Data Structures and Algorithms [1]

BITS-Pilani K. K. Birla Goa Campus

[1]Material for the presentation taken from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Third Edition*;

Handout Discussion

My expectations from students

- Read the textbook:

  Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*

My expectations from students

- Read the textbook:

  Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*
- Solve the exercise problems.

My expectations from students

- Read the textbook:

  Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*
- Solve the exercise problems.
- Attend as many lectures as possible.

My expectations from students

- Read the textbook:

  Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*
- Solve the exercise problems.
- Attend as many lectures as possible.
- Attend all the labs.

My expectations from students

- Read the textbook:

  Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*
- Solve the exercise problems.
- Attend as many lectures as possible.
- Attend all the labs.
- Use PYQs only for solving additional problems.

Role of algorithms in computing

- What is an Algorithm?

Role of algorithms in computing

- What is an Algorithm?

  It is a well-defined computational procedure which takes an input and produces an output.

Role of algorithms in computing

- What is an Algorithm?

  It is a well-defined computational procedure which takes an input and produces an output.
- An algorithm for linear search.

Role of algorithms in computing

- What is an Algorithm?

  It is a well-defined computational procedure which takes an input and produces an output.
- An algorithm for linear search.

  Input : An array A of elements and a key k to be searched.

Role of algorithms in computing

- What is an Algorithm?

  It is a well-defined computational procedure which takes an input and produces an output.
- An algorithm for linear search.

  Input : An array A of elements and a key k to be searched.

  Output : The index of the key k in A, if found; otherwise, -1.

Role of algorithms in computing

- What is an Algorithm?

  It is a well-defined computational procedure which takes an input and produces an output.

- An algorithm for linear search.

  Input : An array A of elements and a key k to be searched.

  Output : The index of the key k in A, if found; otherwise, -1.

procedure $\textsc{LinearSearch}(A, k)$
    for $i \leftarrow 1$ to length of array A **do**
        if $A[i] = k$ **then**
            return i $\triangleright$ Key found at index i
    return -1 $\triangleright$ Key not found

Role of algorithms in computing

- An algorithm solves a computational problem.

Role of algorithms in computing

- An algorithm solves a computational problem.
- Sorting Problem

Role of algorithms in computing

- An algorithm solves a computational problem.
- Sorting Problem

  Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$

Role of algorithms in computing

- An algorithm solves a computational problem.
- Sorting Problem

  Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$
  Output: A permutation $< a'_1, a'_2, \ldots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Role of algorithms in computing

- An algorithm solves a computational problem.
- Sorting Problem

  Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$
  Output: A permutation $< a'_1, a'_2, \ldots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- E.g., Input sequence : $< 31, 41, 59, 26, 41, 58 >$

Role of algorithms in computing

- An algorithm solves a computational problem.
- Sorting Problem

  Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$
  Output: A permutation $< a'_1, a'_2, \ldots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- E.g., Input sequence : $< 31, 41, 59, 26, 41, 58 >$
- Sorting algorithm should give the output: $< 26, 31, 41, 41, 58, 59 >$

Role of algorithms in computing

- An algorithm is *correct* if it halts for every input with the correct output.

Role of algorithms in computing

- An algorithm is *correct* if it halts for every input with the correct output.
- All these computational problems require different algorithms.

Role of algorithms in computing

- An algorithm is *correct* if it halts for every input with the correct output.
- All these computational problems require different algorithms.

  e.g. Make cheapest air travel plan between two cities with at most k connecting flights,

Role of algorithms in computing

- An algorithm is *correct* if it halts for every input with the correct output.
- All these computational problems require different algorithms.

  e.g. Make cheapest air travel plan between two cities with at most k connecting flights,

  Solve Sudoku puzzle,

Role of algorithms in computing

- An algorithm is *correct* if it halts for every input with the correct output.
- All these computational problems require different algorithms.

  e.g. Make cheapest air travel plan between two cities with at most k connecting flights,

  Solve Sudoku puzzle,

  Compile C++ program into machine code etc.

