CS302 – Analysis and Design of Algorithms

Algorithm Analysis

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω −Notations
Comparing Functions
Exercises

Introduction

- Algorithm analysis: predicting the resources that the algorithm requires.
- Resources can be:
  - Computational time → most common and important
  - Memory
  - Communication bandwidth
  - Energy consumption
- Computational model:
  - Random Access Machine (RAM) with one processor
  - Execute one operation at time; no concurrent operations
  - Each instruction or data access takes a constant amount of time

Introduction

- Our RAM model have the following instructions:
  - Arithmetic: +, -, %, *, /, floor, ceiling
  - Data movement: load, store, copy
  - Control: conditional and unconditional branch, call, return
  - Bitwise: AND, OR, NOT, XOR, shifting
- RAM model has the following data types:
  - Float, int, char
- RAM model doesn’t have instructions for:
  - Sorting, exponentiation, summation

Introduction

- We cannot compute the running time of an algorithm by implementing it on a programming language and compute the execution time.
  - It depends on the programming language.
  - It depends on the compiler used to compile the program
  - It depends on the libraries used in the program
  - It depends on the architecture and the specifications of your machine
  - It depends on how the user inputs the data

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω —Notations
Comparing Functions
Exercises

Analyzing Insertion Sort

- The running time depends on the input size.
  - Larger arrays take longer than smaller arrays.

- Insertion sort depends on the size of the array and how sorted it is.

- The input size has the greatest effect on the performance of the algorithm.
  - Hence, we will describe the running time as a function of the size of its input.
  - To do so, we need to define the terms **running time** and **input size** carefully.

Analyzing Insertion Sort

- The definition of “input size” depends on the application.
  - For sorting an array, the input size means the number of elements in the array.
  - For multiplying two integers, the input size refers to the bit size of the number.
  - For graph-based algorithms, the input size can be the number of edges and vertices.
- The running time is the number of instructions and data accesses executed.
  - Should be independent of any particular computer, but within the framework of the RAM model.
  - We assume that each line in our pseudocode, takes a constant amount of time, c_k, where k is the line number.

Analyzing Insertion Sort

```
INSERTION-SORT(A, n)                                          cost    times
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i - 1].
    j = i - 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j - 1
    A[j + 1] = key
```

