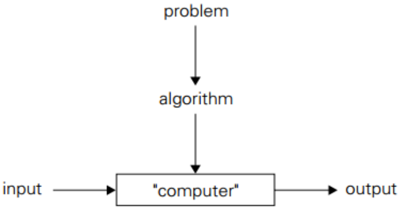
Data Structures and Algorithms

(ECEG 4171)

Chapter One
Algorithm Analysis

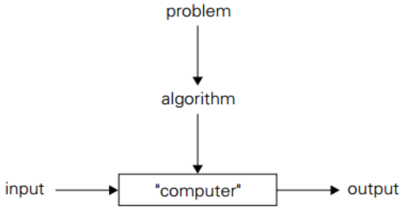
Introduction

- An **algorithm** is a well-defined computational procedure that takes some input and produces some output.
  - A tool for solving a wel-specified computational problem



Introduction

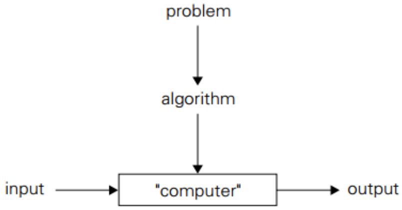
- An **algorithm** is a well-defined computational procedure that takes some input and produces some output.
  - A tool for solving a wel-specified computational problem



- For example: **the sorting problem**

Introduction

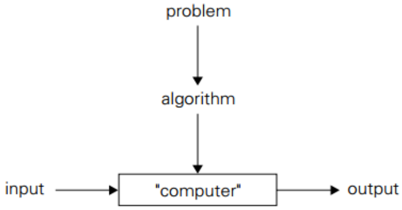
- An **algorithm** is a well-defined computational procedure that takes some input and produces some output.
  - A tool for solving a wel-specified computational problem



- For example: **the sorting problem**
  - **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Introduction

- An **algorithm** is a well-defined computational procedure that takes some input and produces some output.
  - A tool for solving a wel-specified computational problem



- For example: **the sorting problem**
  - **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output:** A permutation (reordering) $\langle a_1^{'}, a_2^{'}, \ldots, a_n^{'} \rangle$ of the input sequence such that $\langle a_1^{'} \leq a_2^{'} \leq \cdots \leq a_n^{'} \rangle$

Algorithms as a Technology

- If computers were infinitely fast and computer memory was free, would you have any reason to study algorithms?

Algorithms as a Technology

- If computers were infinitely fast and computer memory was free, would you have any reason to study algorithms?

The Answer is **YES**

Algorithms as a Technology

- If computers were infinitely fast and computer memory was free, would you have any reason to study algorithms?

The Answer is **YES**

- You would still like to demonstrate that your solution method terminates and does so with the correct answer.

Algorithms as a Technology

- If computers were infinitely fast and computer memory was free, would you have any reason to study algorithms?

The Answer is **YES**

- You would still like to demonstrate that your solution method terminates and does so with the correct answer.
- In reality, computers are not infinitely fast and memory is not free. Computing time and space in memory are bounded resources.
- Use algorithms that are efficient in terms of time and space.

Consider the following example

Computer A

- Implements **insertion sort** to sort n items.
- takes time $\approx c_1 n^2$
- executes 10 billion instructions per second
- insertion sort was written in machine language with code taking $2n^2$ instructions.
- for $n = 10$ million $\frac{2.(10^7)^2 instructions}{10^{10} instructions/second} = 20{,}000$ seonds (more than 5.5 hours).

Computer B

- Implements **merge sort** to sort n items.
- takes time $\approx c_2 nlgn$
- executes 10 million instructions per second
- merge sort was written in high-level language with an inefficient compiler with the code taking $50nlgn$ instructions.
- for $n = 10$ million $\frac{50.(10^7)lg10^7 instructions}{10^7 instructions/second} = 1163$ seonds (less than 20 minutes).

By using a faster algorithm, even with a poor compiler and slower execetion speed, computer B runs more than 17 times faster than computer A!

Algorithms and other technologies

- Example above shows we should consider algorithms, like computer hardware, as a technology.
- The importance of algorithms is comparable to other advanced techologies such as:
  - advanced computer architecture and fabrication technologies.
  - easy-to-use, intuitive, GUIs
  - object-oriented systems
  - integrated web technologies
  - fast networking, both wired and wireles

Pseudocodes

- In this chapter and the next, we use pseudocodes to represent algorithms
- Pseucodes are used to represent algorithms clearly and succinctly
- Ignore the details of a particular programming language.
  - Do not address error-handling and other software engineering issues.