Role of algorithms in computing

- An algorithm is *correct* if it halts for every input with the correct output.
- All these computational problems require different algorithms.

  e.g. Make cheapest air travel plan between two cities with at most k connecting flights,

  Solve Sudoku puzzle,

  Compile C++ program into machine code etc.
- Are we using any algorithm right now?

Role of algorithms in computing

- What is a Data Structure?

Role of algorithms in computing

- What is a Data Structure?

  A data structure is a way to store and organize data in order to facilitate access and modification.

Role of algorithms in computing

- What is a Data Structure?

  A data structure is a way to store and organize data in order to facilitate access and modification.

  Can we organize data such that minimum element, maximum element and any element k can be found efficiently even after insertion and deletion of elements?

Role of algorithms in computing

- What is a Data Structure?

  A data structure is a way to store and organize data in order to facilitate access and modification.

  Can we organize data such that minimum element, maximum element and any element k can be found efficiently even after insertion and deletion of elements? Binary Search Tree

Role of algorithms in computing

- What is a Data Structure?

  A data structure is a way to store and organize data in order to facilitate access and modification.

  Can we organize data such that minimum element, maximum element and any element k can be found efficiently even after insertion and deletion of elements? Binary Search Tree

  Can we organize data such that set membership and set union operations can be performed efficiently?

Role of algorithms in computing

- What is a Data Structure?

  A data structure is a way to store and organize data in order to facilitate access and modification.

  Can we organize data such that minimum element, maximum element and any element k can be found efficiently even after insertion and deletion of elements? Binary Search Tree

  Can we organize data such that set membership and set union operations can be performed efficiently? Disjoint-sets Forest data structure

Which Algorithm should be preferred?

- There can be more than one algorithm that correctly solves a given computational problem. Which one should we prefer?

Which Algorithm should be preferred?

- There can be more than one algorithm that correctly solves a given computational problem. Which one should we prefer?
- Computers have limited memory.

Which Algorithm should be preferred?

- There can be more than one algorithm that correctly solves a given computational problem. Which one should we prefer?
- Computers have limited memory.
- Only a finite number of instructions can be excecuted per second.

Which Algorithm should be preferred?

- There can be more than one algorithm that correctly solves a given computational problem. Which one should we prefer?
- Computers have limited memory.
- Only a finite number of instructions can be excecuted per second.
- We should prefer an algorithm that uses lesser memory and fewer instructions to solve a computational problem.

Which Algorithm should be preferred?

- There can be more than one algorithm that correctly solves a given computational problem. Which one should we prefer?
- Computers have limited memory.
- Only a finite number of instructions can be excecuted per second.
- We should prefer an algorithm that uses lesser memory and fewer instructions to solve a computational problem.
- A good algorithm would be *efficient* in terms of computing time and memory that is used.

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$ $\qquad$ $c_2 n \lg n = 50 \times 8 \times 3$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$ $\qquad$ $c_2 n \lg n = 50 \times 8 \times 3 = 1200$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$ $\qquad$ $c_2 n \lg n = 50 \times 8 \times 3 = 1200$
- Instructions that must be executed by each of the two algorithms for $n = 1$ million $(10^6 \approx 2^{20})$?

  $c_1 n^2$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$ $\quad$ $c_2 n \lg n = 50 \times 8 \times 3 = 1200$
- Instructions that must be executed by each of the two algorithms for $n = 1$ million $(10^6 \approx 2^{20})$?

  $c_1 n^2 \approx 2 \times 2^{20} \times 2^{20}$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$ $\qquad$ $c_2 n \lg n = 50 \times 8 \times 3 = 1200$
- Instructions that must be executed by each of the two algorithms for $n = 1$ million ($10^6 \approx 2^{20}$)?

  $c_1 n^2 \approx 2 \times 2^{20} \times 2^{20} = 2^{41}$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$ $\qquad$ $c_2 n \lg n = 50 \times 8 \times 3 = 1200$
- Instructions that must be executed by each of the two algorithms for $n = 1$ million $(10^6 \approx 2^{20})$?