Analyzing Insertion Sort

```
INSERTION-SORT(A, n)                                   cost   times
for i = 2 to n                                         c_1    n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Analyzing Insertion Sort

Note that the loop condition is executed one time more than the loop body – that is when the condition is false

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n - 1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.		
$\quad j = i - 1$		
$\quad$ **while** $j > 0$ and $A[j] > key$		
$\quad\quad A[j + 1] = A[j]$		
$\quad\quad j = j - 1$		
$\quad A[j + 1] = key$		

Analyzing Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad$ *key* $= A[i]$	c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$\quad$ $j = i - 1$		
$\quad$ **while** $j > 0$ and $A[j] >$ *key*		
$\quad\quad$ $A[j+1] = A[j]$		
$\quad\quad$ $j = j - 1$		
$\quad$ $A[j+1] =$ *key*		

Analyzing Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n - 1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.	0	$n - 1$
$j = i - 1$	c_4	$n - 1$
while $j > 0$ and $A[j] > key$		
$A[j + 1] = A[j]$		
$j = j - 1$		
$A[j + 1] = key$		

Analyzing Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n - 1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.	0	$n - 1$
$j = i - 1$	c_4	$n - 1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j + 1] = A[j]$		
$j = j - 1$		
$A[j + 1] = key$		

Analyzing Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n - 1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.	0	$n - 1$
$j = i - 1$	c_4	$n - 1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j + 1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j - 1$		
$A[j + 1] = key$		

Analyzing Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n - 1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.	0	$n - 1$
$j = i - 1$	c_4	$n - 1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j + 1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j + 1] = key$		

Analyzing Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n - 1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n - 1$
$j = i - 1$	c_4	$n - 1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
$j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
$A[j+1] = key$	c_8	$n - 1$

Analyzing Insertion Sort

- The running time of the algorithm, $T(n)$ is

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$\quad j = i-1$	c_4	$n-1$
$\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$\quad\quad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
$\quad\quad j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
$\quad A[j+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1) .$$

Analyzing Insertion Sort

- Even for two arrays of the same size, the running time depends on how sorted the array is.

- The best case happens when the array is already sorted.

- In this case, the while loop at line 5 and its body will not execute.
  - The loop header will be executed as condition only - so it is counted as a statement with cost $c_5(n-1)$

Analyzing Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$\quad j = i - 1$	c_4	$n-1$
$\quad$ **while** $j > 0$ and $A[j] > key$ $\quad c_5(n-1)$	$\cancel{c_5}$	$\cancel{\sum_{i=2}^{n} t_i}$
$\quad\quad \cancel{A[j+1] = A[j]}$	$\cancel{c_6}$	$\cancel{\sum_{i=2}^{n} (t_i - 1)}$
