Data Structures
Analysis of Algorithms

Outline

1. Introduction
2. Big-Oh and other Notations
3. Typical Growth Rates
4. Sum and Product Rules
5. Examples
6. Summary

What is a Good Algorithm?

The efficiency of an algorithm is measured by 2 important factors :

T(n): Time / Steps taken by an algorithm for input of size n

S(n): Space (memory cells) required by an algorithm for input of size n

Measuring the Running Time

How should we measure the running time of an algorithm?

Experimental Study

- Write a program that implements the algorithm.
- Run the program with data sets of varying size and composition.
- Use command **time** to get an accurate measure of the actual running time.

Limitations of Experimental Studies

- It is necessary to implement and test the algorithm in order to determine its running time.

- Experiments can be done only on a limited sets of inputs, and may not be indicative of the running time on other inputs not included in the experiment.

- In order to compare two algorithms, the same hardware and software environments should be used.

Beyond Experimental Studies

We will develop a general methodology for analyzing running time of algorithms. This approach

- Uses a high-level description of the algorithm instead of testing one of its implementations.
- Takes into account all possible inputs
- Allows one to evaluate the efficiency of any algorithm in a way that is independent of the hardware and software environment.

PseudoCode

A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure and algorithm.

- Eg. **Algorithm arrayMax(A,n)**:

```
Input: An array A storing n integers
Output: The maximum element in A.
currentMax ⟵ A[0]
for i ⟵ 1 to n-1 do
        if currentMax<A[i]
                then currentMax ⟵ A[i]
return currentMax
```

PseudoCode

It is more structured than usual prose but less formal than a programming language.

- **Expressions:**
  - Use standard mathematical symbols to describe numeric and boolean expressions.
  - Use $\longleftarrow$ for assignment ("=" in C)
  - Use = for the equality relationship ("= =" in C)

PseudoCode

- **Programming Constructs:**
  - decision Structures : if….then…..(else….)
  - while-loops : while…..do
  - for-loops: for…do
  - array indexing : A[i], A[i, j]

Analysis of Algorithms

- **Primitive Operation:** Low-level operation independent of programming language. Can be identified in pseudo-code. For eg:
  - Data movement (assign)
  - Control (branch, subroutine call, return)
  - arithmetic an logical operations (e.g. addition, comparison)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

A Computational Model

To summarize algorithm runtimes, we can use a computer independent model

- instructions are executed sequentially
- count all assignments, comparisons, and increments
- every simple instruction takes one unit of time (cost of executing instruction)

Simple Instructions

Count the simple instructions

- assignments have cost of 1
- comparisons have a cost of 1
- let's count all parts of the loop

for (int j = 0; j < n; j++)

• j=0 has a cost of 1, j<n executes n+1 times, and j++ executes n times for a total cost of 2n+2

- each statement in the repeated part of a loop have a cost equal to number of iterations

Examples

		Cost
`sum = 0;`	->	1
`sum = sum + next;`	->	1 *Total Cost: 2*

		Cost
`for (int i = 1; i <= n; i++)`	->	1 + n+1 + n = 2n+2
`  sum = sum++;`	->	n *Total Cost: $3n + 2$*

		Cost
`k = 0`	->	1
`for (int i = 0; i < n; i++)`	->	2n+2
`  for (int j = 0; j < n; j++)`	->	n(2n+2) = $2n^2$ +2n
`    k++;`	->	n^2 *Total Cost: $3n^2 + 4n + 3$`*

Case Study: Linear Search

	Cost
`for(i = 0; i < n; i++)`	$2n+2$
`if(item[i] == search_item)`	n
`return i;   //index of item found`	*0 or 1*
`return -1;   //item Not Found`	*0 or 1*

Total Cost = $3n+3$

Different Cases

- The total cost of sequential search is 3n+3
  - But is it always exactly 3n+3 instructions?
  - How many times will the loop actually execute?
    - that depends
  - If search_item is found at index 0: ______ iterations
    - best case
  - If search_item is found at index n-1:______ iterations
    - worst case

