i
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
4	$8 = 2^3$
…	…
1 + log n	$n = 2^{\log n}$

```
x = 0;
for (int i=1; i<=n; i=i*2)
{
      x = x + 1;
}
```

- Hence, the number of **iterations** performed by the loop is **1 + log n**
- The total number of **operations** performed by the loop is

  $T(n) = P + k(1 + \log n) = P + k + k \log n.$

- The dominating term is **k log n**, so the time complexity is **O(log n)**

Typical Growth Rates

- *n* is problem size

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

Big-O Notation

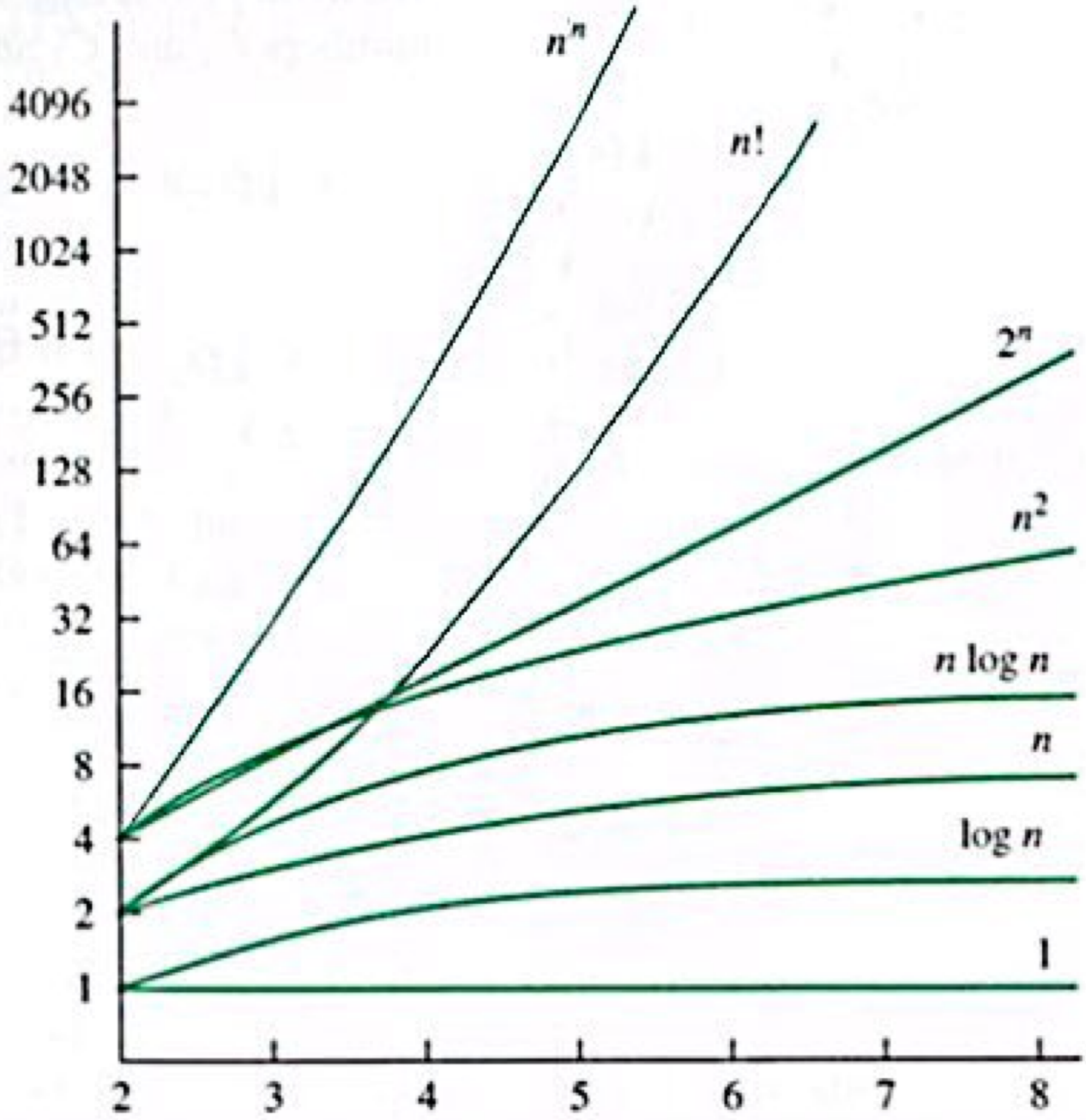
- O(1): Constant
- O(log n): Logarithmic
- O(n): Linear
- O(n log n)
- $O(n^2)$: Quadratic
- $O(2^n)$: Exponential
- O(n!): Factorial

Order-of-Magnitude Analysis and Big O Notation

(a)

Function	n = 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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A comparison of growth-rate functions: a) in tabular form



$n!$

2^n

n^3

n^2

$n \log n$

n

$\sqrt{n} = n^{1/2}$

$\log n$

1