$\quad\quad \cancel{j = j - 1}$	$\cancel{c_7}$	$\cancel{\sum_{i=2}^{n} (t_i - 1)}$
$\quad A[j+1] = key$	c_8	$n-1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}
$$

Analyzing Insertion Sort

INSERTION-SORT(A, n)		cost	times
for $i = 2$ **to** n		c_1	n
$\quad key = A[i]$		c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.		0	$n-1$
$\quad j = i - 1$		c_4	$n-1$
$\quad$ **while** $j > 0$ and $A[j] > key$	$c_5(n-1)$	~~c_5~~	~~$\sum_{i=2}^{n} t_i$~~
~~$\quad\quad A[j+1] = A[j]$~~		~~c_6~~	~~$\sum_{i=2}^{n} (t_i - 1)$~~
~~$\quad\quad j = j - 1$~~		~~c_7~~	~~$\sum_{i=2}^{n} (t_i - 1)$~~
$\quad A[j+1] = key$		c_8	$n-1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= \underbrace{(c_1 + c_2 + c_4 + c_5 + c_8)}_{a} n - \underbrace{(c_2 + c_4 + c_5 + c_8)}_{b} .
\end{aligned}
$$

Analyzing Insertion Sort

INSERTION-SORT(A, n)		*cost*	*times*
for $i = 2$ **to** n		c_1	n
$\quad key = A[i]$		c_2	$n - 1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.		0	$n - 1$
$\quad j = i - 1$		c_4	$n - 1$
$\quad$ **while** $j > 0$ and $A[j] > key$	$c_5(n-1)$	$\cancel{c_5}$	$\cancel{\sum_{i=2}^{n} t_i}$
~~$A[j + 1] = A[j]$~~		$\cancel{c_6}$	$\cancel{\sum_{i=2}^{n} (t_i - 1)}$
~~$j = j - 1$~~		$\cancel{c_7}$	$\cancel{\sum_{i=2}^{n} (t_i - 1)}$
$\quad A[j + 1] = key$		c_8	$n - 1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= \underbrace{(c_1 + c_2 + c_4 + c_5 + c_8)n}_{a} - \underbrace{(c_2 + c_4 + c_5 + c_8)}_{b} .
\end{aligned}
$$

The running time is thus a linear function of n.

$an + b$

Analyzing Insertion Sort

- The worst case arises when the array is in reverse sorted order.
  - That is, it starts out in decreasing order.

- The procedure must compare each element $A[i]$ with each element in the entire sorted subarray.
  - The procedure finds that $A[i] > key$ every time in line 5, and the while loop exits only when j reaches 0.
- So, we get the running time expressed as (the same first equation)

$$
\begin{aligned}
T(n) \;=\; & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\
& + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)\,.
\end{aligned}
$$

Analyzing Insertion Sort

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)\,.$$

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Analyzing Insertion Sort

- So, the worst-case running time is

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\ .
\end{aligned}
$$

- Thus, the running time is a quadratic function

$$an^2 + bn + c$$

Content

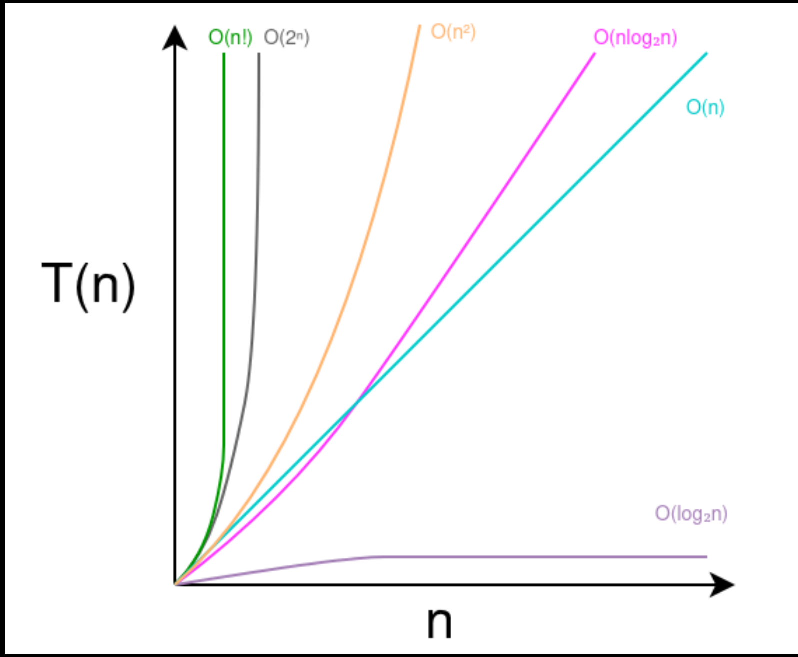
Worst-case and Average-case Analysis

- When analyzing an algorithm, we focus on the worst-case running time. Why?
  1. It gives an upper bound on the running time for any input.
     - If you know it, then you have a guarantee that the algorithm never takes any longer.
  2. For some algorithms, the worst case occurs fairly often.
     - In searching a database, the searching algorithm’s worst case often occurs when the information is not present in the database.
  3. The “average case” is often roughly as bad as the worst case.

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω −Notations
Comparing Functions
Exercises

Order of growth

- Order of growth: how the number of computational steps is increasing as the input size n grows.
- To get the order of growth, we ignore constants and low-order terms.
  - For the insertion sort algorithm, the order of growth is n^2.