  $c_1 n^2 \approx 2 \times 2^{20} \times 2^{20} = 2^{41}$ $\qquad$ $c_2 n \lg n \approx 50 \times 2^{20} \times 20$

Efficiency of Algorithms

- Two algorithms for the sorting problem: Insertion sort and Merge sort.
- Insertion sort takes roughly $c_1\, n^2$ machine-level instructions to solve a sorting problem.
- Merge sort takes roughly $c_2\, n\, lg\, n$ machine-level instructions.
- Let $c_1 = 2$, $c_2 = 50$. (c_1 is usually much smaller than c_2.)
- Instructions that must be executed by each of the two algorithms for $n = 8$?

  $c_1 n^2 = 2 \times 8 \times 8 = 128$ $\qquad$ $c_2 n \lg n = 50 \times 8 \times 3 = 1200$
- Instructions that must be executed by each of the two algorithms for $n = 1$ million ($10^6 \approx 2^{20}$)?

  $c_1 n^2 \approx 2 \times 2^{20} \times 2^{20} = 2^{41}$ $\qquad$ $c_2 n \lg n \approx 50 \times 2^{20} \times 20 \approx 2^{30}$

  Insertion sort requires 2^{11} (≈ 2000) times more machine-level instructions for solving the same problem!

Importance of a good algorithm

- Let us compare a faster computer A (running insertion sort) with a slower computer B (running merge sort).

Importance of a good algorithm

- Let us compare a faster computer A (running insertion sort) with a slower computer B (running merge sort).
- Suppose computer A executes 10 billion instructions per second and computer B executes 10 million instructions per second.

Importance of a good algorithm

- Let us compare a faster computer A (running insertion sort) with a slower computer B (running merge sort).
- Suppose computer A executes 10 billion instructions per second and computer B executes 10 million instructions per second.
- Let $n = 10$ million, $c_1 = 2$ and $c_2 = 50$.

Importance of a good algorithm

- Let us compare a faster computer A (running insertion sort) with a slower computer B (running merge sort).
- Suppose computer A executes 10 billion instructions per second and computer B executes 10 million instructions per second.
- Let $n = 10$ million, $c_1 = 2$ and $c_2 = 50$.
- Time taken by faster computer A: $\dfrac{2 \times (10^7)^2}{10^{10} \text{ instructions/second}}$

Importance of a good algorithm

- Let us compare a faster computer A (running insertion sort) with a slower computer B (running merge sort).
- Suppose computer A executes 10 billion instructions per second and computer B executes 10 million instructions per second.
- Let $n = 10$ million, $c_1 = 2$ and $c_2 = 50$.
- Time taken by faster computer A: $\dfrac{2 \times (10^7)^2}{10^{10} \text{ instructions/second}}$
- Time taken by slower computer B: $\dfrac{50 \times (10^7) \lg 10^7}{10^7 \text{ instructions/second}}$

Importance of a good algorithm

- Let us compare a faster computer A (running insertion sort) with a slower computer B (running merge sort).
- Suppose computer A executes 10 billion instructions per second and computer B executes 10 million instructions per second.
- Let $n = 10$ million, $c_1 = 2$ and $c_2 = 50$.
- Time taken by faster computer A: $\dfrac{2 \times (10^7)^2}{10^{10} \text{ instructions/second}}$
- Time taken by slower computer B: $\dfrac{50 \times (10^7) \lg 10^7}{10^7 \text{ instructions/second}}$
- Faster computer A takes more than 20,000 seconds (5.5 hours), whereas slower computer B takes around 1163 seconds ($<$ 20 minutes).

Importance of efficiency

- We should learn how to analyse and design efficient algorithms.

Comparison of running times

- For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

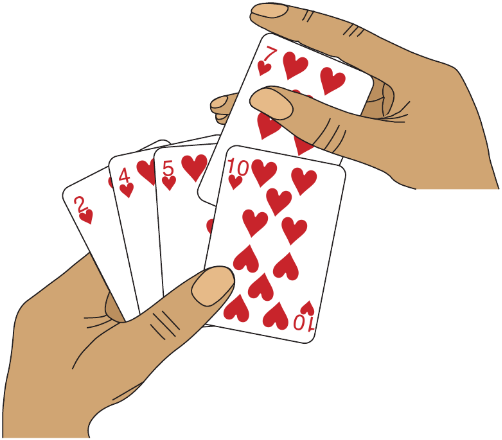
Running time of Insertion sort

Input: A sequence of n numbers $< a_1, a_2, \ldots, a_n >$
Output: A permutation $< a'_1, a'_2, \ldots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- E.g., Input sequence : $< 31, 41, 59, 26, 41, 58 >$
- Sorting algorithm should give the output: $< 26, 31, 41, 41, 58, 59 >$

Insertion sort

- Main idea: Works the way people sort a hand of playing cards.