Example: Sorting

INPUT
sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7

OUTPUT
a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

Correctness (requirements for the output)
For any given input the algorithm halts with the output:
- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time
Depends on
- number of elements (n)
- how (partially) sorted they are
- algorithm

Insertion Sort

Strategy

- Start "empty handed"
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤ A[2]≤ …≤A[n]

```
for j←2 to n do
    key ←A[j]
    Insert A[j] into the sorted sequence
    A[1..j-1]
    i←j-1
    while i>0 and A[i]>key
      do A[i+1]←A[i]
         i--
    A[i+1]←key
```

Analysis of Insertion Sort

	cost	times
for j←2 **to** *n* **do**	c_1	n
key←A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1..j-1]	0	n-1
i←j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1] ← key	c_7	n-1

$$\text{Total time} = n(c_1+c_2+c_3+c_7) + \sum_{j=2}^{n} t_j \ (c_4+c_5+c_6) - (c_2+c_3+c_5+c_6+c_7)$$

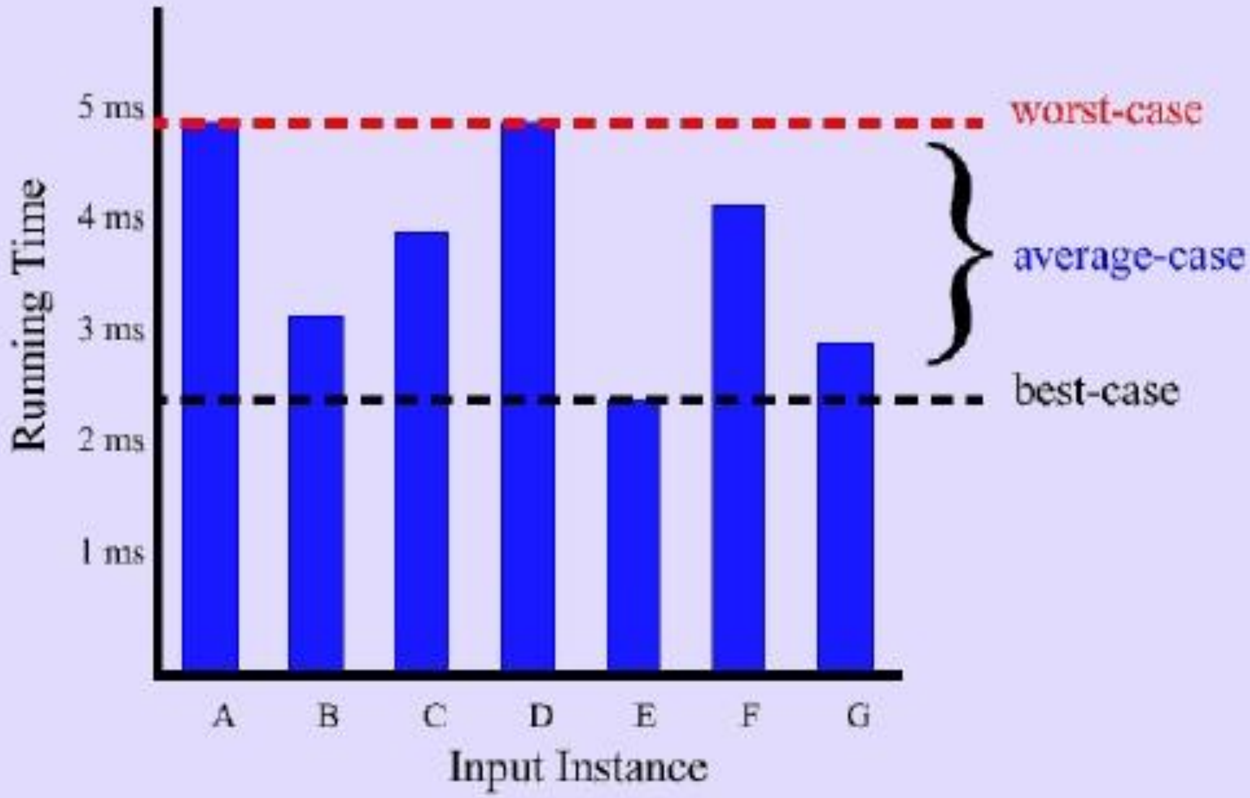
Best/Worst/Average Case

Total time $= n(c_1+c_2+c_3+c_7) + \sum^{n}_{j=2} t_j \ (c_4+c_5+c_6)$
$- (c_2+c_3+c_5+c_6+c_7)$

- **Best case**: elements already sorted; $t_j=1$, running time = f(n), i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order; $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

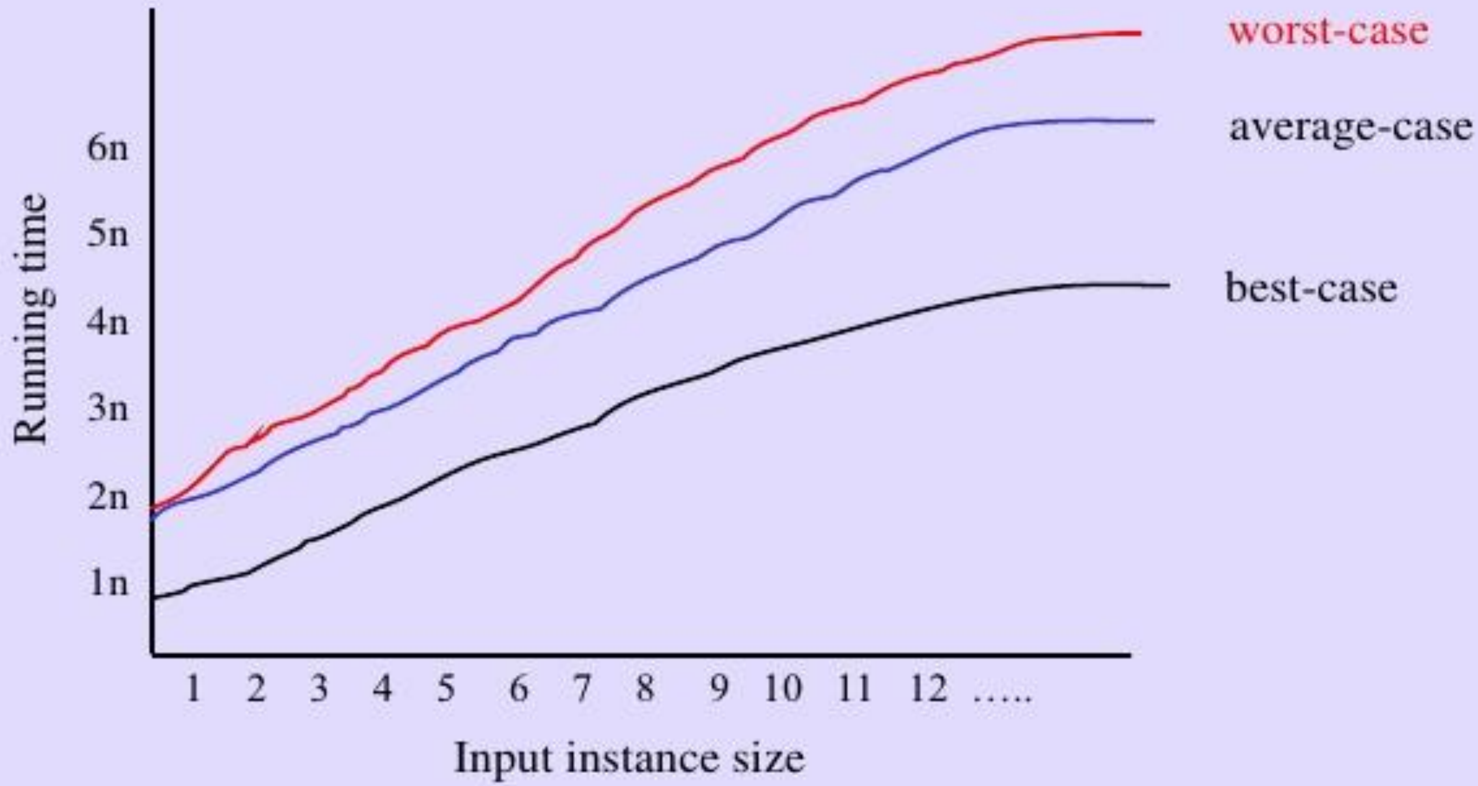
Best/Worst/Average Case (2)