Order of growth

- For example, suppose that an algorithm implemented on a particular machine takes $n^2/100 + 100n + 17$ microseconds on an input of size n.
  - If you consider some small values for $n = 5, 10, 90$, you find that the $1/100$ coefficient of n^2 is less significant than the 100 coefficient of n.
  - But, when you consider large values for $n = 200$, you will find that n^2 dominates the n.
  - Large data size is what reflects real-life problems.

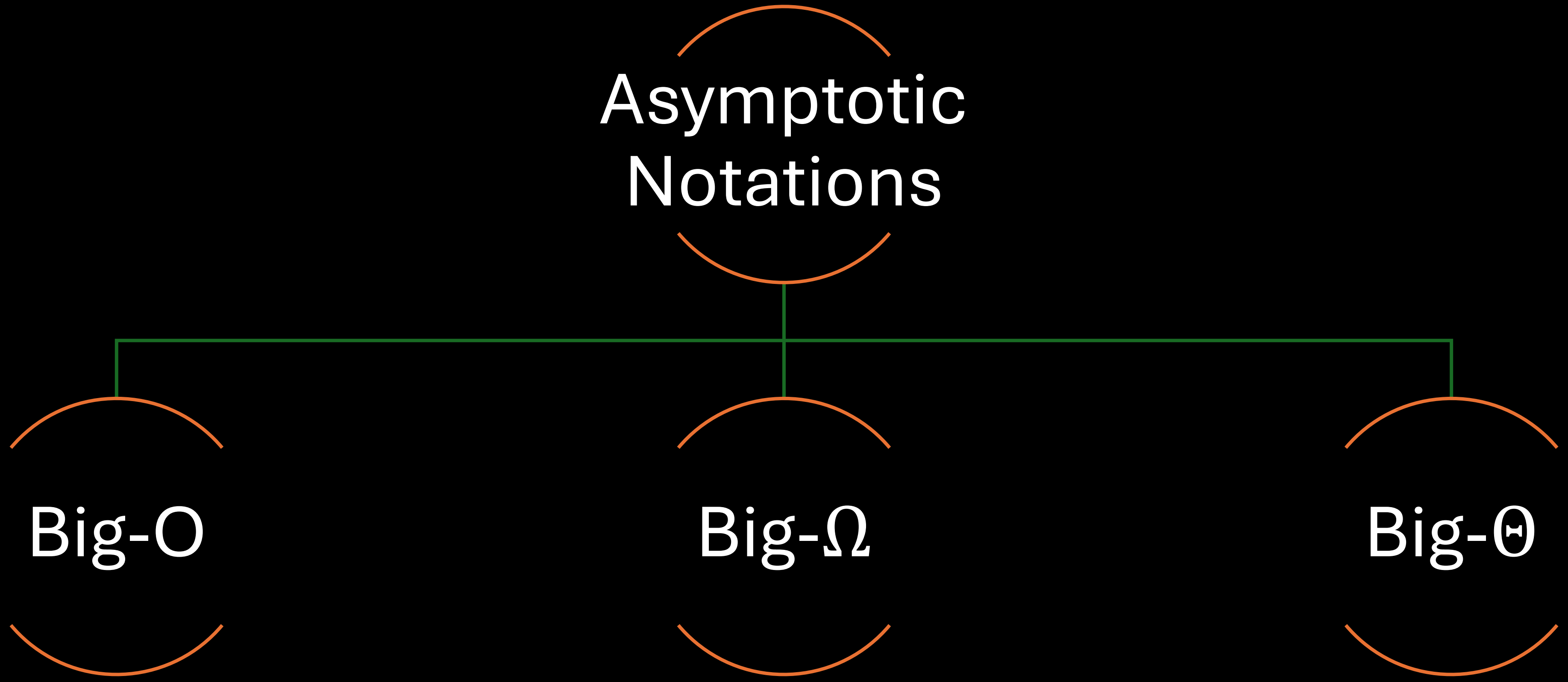
Order of growth

- When we consider only the dominant variable of high-order of a function, e.g., n^2, we are studying **asymptotic** efficiency.

- **Asymptotic analysis** is a method used to describe the behavior of algorithms or functions as their input size grows towards infinity.

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω —Notations
Comparing Functions
Exercises

Asymptotic Notations



Asymptotic Notations - O

- O-notation characterizes an **upper bound** on the asymptotic behavior of a function.
  - A function grows no faster than a certain rate, based on the highest-order term.
- For example, the function $7n^3 + 100n^2 - 20n + 6$.
  - Its highest-order term is $7n^3$
  - So, we say that this function's rate of growth is n^3
  - Since the function grows no faster than n^3, we say it is $O(n^3)$
- Can we say it is $O(n^4)$?

Asymptotic Notations - O

- O-notation characterizes an **upper bound** on the asymptotic behavior of a function.
  - A function grows no faster than a certain rate, based on the highest-order term.
- For example, the function $7n^3 + 100n^2 - 20n + 6$.
  - Its highest-order term is $7n^3$
  - So, we say that this function's rate of growth is n^3
  - Since the function grows no faster than n^3, we say it is $O(n^3)$
- Can we say it is $O(n^4)$?
  - Yes, because the function still grows no faster than n^4.
  - It is also $O(n^5), O(n^6)$, ...
  - Generally, it's $O(n^c)$ for $c \geq 3$

Asymptotic Notations - Ω

- Ω-notation characterizes a **lower bound** on the asymptotic behavior of a function.
  - A function grows at least as fast as a certain rate, based on the highest-order term.

- The function $7n^3 + 100n^2 - 20n + 6$.
  - It grows at least as fast as $\Omega(n^3)$.
  - It is also $\Omega(n^2), \Omega(n), \ldots$
  - Generally, it is $\Omega(n^c)$ for $c \leq 3$

Asymptotic Notations - Θ

- Θ-notation characterizes a **tight bound** on the asymptotic behavior of a function.
  - A function grows precisely at a certain rate, based on the highest-order term.
- It characterizes the rate of growth of the function to within a constant factor from above and to within a constant factor from below.
  - These two constant factors need not be equal.
- If you show that a function is $O\big(f(n)\big)$ and $\Omega\big(f(n)\big)$, then you have shown it is $\Theta\big(f(n)\big)$
- The function $7n^3 + 100n^2 - 20n + 6$ is $\Theta(n^3)$

Asymptotic Notations

- Analyzing insertion sort running time without evaluating summations.

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Asymptotic Notations

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

- The procedure has 2 loops:
  - The outer loop runs $n - 1$ times, regardless of the values being sorted.
  - The inner loop depends on the values being sorted.