Example:

Sample Java Code

```
void insertionSort(int[] A){
  int key, j, i;
  for(int j = 1; j < A.length; j++){
    key = A[j];
    //Insert A[j] into the sorted A[1..j-1]
    i = j - 1;
    while(i >= 0 && A[i] > key){
      A[i + 1] = A[i];
      i = i - 1;
    }
    A[i + 1] = key;
  }
}
```

Sample pseudocode

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

Pseudocode Conventions

- Indentation indicates block structures. **for** and **while** loops and **if-else** statements are block structures.
- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditionals have interpretations similar to those in C, C++, Java, Python, and Pascal.
- Use the keyword **to** when a for loop increments its loop counter in each iteration, and use keyword **downto** when a loop decrements its loop counter. When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.
- The symbol "//"indicates that the remainder of the line is a comment.
- Multiple assignment of the form $i = j = e$ is equivalent to the assignment $j = e$ followed by $i = j$.

Pseudocode Conventions

- Variables (such as i, j, and *key*) are local to the given procedure.
- Accessing array elements: $A[i]$ indicates the *ith* element and $A[1..j]$ indicates elements $A[1], A[2], \ldots, A[j]$
- A **return** statement immediately transfers control back to the point of call in the calling procedure. They also take a value to pass back to the caller. They also allow multiple values to be returned in a single **return** statement.
- The boolean operators **and** and **or** are **short circuiting**.
- The keyword **error** indicates that an error occured because conditions were wrong for the procedure to have been called.
- Array indexing always starts with **1**.

Analysis of Algorithms

- Predicting the resources that the algorithm requires.
  - ⇒ Memory
  - ⇒ Bandwidth
  - ⇒ Computer hardware
  - ⇒ Computational time
- We will usually use a generic uniprocessor random access machine (RAM) model.
  - All memory are equally expensive to access.
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, functions calls.
  - Constant word size.
    - Unless we are explicitly manipulating bits.

Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
- Time and space complexity are generally a function of input size.
  - Sorting: number of input items
  - Multiplication: total number of bits
  - Graph algorithms: number of nodes and edges
  - Etc

Types of Analysis

1. Best-case analysis
   - Provides a lower bound on running time
   - We must know the case that causes minimum number of operations to be executed.
2. Worst-case analysis
   - Provides an upper bound on running time
   - We must know the case that causes maximum number of operations to be executed.
   - An absolute guarantee
3. Average-case analysis
   - Provides the expected running time
   - We must know (or predict) the mathematical distribution of all possible inputs.
   - Very useful, but treat with care: what is "average"?
     - Random (equally likely) inputs
     - Real-life inputs

Analsis of insertion sort

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

An Example: Insertion Sort

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

30	10	40	20
1	2	3	4

$j = \emptyset$ $\quad i = \emptyset$ $\quad key = \emptyset$

$A[i] = \emptyset$ $\quad A[i+1] = \emptyset$

An Example: Insertion Sort

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

10	20	30	40
1	2	3	4

$j = 4$ $\quad i = 2$ $\quad key = 20$

$A[i] = 10$ $\quad A[i+1] = 20$

Done!

Proof of Correctness

- We will use a technique known as loop invariant
- Help us understand why an algorithm is correct. We must show three things about a loop invariant:
  - **Initialization**: The loop invariant is satisfied at the beginning of the for loop.
  - **Maintenance**: If the loop invariant is true before the ith iteration, then the loop invariant will be true before the `i + 1st` iteration.
  - **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Loop Invariant for Insertion Sort

- **Initialization**: Before the first iteration (which is when `j = 2`), the subarray `[1..j - 1]` is just the first element of the array, `A[1]`. This subarray is sorted, and consists of the elements that were originally in `A[1..1]`.
- **Maintenance**: Suppose `A[1..j - 1]` is sorted. Informally, the body of the for loop works by moving `A[j - 1], A[j - 2], A[j - 3]` and so on by one position to the right until it finds the proper position for `A[j]` (lines 4-7), at which point it inserts the value of `A[j]` (line 8). The subarray `A[1..j]` then consists of the elements originally in `A[1..j]`, but in sorted order. Incrementing `j` for the next iteration of the for loop then preserves the loop invariant.
- **Termination**: The condition causing the for loop to terminate is that `j > n`. Because each loop iteration increases `j` by 1, we must have `j = n + 1` at that time. By the initialization and maintenance steps, we have shown that the subarray `A[1..n + 1 - 1] = A[1..n]` consists of the elements originally in `A[1..n]`, but in sorted order.