- For a specific size of input n, investigate running times for different input instances:



Best/Worst/Average Case (3)

For inputs of all sizes:



Best/Worst/Average Case (4)

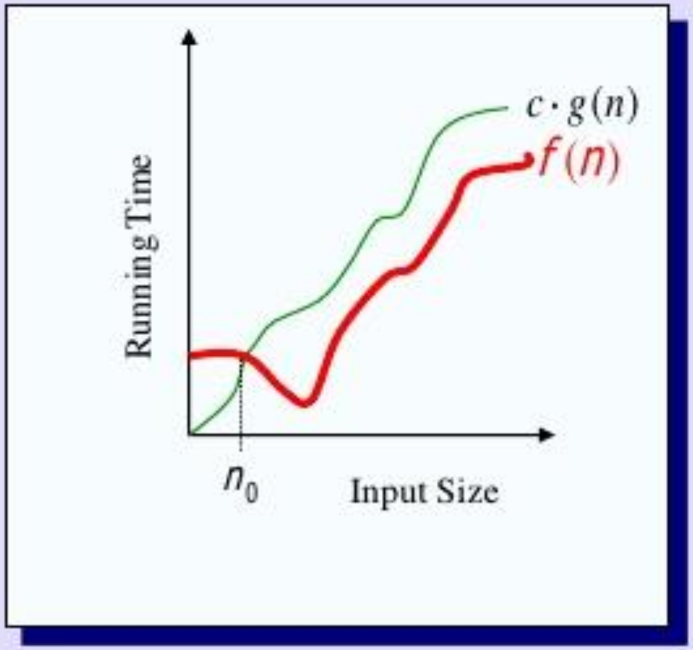
- **Worst case** is usually used: It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- For some algorithms **worst case** occurs fairly often
- **Average case** is often as bad as the **worst case**
- Finding **average case** can be very difficult

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
  - like "rounding": $1{,}000{,}001 \approx 1{,}000{,}000$
  - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit*.
  - Asymptotically more efficient algorithms are best for all but small inputs

Asymptotic Notation

- The "big-Oh" *O*-Notation
  - asymptotic upper bound
  - f(n) = O(g(n)), if there exists constants *c* and n_0, *s.t.* $\mathbf{f(n) \leq c\ g(n)}$ for n $\geq n_0$
  - f(n) and g(n) are functions over non-negative integers
- Used for *worst-case* analysis