  - The *while* loop variable j starts at $i - 1$ and decreases by 1 until it either reaches 0 or $A[j] > key$.
  - The while loop might iterate 0 times, $i \ - 1$ times, or anywhere in between.
  - So, the outer loop runs $n - 1$ times, the second loop runs at most $i - 1$times, and because i is at most n. Hence, it is $(n - 1)(n - 1) = \ O(n^2)$

Asymptotic Notations

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

- So, $O(n^2)$ describes an upper-bound for any case of insertion sort.

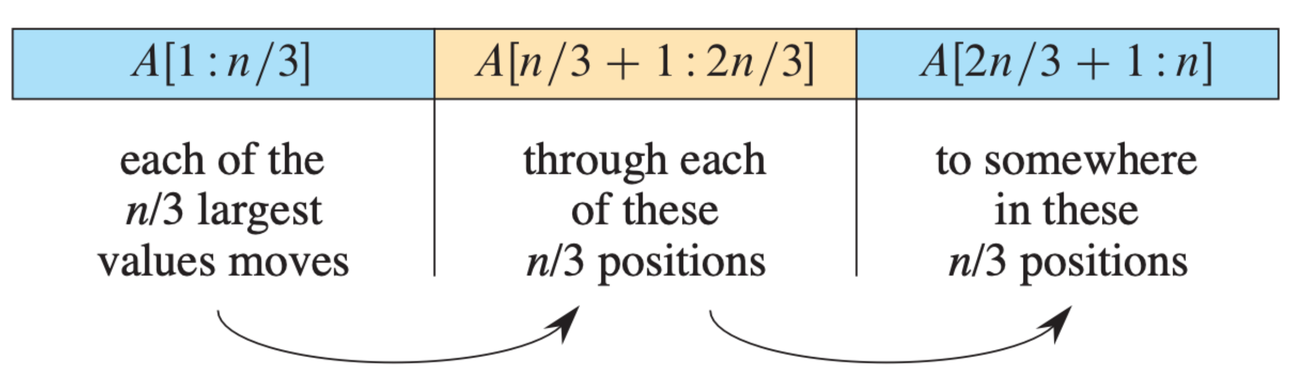
Asymptotic Notations

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

- The lower-bound of the worst-case running time is $\Omega(n^2)$.
- WHY?

Asymptotic Notations

- Assume that we have an array A of size n (multiple of 3), and the $n/3$ largest values are on the positions $A[1:n/3]$



- To sort the array, we will need to move each of the $A[1:n/3]$ to positions $A[2n/3+1:n]$.
  - Thus, the values will pass through the middle $A[n/3+1:2n/3]$ positions.
- Hence, the <u>worst-case lower-bound</u> is $(n/3)(n/3) = \Omega(n^2)$

Asymptotic Notations

- Because we have shown that insertion sort runs in $O(n^2)$ time in all cases.

- And that there is an input that makes it take $\Omega(n^2)$ time.

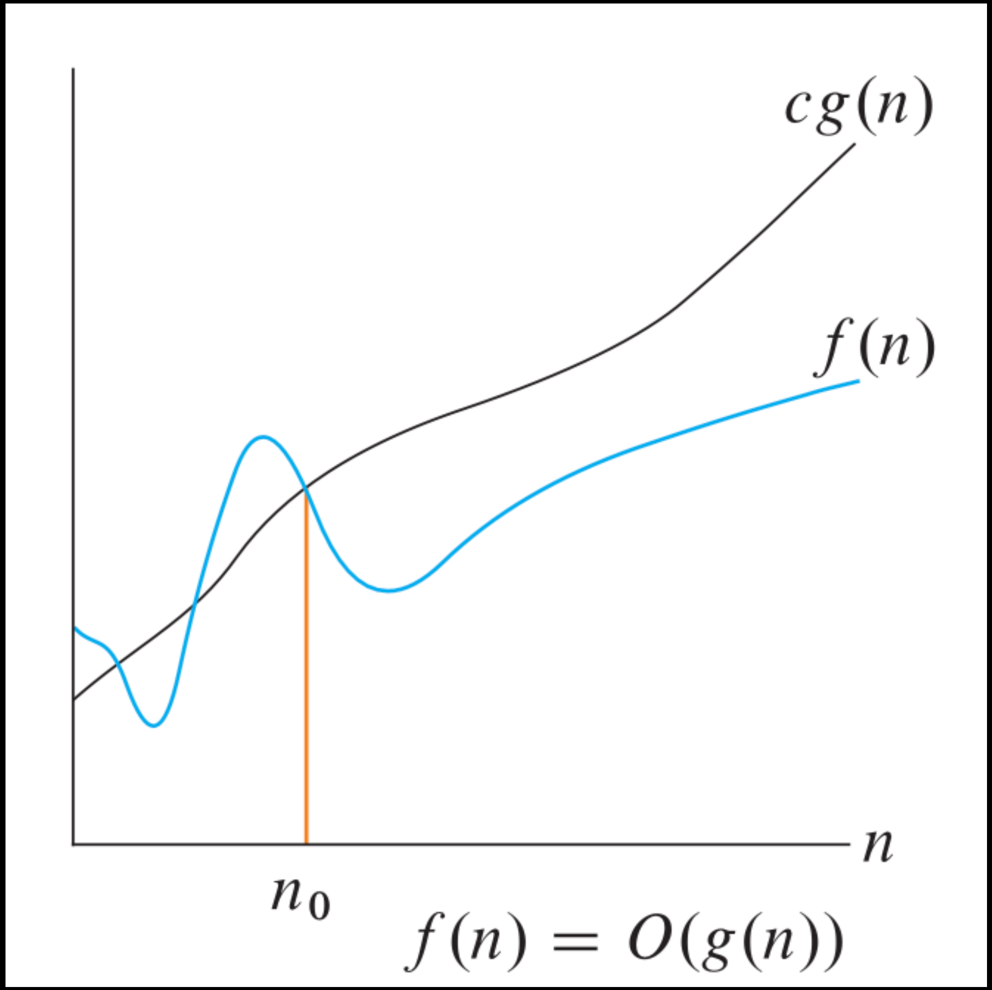
- We can conclude that the worst-case running time of insertion sort is $\Theta(n^2)$.

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω –Notations
Comparing Functions
Exercises

Asymptotic notation: Formal Definitions

- O-notation describes an asymptotic upper bound on a function, to within a constant factor.
- For a given function $g(n)$, we denote by $O\big(g(n)\big)$ the set of functions:

$$O\big(g(n)\big) = \{f(n){:}\ there\ exist\ positive\ constants\ c\ and\ n_0\ such\ that\ 0 \le f(n) \le cg(n)\ for\ all\ n \ge n_0\}$$



Asymptotic notation: Formal Definitions

- The definition of $O(g(n))$ requires that every function $f(n)$ in the set be asymptotically nonnegative
- Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $O(g(n))$is empty.
- Since $f(n)$ is a function that belongs to the set $O(g(n))$, we should have written it like $f(n) \in O(g(n))$
- But we will adopt the $f(n) = O(g(n))$ notation to indicate the same thing.

Asymptotic notation: Formal Definitions

- Example: show that $4n^2 + 100n + 500 = O(n^2)$.

- To do that, we have to find some positive c and n_0 to make
$$4n^2 + 100n + 500 \leq cn^2$$
- Divide both sides by n^2, we get $4 + 100/n + 500/n^2 \leq c$

- This equation can be satisfied for many choices:
  - $n_0 = 1$ and $c = 604$
  - $n_0 = 10$ and $c = 19$

Asymptotic notation: Formal Definitions

- Example: show that $n^3 - 100n^2$ does not belong to the set $O(n^2)$.

- We express it as $n^3 - 100n^2 \leq cn^2$.

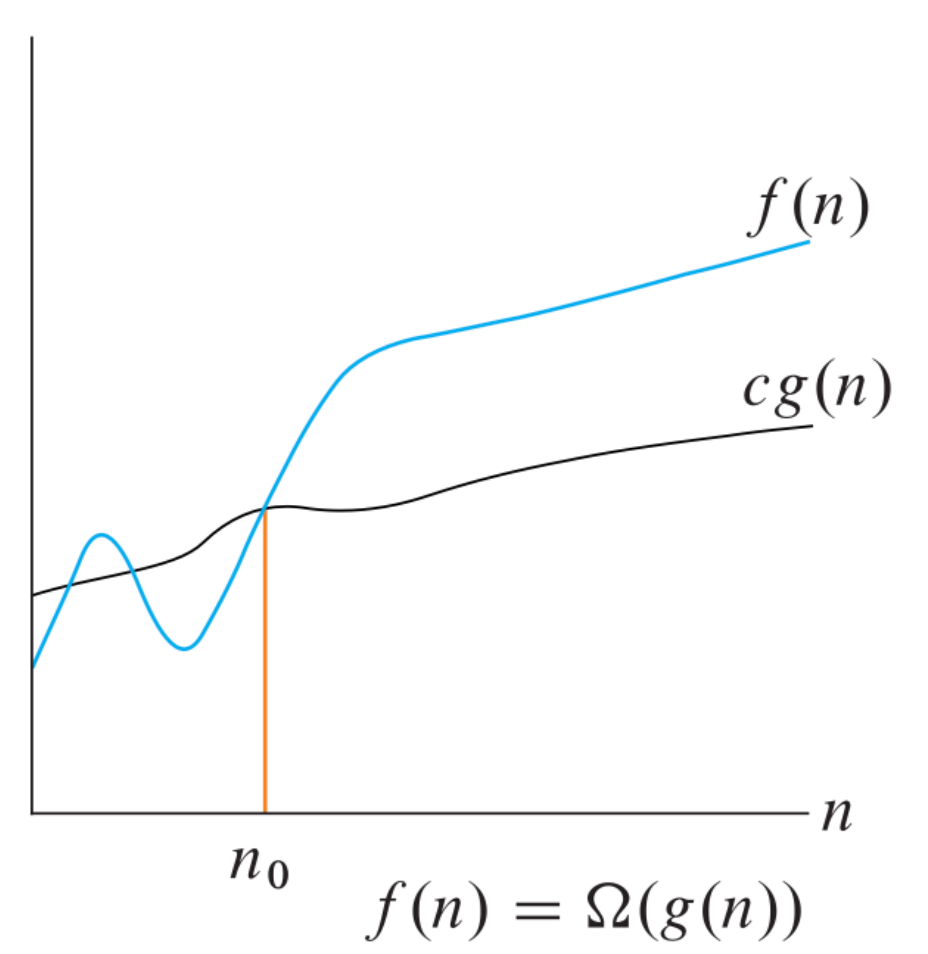
- Divide both sides by n^2, we get $n - 100 \leq c$.

- Regardless of what value we choose for the constant c, this inequality does not hold for any value of n.

Asymptotic notation: Formal Definitions

- Ω-notation provides an asymptotic lower bound.
- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions:

  $\Omega(g(n)) = \{f(n)\colon \textit{there exist positive constants } c \textit{ and } n_0 \textit{ such that } 0 \leq cg(n) \leq f(n) \textit{ for all } n \geq n_0\}$



Asymptotic notation: Formal Definitions

- Example: show that $4n^2 + 100n + 500 = \Omega(n^2)$