Insertion sort

- `https://visualgo.net/en/sorting`

  5,2,4,6,1,3

Pseudocode for Insertion sort

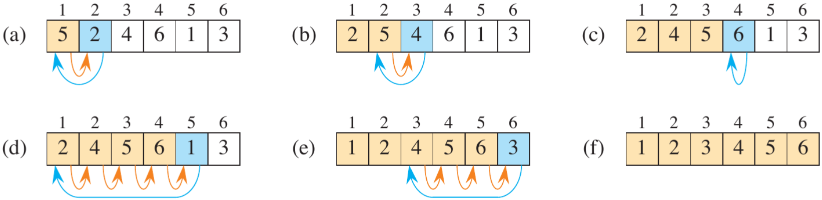
- Convention used : Array index starts from 1.

Pseudocode for Insertion sort

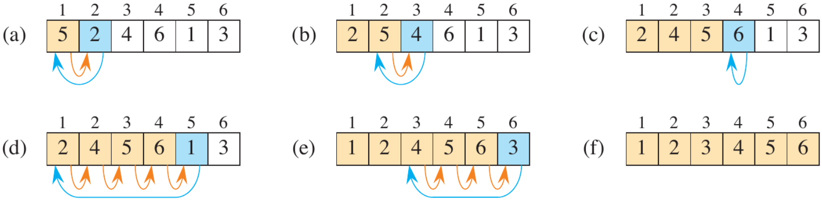
- Convention used : Array index starts from 1.

```
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
```

Outline for correctness of Insertion sort

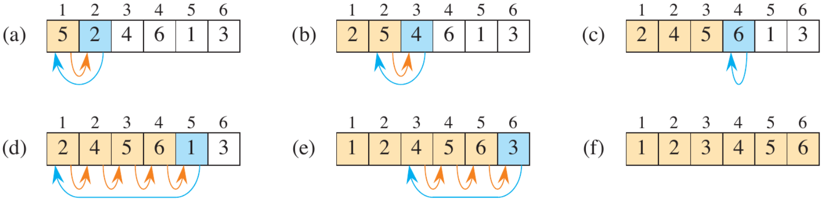


Outline for correctness of Insertion sort



- Loop invariant: At the start of each iteration of the **for** loop, the subarray $A[1..i-1]$ contains elements originally in $A[1..i-1]$, but in sorted order.

Outline for correctness of Insertion sort



- Loop invariant: At the start of each iteration of the **for** loop, the subarray $A[1..i-1]$ contains elements originally in $A[1..i-1]$, but in sorted order.
- We use loop invariant to argue for the correctness of an algorithm.

Outline for correctness of Insertion sort

Loop invariant: At the start of each iteration of the **for** loop, the subarray $A[1..i-1]$ contains elements originally in $A[1..i-1]$, but in sorted order.

```
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
```

- **Initialization,**

Outline for correctness of Insertion sort

Loop invariant: At the start of each iteration of the **for** loop, the subarray $A[1..i-1]$ contains elements originally in $A[1..i-1]$, but in sorted order.

```
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
```

- **Initialization, Maintenance,**

Outline for correctness of Insertion sort

Loop invariant: At the start of each iteration of the **for** loop, the subarray $A[1..i-1]$ contains elements originally in $A[1..i-1]$, but in sorted order.

```
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
```

- **Initialization, Maintenance, Termination**

Analysis of algorithms

- Running time of an algorithm is expressed as a function of the size of its input.