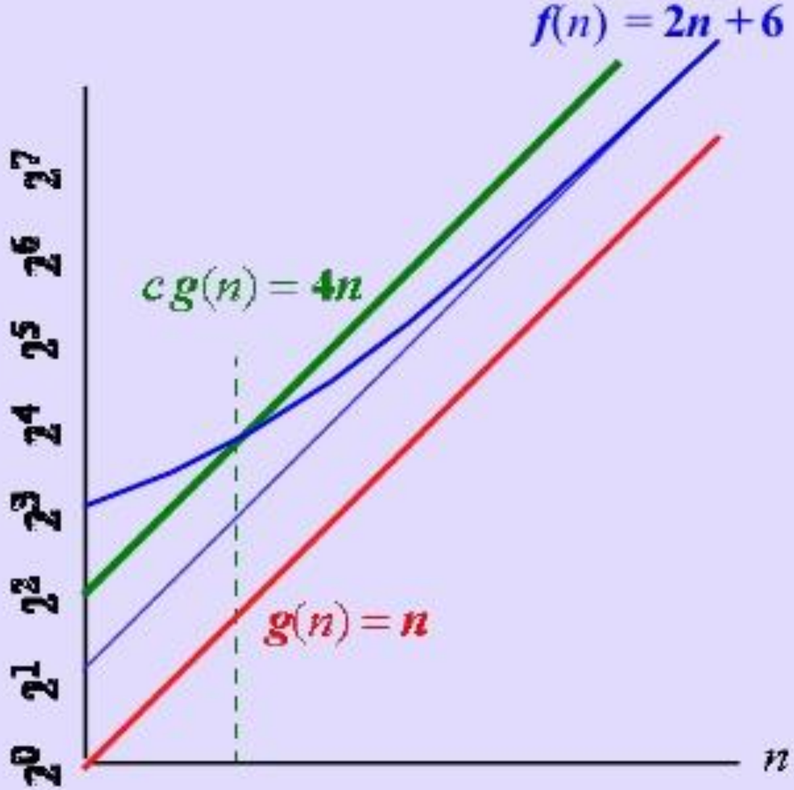


Big – Oh Example

- Suppose $T(n) = 3n^3 + 2n^2$
- Claim: For $n_0 = 1$ and $c = 5$, $3n^3 + 2n^2 \leq 5n^3$, $\forall n \geq 1$
- Such a claim may be proved using Mathematical Induction
- If this claim is true, then the complexity (growth rate) can be expressed as $O(n^3)$
- Technically, we can also say that this $T(n)$ is $O(n^4)$
- But, that is a weak statement, and it is understood that we need to find the "least upper bound"

Example

For functions f(n) and g(n) there are positive constants c and n_0 such that: $f(n) \le c\, g(n)$ for $n \ge n_0$



conclusion:

2n+6 is O(n).

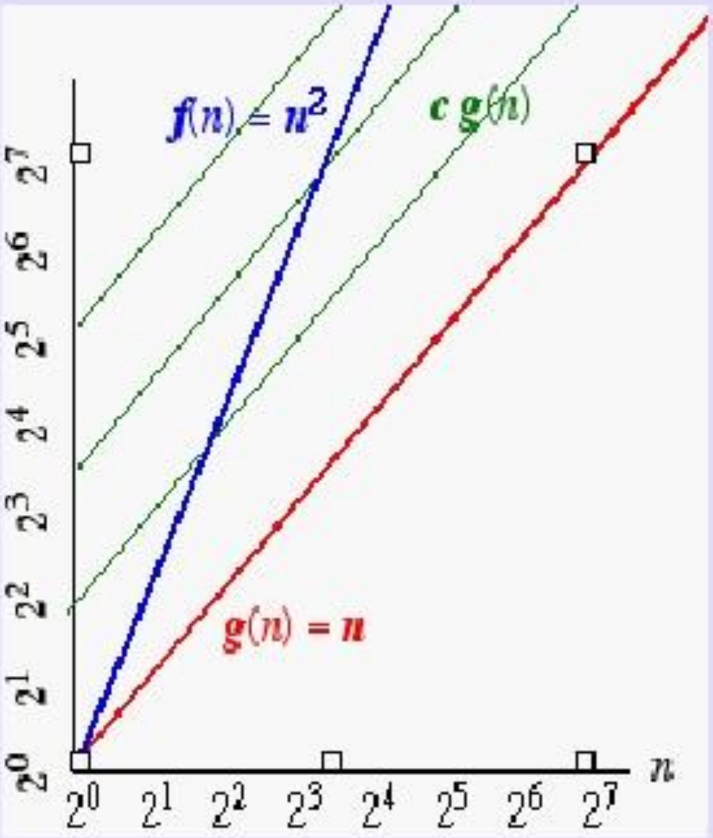
Another Example

On the other hand…

n^2 is not $O(n)$ because there is no c and n_0 such that:

$$n^2 \leq cn \text{ for } n \geq n_0$$

The graph to the right illustrates that no matter how large a c is chosen there is an n big enough that $n^2 > cn$)