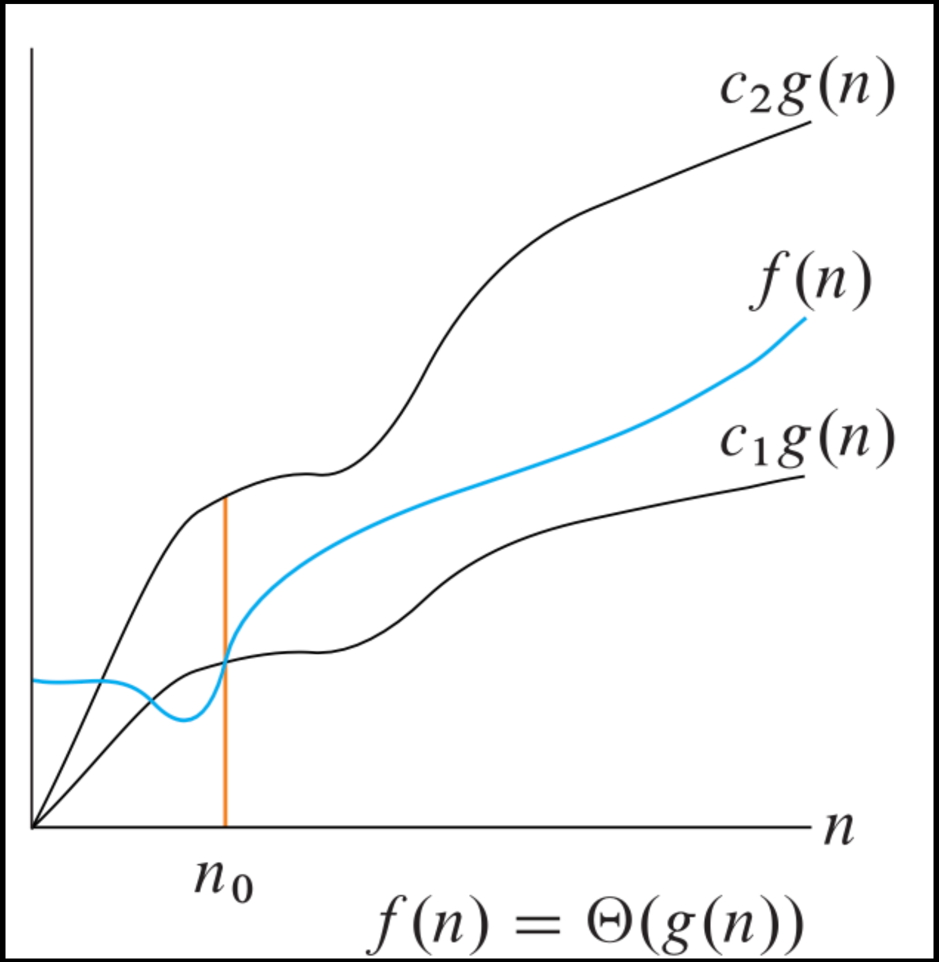
- We need to find positive constants c and n_0 such that $4n^2 + 100n + 500 \geq cn^2$ for all $n \geq n_0$
- Divide both sides by n^2, we get $4 + 100/n + 500/n^2 \geq c$.

- The inequality holds for any positive integer n_0 and $c = 4$.

Asymptotic notation: Formal Definitions

- Θ-notation provides an asymptotic tight bounds.
- For a given function $g(n)$, we denote by $\Theta\big(g(n)\big)$ the set of functions:

  $\Theta\big(g(n)\big) = \{f(n): \textit{there exist positive constants } c_1, c_2 \textit{ and } n_0 \textit{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \textit{ for all } n \ge n_0\}$



Asymptotic notation: Formal Definitions

- Based on the definition of $O-, \Omega-$, and $\Theta -$ notations, we get the following theorem:

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta\big(g(n)\big)$ if and only if $f(n) = O\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$

Asymptotic Notation and Running Times

- The asymptotic notation we use should be as precise as possible without overstating which running time it applies to.

Asymptotic Notation and Running Times

- Example: for the insertion sort algorithm,
  - We can correctly say that the worst-case running time is $O(n^2), \Omega(n^2)$, and $\Theta(n^2)$
    - The $\Theta(n^2)$ is the most precise one.
  - We can correctly say that the best-case running time is $O(n), \Omega(n)$, and $\Theta(n)$
    - The $\Theta(n)$ is the most precise one.
  - We cannot say that the running time is $\Theta(n^2)$
    - By omitting ”worst-case” we are referring to all the cases, which is not valid.
  - We can correctly say that the running time is $O(n^2)$
    - Because $O(n^2)$ is an upper bound; in all the cases the running time grows no faster than n^2.
  - Likewise, we cannot say that the running time $\Theta(n)$
    - But we can say that its running time is $\Omega(n)$

Asymptotic Notation and Running Times

- Example: for the merge sort algorithm, it runs in $\Theta(n \lg n)$ time in all cases.

- It is correct to say that its running time is $\Theta(n \lg n)$ without specifying ”worst-case”, ”best-case”, or any other case.

Asymptotic Notation and Running Times

- We typically use asymptotic notation to provide the simplest and most precise bounds possible.

- For example, if an algorithm has a running time of $3n^2 + 20n$ in all cases, we use asymptotic notation to write that its running time is $\Theta(n^2)$
- It is correct to say its running time is $O(n^3)$ or $\Theta(3n^2 + 20n)$.
  - Neither of these expressions is useful as $\Theta(n^2)$.
  - $O(n^3)$ is less precise.
  - $\Theta(3n^2 + 20n)$ introduces complexity that obscures the order of growth.

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω −Notations
Comparing Functions
Exercises

Asymptotic Notation in Equations and Inequalities

- Although we formally define asymptotic notation in terms of sets, we use the = instead of ∈ within formulas.

- For example, we wrote that $4n^2 + 100n + 500 = O(n^2)$.
- We might also write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$.

- How do we interpret such formulas?

Asymptotic Notation in Equations and Inequalities

- When the asymptotic notation stands alone on the right-hand side of an equation (or inequality), the equal sign means set membership:
  - $4n^2 + 100n + 500 = O(n^2). \rightarrow 4n^2 + 100n + 500 \in O(n^2)$.

- When asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name.
  - $2n^2 + 3n + 1 = 2n^2 + \Theta(n) \rightarrow 2n^2 + 3n + 1 = 2n^2 + \mathrm{f}(n)$ such that $f(n) \in \Theta(n)$
    - For instance, $f(n) = 3n + 1$, which belongs to $\Theta(n)$

Asymptotic Notation in Equations and Inequalities

- We can chain together a number of such relationships, as in
$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω —Notations
Comparing Functions
Exercises

o, ω –notations

- 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 tight.

- We use o-notation to denote an upper bound that is not asymptotically tight.

$$o\big(g(n)\big) = \{f(n)\colon there\ exist\ positve\ constants\ c\ and\ n_0\ such\ that\ 0 \le f(n) < cg(n)\ for\ all\ n \ge n_0\}$$

- For example, $2n = o(n^2)$ but $2n^2 \neq o(n^2)$

o, ω —notations

- Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n gets large:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

o, ω –notations

- The ω-notation denotes a lower bound that is not asymptotically tight.

$$\omega\big(g(n)\big) = \{f(n): there\ exist\ positive\ constants\ c\ and\ n_0$$
$$such\ that\ 0 \le cg(n) < f(n)\ for\ all\ n \ge n_0\ \}$$

- The relation $f(n) = \omega(n)$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

- Note that: $f(n) \in \omega\big(g(n)\big)$ if and only if $g(n) \in o\big(f(n)\big)$.
  - For example: $n^2/2 = \omega(n)$ but $n^2/2 \neq \omega(n^2)$

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω −Notations
Comparing Functions
Exercises

Comparing Functions

- Asymptotic notations has properties.
- Assume that $f(n)$ and $g(n)$ are asymptotically positive.

Transitivity:

$$
\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) , \\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) , \\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)), \\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) , \\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)) .
\end{aligned}
$$

Comparing Functions

- Asymptotic notations has properties.
- Assume that $f(n)$ and $g(n)$ are asymptotically positive.

Reflexivity:

$$f(n) = \Theta(f(n)) ,$$
$$f(n) = O(f(n)) ,$$
$$f(n) = \Omega(f(n)).$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)),$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .$$

Comparing Functions

- We can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$\begin{array}{lll} f(n) = O(g(n)) & \text{is like} & a \leq b\ , \\ f(n) = \Omega(g(n)) & \text{is like} & a \geq b\ , \\ f(n) = \Theta(g(n)) & \text{is like} & a = b\ , \\ f(n) = o(g(n)) & \text{is like} & a < b\ , \\ f(n) = \omega(g(n)) & \text{is like} & a > b\ . \end{array}$$

- $f(n)$ is asymptotically smaller than $g(n)$ → $f(n) = o\big(g(n)\big)$
- $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega\big(g(n)\big)$

Comparing Functions

- One property of real numbers, however, does not carry over to asymptotic notation:

  Trichotomy: For any two real numbers a and b, exactly one of the following must hold: $a < b, a = b, or\ a > b$.

- This means that in asymptotic notations, the $<$, $>$, and $=$ can hold at the same time.

Content
Introduction
Analyzing Insertion Sort
Worst-case and Average-case Analysis
Order of Growth
Asymptotic Notations
Asymptotic Notations: Formal Definitions