Analyzing Insertion Sort

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

Analyzing Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	
$key = A[j]$		
//Insert $A[j]$ to the sorted $A[1..j-1]$		
$i = j - 1$		
while $i > 0$ and $A[i] > key$		
$A[i+1] = A[i]$		
$i = i - 1$		
$A[i+1] = key$		

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

cost	*times*
c_1	n
c_2	$n-1$

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ //Insert $A[j]$ to the sorted $A[1..j-1]$	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	
$\quad\quad A[i+1] = A[i]$		
$\quad\quad i = i - 1$		
$\quad A[i+1] = key$		

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Analyzing Insertion Sort

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

Running time = sum the prodcuts of the *cost* and *times* columns

Analyzing Insertion Sort

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

Running time = sum the prodcuts of the *cost* and *times* columns

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

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

- **Best-case analysis**

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

- **Best-case analysis**
  - Occurs if the array is aleardy sorted.

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

- **Best-case analysis**
  - ◇ Occurs if the array is aleardy sorted.
    - For each $j = 2, 3, \ldots, n$, $A[i] \leq key$ in line 5 $\Rightarrow t_j = 1$ for $j = 2, 3, \ldots, n$. Therefore,

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

- **Best-case analysis**
  - ◇ Occurs if the array is aleardy sorted.
    - For each $j = 2, 3, \ldots, n$, $A[i] \leq key$ in line 5 $\Rightarrow t_j = 1$ for $j = 2, 3, \ldots, n$. Therefore,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\ &= an + b \text{ (linear function)} \end{aligned}$$

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

- **Worst-case analysis**

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

- **Worst-case analysis**
  - ◇ Occurs if the array is reverse sorted.

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

- **Worst-case analysis**
  - ◇ Occurs if the array is reverse sorted.
    - – Must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1] \Rightarrow t_j = j$ for $j = 2, 3, \ldots, n$.

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

- **Worst-case analysis**
  - ◇ Occurs if the array is reverse sorted.
    - – Must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1] \Rightarrow t_j = j$ for $j = 2, 3, \ldots, n$.
      Note that,
      $$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$
      and
      $$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$
      Therefore,

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\Big(\frac{n(n+1)}{2} - 1\Big) \\
&\qquad + c_6\Big(\frac{n(n-1)}{2}\Big) + c_7\Big(\frac{n(n-1)}{2}\Big) + c_8(n-1) \\
&= \Big(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\Big)n^2 + \Big(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\Big)n \\
&\qquad - \big(c_2 + c_4 + c_5 + c_8\big) \\
&= an^2 + bn + c \text{ (Quadratic running time)}
\end{aligned}
$$

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

- **Average-case analysis**

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

- **Average-case analysis**
  - ◇ Roughly equivalent to randomly choosing n numbers and applying insertion sort.

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

- **Average-case analysis**
  - ◇ Roughly equivalent to randomly choosing n numbers and applying insertion sort.
    - – On average half elements are less than $A[j]$ and half elements are greater $\Rightarrow t_j \approx j/2$. By subsititution, the running time will be a quadratic function of the input size.

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

- **Average-case analysis**
  - ◇ Roughly equivalent to randomly choosing n numbers and applying insertion sort.
    - – On average half elements are less than $A[j]$ and half elements are greater $\Rightarrow t_j \approx j/2$. By subsititution, the running time will be a quadratic function of the input size.

$$= an^2 + bn + c$$

Asymptotic Performance

- How does an algorithm behave when the input size get very large?
- Asymptotic efficiency is studying the running time for large enough input sizes.
- This makes only the *the order of growth* of the running relevant.
- Asymtotic behavior (as n gets large) is determined entirely by the leading term.
- **Example**: $T(n) = 10n^3 + n^2 + 40n + 80$
  - If $n = 1000$, then $T(n) = 10,001,040,800$
  - error is 0.01% if we drop all but the n^3 term.