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors.
  - $50\ n \log n$ is $O(n \log n)$
  - $7n - 3$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- Note: Even though $(50\ n \log n)$ is $O(n^5)$, it is expected that such an approximation be of as small an order as possible

Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

```
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]   ← 1 step
    A[i] ← a/(i+1)
return array A
```

$\}$ i iterations with i=0,1,2...n-1

$\}$ n iterations

Analysis: running time is $O(n^2)$

A Better Algorithm

Algorithm prefixAverages2(X):

Input: An *n*-element array X of numbers.

Output: An *n*-element array A of numbers such that A[*i*] is the average of elements X[0], ... , X[*i*].

$s \leftarrow 0$

for $i \leftarrow 0$ **to** n **do**

  $s \leftarrow s + X[i]$

  $A[i] \leftarrow s/(i+1)$

return array A

Analysis: Running time is O(n)

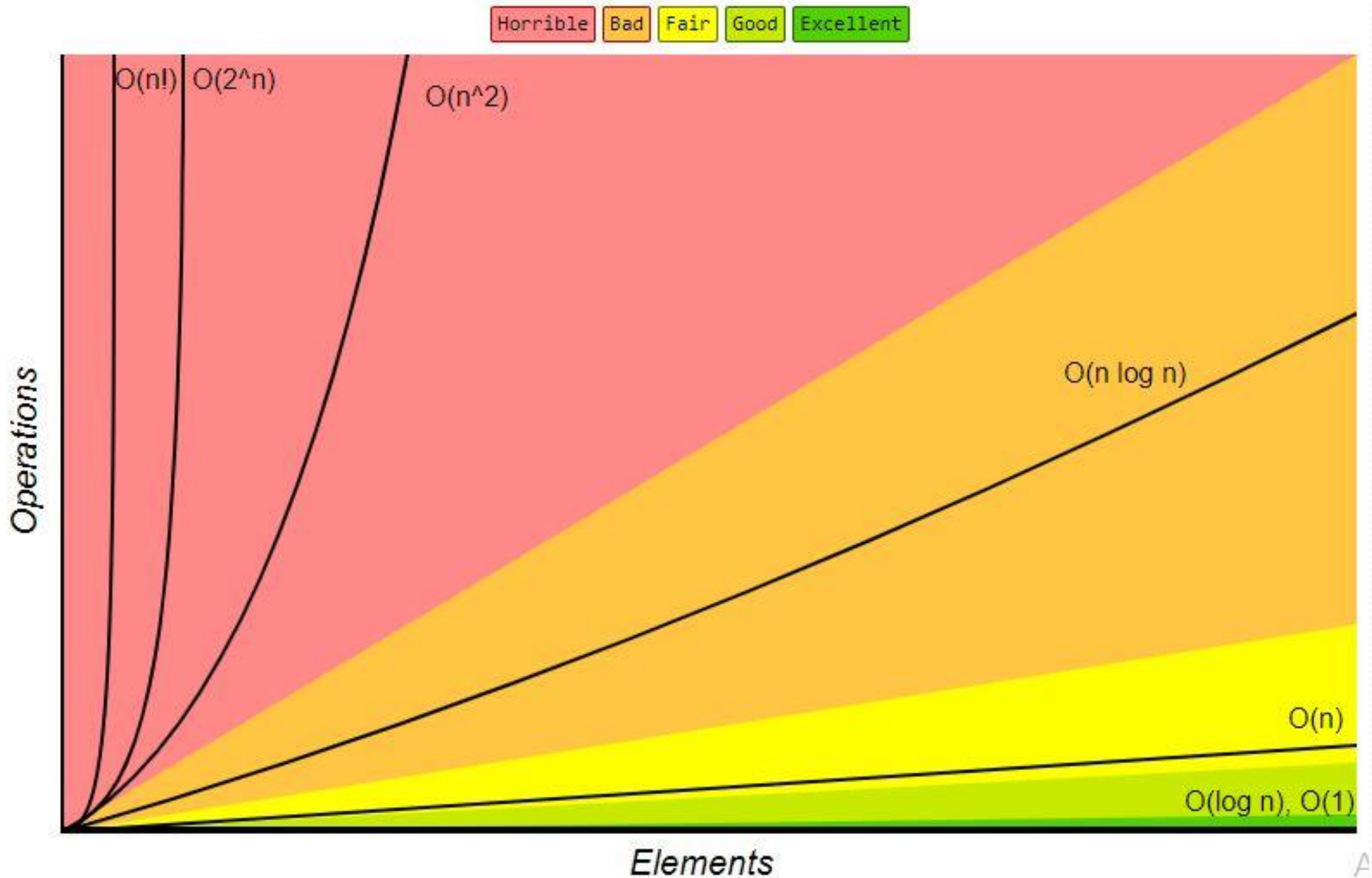