Asymptotic Notations and Running Times
Asymptotic Notations in Equations and Inequalities
o, ω −Notations
Comparing Functions
Exercises

Exercises

3.2-1

Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

Exercises

- **To prove that $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$, we start by recalling the definition of Θ notation**:
  A function $h(n)$ is in $\Theta(k(n))$ if there exist constants $c_1, c_2 > 0$ and n_0 such that $n \geq n_0$, $c_1 k(n) \leq h(n) \leq c_2 k(n)$
- **Establish an upper bound**:

Show that $\max\{f(n), g(n)\} \leq c_2(f(n) + g(n))$ for some constant c_2.

Thus, c_2 can be 1.

So, $\max\{f(n), g(n)\} \leq 1 \cdot (f(n) + g(n))$

Exercises

- **Establish a lower bound:**

Show that $\max\{f(n), g(n)\} \geq c_2\big(f(n) + g(n)\big)$

Thus, c_2 can be $1/2$

So, $\max\{f(n), g(n)\} \geq \frac{1}{2}\big(f(n) + g(n)\big)$

- **Combine the upper and lower bounds:**

$$½\big(f(n) + g(n)\big) \leq \max\{f(n), g(n)\} \leq 1 \cdot \big(f(n) + g(n)\big)$$

For sufficiently large n

- Thus, by the definition of Θ notation, we conclude that: $\max\{f(n)\,, g(n)\} = \Theta\big(f(n) + g(n)\big)$.

Exercises

3.2-2

Explain why the statement, “The running time of algorithm A is at least $O(n^2)$,” is meaningless.

Exercises

- The statement says "at least $O(n^2)$" implies that the running time can be bounded above by a quadratic function.
- However, Big O notation doesn't inherently provide a lower bound. It's about the upper limit on the growth of the function, not the minimum.
- The correct usage is to say either:
  - "The running time of algorithm A is $O(n^2)$ " to indicate that it does not grow faster than a quadratic function.
  - If you want to express a lower bound, you should use Ω notation, e.g., "The running time of algorithm A is $\Omega(n^2)$ " to indicate it grows <u>at least as</u> fast as a quadratic function.

Exercises

3.2-3

Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

Exercises

- $2^{n+1} = 2 \cdot 2^n$ for all $n \geq 0$. So, $2^{n+1} = O(2^n)$, where $c = 2$.

- $2^{2n} = 2^n \cdot 2^n \neq O(2^n)$. There is no n_0 such that $n \geq n\_0$ and there is no c.

Task

3.2-5
Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

3.2-6
Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

Task- answer

Suppose the running time is $\Theta(g(n))$. By Theorem 3.1, the running time is $O(g(n))$, which implies that for any input of size $n \geq n_0$ the running time is bounded above by $c_1 g(n)$ for some c_1. This includes the running time on the worst-case input. Theorem 3.1 also imlpies the running time is $\Omega(g(n))$, which implies that for any input of size $n \geq n_0$ the running time is bounded below by $c_2 g(n)$ for some c_2. This includes the running time of the best-case input.

On the other hand, the running time of any input is bounded above by the worst-case running time and bounded below by the best-case running time. If the worst-case and best-case running times are $O(g(n))$ and $\Omega(g(n))$ respectively, then the running time of any input of size n must be $O(g(n))$ and $\Omega(g(n))$. Theorem 3.1 implies that the running time is $\Theta(g(n))$.

Suppose we had some $f(n) \in o(g(n)) \cap \omega(g(n))$. Then, we have

$$0 = \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

a contradiction.