Asymptotic Performance

- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

- There are three commonly used asymptotic notations:
  - O – *notation*
  - Ω – *notation*
  - Θ – *notation*

Commonly encountered functions in the analysis of algorithms

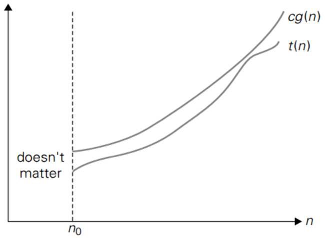
description	notation	definition
floor	$\lfloor x \rfloor$	largest integer not greater than x
ceiling	$\lceil x \rceil$	smallest integer not smaller than x
natural algorithm	lnN	$\log_e N$
binary algorithm	lgN	$\log_2 N$
logarithmic exponentiation	$lg^k n$	$(lgn)^k$
logarithmic composition	$lglgn$	$lg(lgn)$
harmonic numbers	H_N	$1 + 1/2 + 1/3 + \cdots + 1/N$
factorial	$N!$	$1X2X3X \ldots XN$

Big-O Notation

Definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \epsilon O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

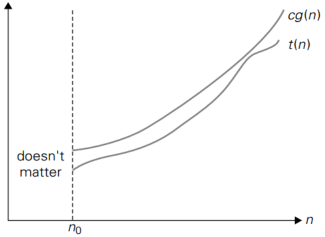


Big-O Notation

Definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \epsilon O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$



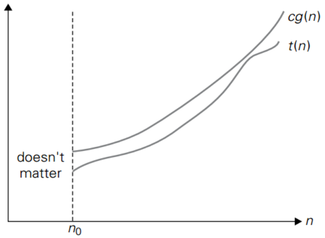
Example: Show that $100n + 5 \; \epsilon \; O(n^2)$

Big-O Notation

Definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \epsilon O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $100n + 5 \ \epsilon \ O(n^2)$

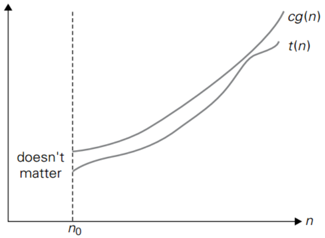
$100n + 5 \leq 100n + n$ for $n \geq 5$

Big-O Notation

Definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \epsilon O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $100n + 5 \ \epsilon \ O(n^2)$

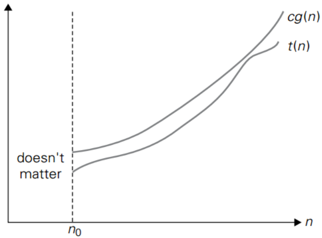
$$\begin{aligned} 100n + 5 &\leq 100n + n \text{ for } n \geq 5 \\ &\leq 101n \end{aligned}$$

Big-O Notation

Definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \epsilon O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $100n + 5 \ \epsilon \ O(n^2)$

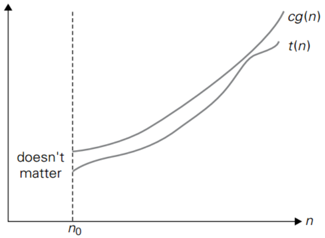
$$\begin{aligned} 100n + 5 &\leq 100n + n \text{ for } n \geq 5 \\ &\leq 101n \\ &\leq 101n^2 \end{aligned}$$

Big-O Notation

Definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \epsilon O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $100n + 5 \ \epsilon \ O(n^2)$

$$\begin{aligned} 100n + 5 &\leq 100n + n \text{ for } n \geq 5 \\ &\leq 101n \\ &\leq 101n^2 \end{aligned}$$

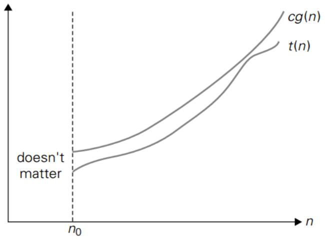
Therefore, for $c = 101$ and $n_0 = 5$
$100n + 5 = O(n^2)$