Asymptotic Notation *(terminology)*

- Special classes of algorithms:
  - Logarithmic: O(log n)
  - Linear: O(n)
  - Quadratic: $O(n^2)$
  - Polynomial: $O(n^k)$, $k \geq 1$
  - Exponential: $O(a^n)$, $a > 1$
- "Relatives" of the Big-Oh
  - Ω (f(n)): Big Omega -asymptotic *lower* bound
  - Θ (f(n)): Big Theta -asymptotic *tight* bound

Asymptotic Analysis of Running Time

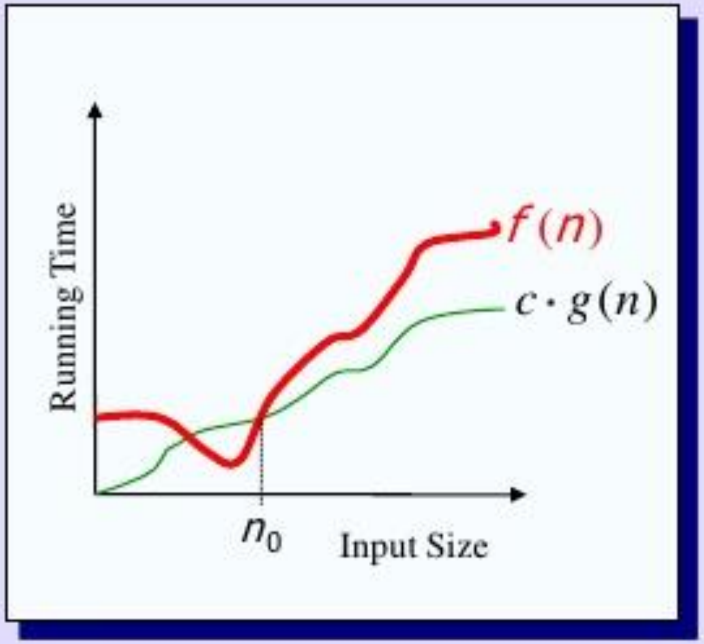
- Use O-notation to express number of primitive operations executed as function of input size.
- Comparing asymptotic running times
  - an algorithm that runs in $O(n)$ time is better than one that runs in $O(n^2)$ time
  - similarly, $O(\log n)$ is better than $O(n)$
  - hierarchy of functions: $\log n < n < n^2 < n^3 < 2^n$
- Caution! Beware of very large constant factors. An algorithm running in time 1,000,000 n is still $O(n)$ but might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Big 0 Complexity Chart