Big-O Notation

Definition

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \epsilon O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $100n + 5 \ \epsilon \ O(n^2)$

$$\begin{aligned} 100n + 5 &\leq 100n + n \text{ for } n \geq 5 \\ &\leq 101n \\ &\leq 101n^2 \end{aligned}$$

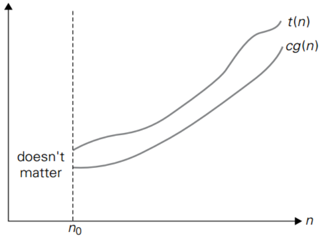
Therefore, for $c = 101$ and $n_0 = 5$
$100n + 5 = O(n^2)$

Ω-Notation

Definition

A function t(n) is said to be in $\Omega(g(n))$, denoted $t(n)\epsilon\Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$

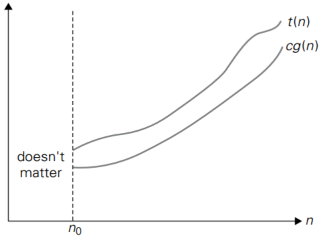


Ω-Notation

Definition

A function t(n) is said to be in $\Omega(g(n))$, denoted $t(n) \epsilon \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$



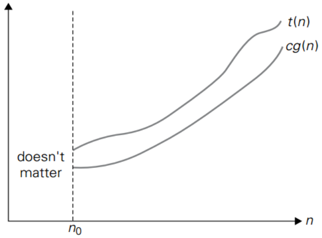
Example: Show that $n^3 \ \epsilon \ \Omega(n^2)$

Ω-Notation

Definition

A function t(n) is said to be in $\Omega(g(n))$, denoted $t(n) \epsilon \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $n^3 \ \epsilon \ \Omega(n^2)$

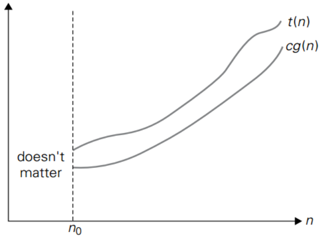
$n^3 \geq n^2$ for all $n \geq 0$

Ω-Notation

Definition

A function t(n) is said to be in $\Omega(g(n))$, denoted $t(n) \epsilon \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $n^3 \ \epsilon \ \Omega(n^2)$

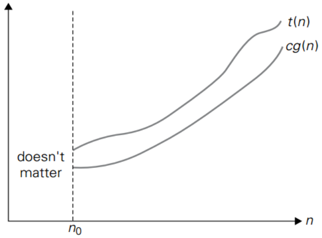
$n^3 \geq n^2$ for all $n \geq 0$
Therefore, for $c = 1$ and $n_0 = 0$,
$n^3 = \Omega(n^2)$

Ω-Notation

Definition

A function t(n) is said to be in $\Omega(g(n))$, denoted $t(n) \epsilon \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$



Example: Show that $n^3 \; \epsilon \; \Omega(n^2)$

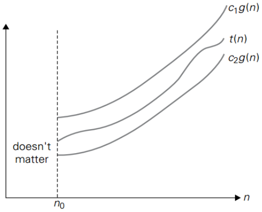
$n^3 \geq n^2$ for all $n \geq 0$
Therefore, for $c = 1$ and $n_0 = 0$,
$n^3 = \Omega(n^2)$

Θ-Notation

Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \epsilon \Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$



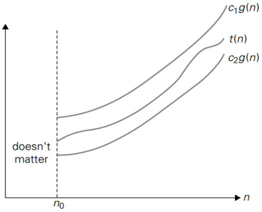
Θ-Notation

Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \epsilon \Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$

Example: Prove that $\frac{1}{2}n(n-1) \ \epsilon \ \Theta(n^2)$

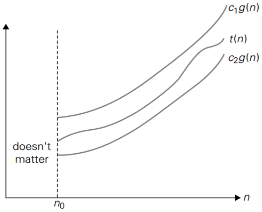


Θ-Notation

Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \epsilon \Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$



Example: Prove that $\frac{1}{2}n(n-1) \ \epsilon \ \Theta(n^2)$

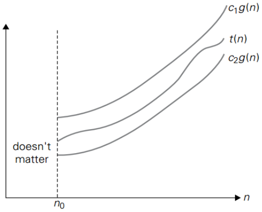
First, we prove the right inequality (the upper bound):

Θ-Notation

Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \epsilon \Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$



Example: Prove that $\frac{1}{2}n(n-1) \ \epsilon \ \Theta(n^2)$

First, we prove the right inequality (the upper bound):

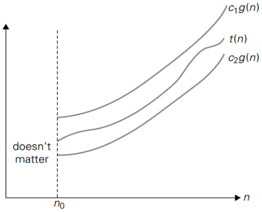
$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$ for all $n \geq 0$

Θ-Notation

Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \epsilon \Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$



Example: Prove that $\frac{1}{2}n(n-1) \ \epsilon \ \Theta(n^2)$

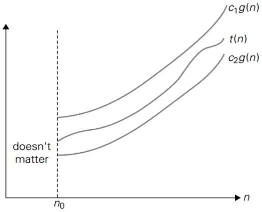
First, we prove the right inequality (the upper bound):
$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$ for all $n \geq 0$
Second, we prove the left inequality (the lower bound):

Θ-Notation

Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \epsilon \Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$



Example: Prove that $\frac{1}{2}n(n-1) \ \epsilon \ \Theta(n^2)$

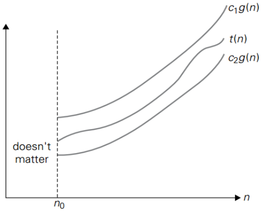
First, we prove the right inequality (the upper bound):
$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$ for all $n \geq 0$
Second, we prove the left inequality (the lower bound):
$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n$ for all $n \geq 2 = \frac{1}{4}n^2$

Θ-Notation

Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n)\epsilon\Theta(g(n))$, if $t(n)$ is bounded both above and below by some constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$



Example: Prove that $\frac{1}{2}n(n-1)\ \epsilon\ \Theta(n^2)$

First, we prove the right inequality (the upper bound):
$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$ for all $n \geq 0$
Second, we prove the left inequality (the lower bound):
$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n$ for all $n \geq 2 = \frac{1}{4}n^2$
Hence, $\frac{1}{2}n(n-1) = \Theta(n^2)$ for $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Useful Properties

1. If $t_1(n) = O(g_1(n))$ and $t_2(n) = O(g_2(n))$, then $t_1(n) + t_2(n) = O(max\{g_1(n), g_2(n)\})$.
2. For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
3. $$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that t(n) has a smaller order of growth than g(n),} \\ c & \text{implies that t(n) has the same order of growth as g(n),} \\ \infty & \text{implies that t(n) has a larger order of growth than g(n).} \end{cases}$$

   - The first two cases mean $t(n) = O(g(n))$, the last two mean that $t(n) = \Omega(g(n))$, and the second case means that $t(n) = \Theta(g(n))$

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

   $\lim_{n\to\infty} \frac{5n^3-n+2}{n^2} = \infty$

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

   $\lim_{n\to\infty} \frac{5n^3-n+2}{n^2} = \infty$

   $\Rightarrow 5n^3 - n + 2$ has larger order of growth than n^2.

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

   $\lim_{n\to\infty} \frac{5n^3-n+2}{n^2} = \infty$

   $\Rightarrow 5n^3 - n + 2$ has larger order of growth than n^2.

   $\Rightarrow 5n^3 - n + 2 = \Omega(n^2)$.

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

   $\lim_{n\to\infty} \frac{5n^3-n+2}{n^2} = \infty$

   $\Rightarrow 5n^3 - n + 2$ has larger order of growth than n^2.

   $\Rightarrow 5n^3 - n + 2 = \Omega(n^2)$.

2. Compare the orders of growth of lgn and $\sqrt{n}$.

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

   $\lim_{n\to\infty} \frac{5n^3-n+2}{n^2} = \infty$

   $\Rightarrow 5n^3 - n + 2$ has larger order of growth than n^2.

   $\Rightarrow 5n^3 - n + 2 = \Omega(n^2)$.

2. Compare the orders of growth of lgn and $\sqrt{n}$.

   $\lim_{n\to\infty} \frac{lgn}{\sqrt{n}} = \lim_{n\to\infty} \frac{(lgn)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)(lnn)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}$
   $= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$.

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

   $\lim_{n\to\infty} \frac{5n^3-n+2}{n^2} = \infty$

   $\Rightarrow 5n^3 - n + 2$ has larger order of growth than n^2.

   $\Rightarrow 5n^3 - n + 2 = \Omega(n^2)$.