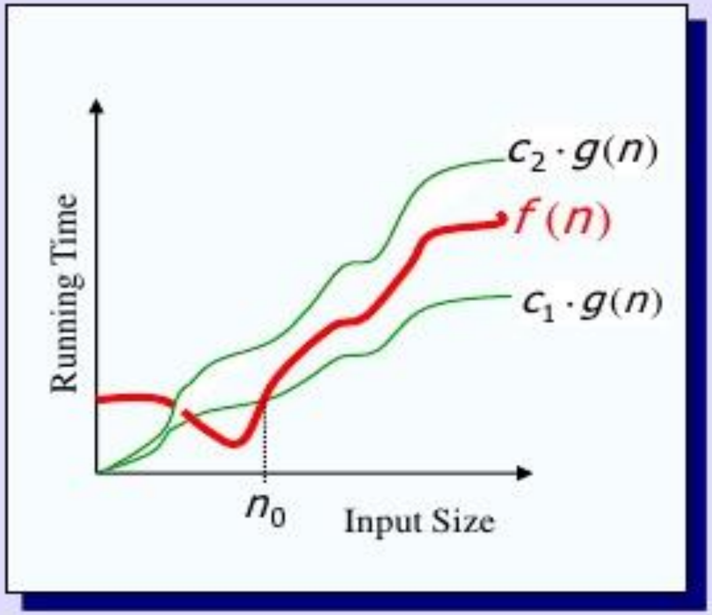
Asymptotic Notation

- The "big-Omega" Ω–Notation
  - asymptotic lower bound
  - $f(n) = \Omega(g(n))$ if there exists constants c and n_0, *s.t.* $\mathbf{c\ g(n) \le f(n)}$ for $n \ge n_0$
- Used to describe *best-case* running times or lower bounds of algorithmic problems
  - E.g., lower-bound of searching in an unsorted array is $\Omega(n)$.



Asymptotic Notation

- The "big-Theta" Θ–Notation
  - asymptotically tight bound
  - $f(n) = \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, *s.t.* $\mathbf{c_1\, g(n) \le f(n) \le c_2\, g(n)}$ for $n \ge n_0$
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $O(f(n))$ is often misused instead of $\Theta(f(n))$



Rule of Sums

Sequential Segments

<

. . . Program Segment 1

. . .

>

<

. . . Program Segment 2

. . .

>

$T_1(n)$

$T_2(n)$

$$T(n) = T_1(n) + T_2(n)$$

Sum Rule

Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T(n)$ is $O(\max(f(n), g(n)))$

Rule of Products

Iteration

<

. . . Program Segment

. . .

>

executed $T_2(n)$ times

$T_1(n)$

$$T(n) = T_1(n) \times T_2(n)$$

Product Rule

Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T(n)$ is $O(f(n)g(n))$

Note that $O(cf(n))$ is same as $O(f(n))$

Empirical Verification: Ratio Analysis

- Is it possible to empirically verify if the running time of an algorithm is $O(f(n))$?
- Implement the algorithm and note down the running time $T(n)$ for different values of n.
- Now find the ratio $T(n)/f(n)$, for those different values on n.
- $f(n)$ is a tight bound if this ratio converges to a positive constant
- If the ratio converges to 0, then $f(n)$ is an over-estimate
- $f(n)$ is an under-estimation, if this ratio diverges.

Summary

- Time / Space complexity of an algorithm are expressed in notations such as Big-Oh and Big-Theta
- These notations bring out the growth rate of time / space wrt the size of the input
- These notations enable us to avoid exact calculations of number of "basic steps" or memory space required — overall growth rate can be estimated based on growth rates of components
- It is possible to come up with several designs of algorithms for a problem, and analysis is important to choose the appropriate one
- It is also possible to perform empirical ratio analysis to determine / verify time complexity of an algorithm

What Next?

- We will take analysis of recursive algorithms in the next lecture
- Meanwhile, read on complexity analysis of algorithms from standard Books
- Later in this week, we will start our discussions on implementation of linear structures
- Topics like trees, graphs, sets, and hashing will follow that