2. Compare the orders of growth of lgn and $\sqrt{n}$.

   $\lim_{n\to\infty} \frac{lgn}{\sqrt{n}} = \lim_{n\to\infty} \frac{(lgn)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)(lnn)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}$
   $= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0.$

   Therefore, lgn has a smaller order of growth than $\sqrt{n}$.

More Examples

1. Compare the orders of growth of $5n^3 - n + 2$ and n^2.

   $\lim_{n\to\infty} \frac{5n^3-n+2}{n^2} = \infty$

   $\Rightarrow 5n^3 - n + 2$ has larger order of growth than n^2.

   $\Rightarrow 5n^3 - n + 2 = \Omega(n^2)$.

2. Compare the orders of growth of lgn and $\sqrt{n}$.

   $\lim_{n\to\infty} \frac{lgn}{\sqrt{n}} = \lim_{n\to\infty} \frac{(lgn)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)(lnn)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}$
   $= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0.$

   Therefore, lgn has a smaller order of growth than $\sqrt{n}$.

   $\Rightarrow lgn = O(\sqrt{n})$

More Examples

Find the time complexity of the following pseudocodes.

1. $sum = 0$
   for $i = 1$ to N **do**
       $sum = sum + i$

More Examples

Find the time complexity of the following pseudocodes.

1. $sum = 0$
   for $i = 1$ to N **do** $\Rightarrow O(N)$
   $\quad sum = sum + i$

More Examples

Find the time complexity of the following pseudocodes.

1. $sum = 0$
   for $i = 1$ to N **do**
   $\quad sum = sum + i$

   $\Rightarrow O(N)$

2. $sum = 0$
   for $i = 1$ to N **do**
   $\quad$**for** $j = 1$ to M **do**
   $\quad\quad sum = sum + (i + j)$

More Examples

Find the time complexity of the following pseudocodes.

1. $sum = 0$
   for $i = 1$ to N **do**
   $\quad sum = sum + i$

   $\Rightarrow O(N)$

2. $sum = 0$
   for $i = 1$ to N **do**
   $\quad$ **for** $j = 1$ to M **do**
   $\quad\quad sum = sum + (i + j)$

   $\Rightarrow T(n) = \sum_{i=1}^{N} \sum_{j=1}^{M} 1 = \sum_{i=1}^{N} M = M \sum_{i=1}^{N} 1 = NM = O(NM)$

More Examples

Find the worst-case, best-case, and average-case running for pseudocode given below:

ALGORITHM *SequentialSearch*$(A[1..n], K)$

//Input: An Array $A[1..n]$ and a search key K
//Output: The index of the first element in A that
//matches K or -1 if there are maching elements
$i = 1$
while $i \leq n$ and $A[i] \neq K$ **do**
    $i = i + 1$
if $i \leq n$ **then**
    return i
else
    return -1

ALGORITHM *SequentialSearch*$(A[1..n], K)$

$i = 1$
while $i \leq n$ and $A[i] \neq K$ **do**
    $i = i + 1$
if $i \leq n$ **then**
    return i
else
    return -1

- Worst-case running time:
  - Occurs when there are no matching elements or the mathing element happens to be the last one in the list.

    $\Rightarrow \quad T_{worst}(n) = n = O(n).$
- Best-case running time:
  - Occurs when the first element is equal to the search key.

    $\Rightarrow \quad T_{best}(n) = 1 = \Theta(1).$

ALGORITHM *SequentialSearch*$(A[1..n], K)$

$i = 1$
while $i \leq n$ and $A[i] \neq K$ **do**
    $i = i + 1$
if $i \leq n$ **then**
    return i
else
    return -1

- Average-case running time:
  - The probability of successful search is equal to p.
  - The probability of the first match occuring in the *ith*

    position is the same for every i.

    $\Rightarrow$ The probability of the first match occurring in the ith position of the list is p/n for every i.

    $\Rightarrow$ The number of comparisons made by the algorithm in such a situation is i.

    $\Rightarrow$ For an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1 - p)$. Therefore,

$$
\begin{aligned}
T_{avg}(n) &= [1.\frac{p}{n} + 2.\frac{p}{n} + \cdots + n.\frac{p}{n}] + n.(1 - p) \\
&= \frac{p}{n}[1 + 2 + \cdots + n] + n(1 - p) \\
&= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - P) \\
&= \frac{p(n+1)}{2} + n(1 - p) \\
&= \Theta(n)
\end{aligned}
$$

More Examples

Find the time complexity of the following algorithm.

//Input: An Array $A[1..n]$ of real numbers.
//Output: The value of the largest element in A

```
maxval = A[1]
for i = 2 to n do
    if A[i] > maxval then
        maxval = A[i]
return maxval
```

More Examples

//Input: An Array $A[1..n]$ of real numbers.
//Output: The value of the largest element in A

$maxval = A[1]$
for $i = 2$ to n **do**
    if $A[i] > maxval$ **then**
        $maxval = A[i]$
return maxval

- Here, the number of comparisons $A[i] > maxval$ will be the same for all arrays of size n; therefore, there is no need to distinguish between the worst, average, and best cases.

More Examples

//Input: An Array $A[1..n]$ of real numbers.
//Output: The value of the largest element in A

$maxval = A[1]$
for $i = 2$ to n **do**
    if $A[i] > maxval$ **then**
        $maxval = A[i]$
return maxval

- Here, the number of comparisons $A[i] > maxval$ will be the same for all arrays of size n; therefore, there is no need to distinguish between the worst, average, and best cases.
    - The algorithm makes one comparison on each execution of the loop:

More Examples

//Input: An Array $A[1..n]$ of real numbers.
//Output: The value of the largest element in A

$maxval = A[1]$
for $i = 2$ to n **do**
    if $A[i] > maxval$ **then**
        $maxval = A[i]$
return maxval

- Here, the number of comparisons $A[i] > maxval$ will be the same for all arrays of size n; therefore, there is no need to distinguish between the worst, average, and best cases.
  - The algorithm makes one comparison on each execution of the loop:

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

ALGORITHM *SequentialSearch*$(A[1..n], K)$

$i = 1$
while $i \leq n$ and $A[i] \neq K$ **do**
    $i = i + 1$
if $i \leq n$ **then**
    return i
else
    return -1

- Average-case running time:
  - The probability of successful search is equal to p.
  - The probability of the first match occuring in the *ith*

    position is the same for every i.
    - $\Rightarrow$ The probability of the first match occurring in the ith position of the list is p/n for every i.
    - $\Rightarrow$ The number of comparisons made by the algorithm in such a situation is i.
    - $\Rightarrow$ For an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1 - p)$. Therefore,

$$\begin{aligned}
T_{avg}(n) &= [1.\frac{p}{n} + 2.\frac{p}{n} + \cdots + n.\frac{p}{n}] + n.(1-p) \\
&= \frac{p}{n}[1 + 2 + \cdots + n] + n(1-p) \\
&= \frac{p}{n}\frac{n(n+1)}{2} + n(1-P) \\
&= \frac{p(n+1)}{2} + n(1-p) \\
&= \Theta(n)
\end{aligned}$$

More Examples

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer. Find its time comlexity.

//Input: A positive decimal integer n.
//Output: The number of binary digits in the n's binary representation

$count = 1$
while $n > 1$ **do**
  $count = count + 1$
  $n = \lfloor \frac{n}{2} \rfloor$
return count

More Examples

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer. Find its time comlexity.

//Input: A positive decimal integer n.
//Output: The number of binary digits in the n's binary representation
$count = 1$
while $n > 1$ **do**
    $count = count + 1$
    $n = \lfloor \frac{n}{2} \rfloor$
return count

- The most frequently executed operation here is the while loop. Since the value of n is about halved on each repetition of the loop, the answer should be about lgn. The exact number of times the comparison $n > 1$ is executed is $\lfloor lgn \rfloor + 1$

More Examples

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer. Find its time comlexity.

//Input: A positive decimal integer n.
//Output: The number of binary digits in the n's binary representation
$count = 1$
while $n > 1$ **do**
    $count = count + 1$
    $n = \lfloor \frac{n}{2} \rfloor$
return count

- The most frequently executed operation here is the while loop. Since the value of n is about halved on each repetition of the loop, the answer should be about lgn. The exact number of times the comparison $n > 1$ is executed is $\lfloor lgn \rfloor + 1$

$$T(n) \approx lgn$$
$$T(n) = O(lgn)$$