Analysis of algorithms

- Running time of an algorithm is expressed as a function of the size of its input.
- **Input size** : depends on the computational problem

Analysis of algorithms

- Running time of an algorithm is expressed as a function of the size of its input.
- **Input size** : depends on the computational problem
  - Sorting an array

Analysis of algorithms

- Running time of an algorithm is expressed as a function of the size of its input.
- **Input size** : depends on the computational problem
  - Sorting an array
  - Finding shortest path in a graph

Analysis of algorithms

- Running time of an algorithm is expressed as a function of the size of its input.
- **Input size** : depends on the computational problem
  - Sorting an array
  - Finding shortest path in a graph
  - Primality test

Analysis of algorithms

- Running time of an algorithm is expressed as a function of the size of its input.
- **Input size** : depends on the computational problem
  - Sorting an array
  - Finding shortest path in a graph
  - Primality test
- **Running time** : time needed for the primitive operations or "steps" executed by the Random-access machine (RAM) model for an input of size n.

Analysis of algorithms

- Random-access machine (RAM) is a generic single-processor model of computation having no concurrent operations.

Analysis of algorithms

- Random-access machine (RAM) is a generic single-processor model of computation having no concurrent operations.
- RAM model contains common operations/instructions (add, subtract, load, store, conditional branch, subroutine call etc.)

Analysis of algorithms

- Random-access machine (RAM) is a generic single-processor model of computation having no concurrent operations.
- RAM model contains common operations/instructions (add, subtract, load, store, conditional branch, subroutine call etc.)
- Each statement in the pseudocode takes a constant amount of time.

Pseudocode with time costs

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	$\sum_{i=2}^{n} t_i$
$\qquad A[j + 1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
$\qquad j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
$\quad A[j + 1] = key$	c_8	$n - 1$

Summation of time costs of all the steps

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	$n-1$

Summation of time costs of all the steps

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	$n-1$

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

Summation of time costs of all the steps

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

Summation of time costs of all the steps

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

- What will be the value of t_i if the input sequence is already sorted?

Summation of time costs of all the steps

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

- What will be the value of t_i if the input sequence is already sorted?

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

Summation of time costs of all the steps

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

- What will be the value of t_i if the input sequence is already sorted?

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

- The running time is of the form $an + b$, where a and b are constants.

Summation of time costs of all the steps

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

- What will be the value of t_i if the input sequence is already sorted?

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

- The running time is of the form $an + b$, where a and b are constants.
- So, the running time is a **linear function** of n if the input sequence is already sorted (best case scenario).

Summation of time costs in the worst case

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

- When will the worst case scenario occur?

Summation of time costs in the worst case

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

- When will the worst case scenario occur?
- What will be the value of t_i in the worst case scenario?

Summation of time costs in the worst case

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

- When will the worst case scenario occur?
- What will be the value of t_i in the worst case scenario?

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

Summation of time costs in the worst case

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

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

Summation of time costs in the worst case

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

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

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1)$$

Summation of time costs in the worst case

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

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

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

Summation of time costs in the worst case

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

Summation of time costs in the worst case

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

- Worst case running time is of the form $an^2 + bn + c$

Summation of time costs in the worst case

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

- Worst case running time is of the form $an^2 + bn + c$
- So, the running time is a **quadratic function** of n if the input sequence is sorted in the reverse order (worst case scenario).

Running time estimation

- What will the average case look like?

Running time estimation

- What will the average case look like?
  - $t_i = \dfrac{i}{2}$,

Running time estimation

- What will the average case look like?
  - $t_i = \dfrac{i}{2}$, $T(n)$ will again be a quadratic function of n.

Running time estimation

- What will the average case look like?
  - $t_i = \dfrac{i}{2}$, $T(n)$ will again be a quadratic function of n.
- Very often we will be interested in only the worst-case running time because it gives the upper bound on the running time for *any* input.

Order of growth: Simplifying abstractions

- We will make simplifying abstractions because we are only interested in the rate of growth (order of growth) of the running time for large values of n.

Order of growth: Simplifying abstractions

- We will make simplifying abstractions because we are only interested in the rate of growth (order of growth) of the running time for large values of n.
- Suppose $T(n) = 5n^2 + 100n + 500$

Order of growth: Simplifying abstractions

- We will make simplifying abstractions because we are only interested in the rate of growth (order of growth) of the running time for large values of n.
- Suppose $T(n) = 5n^2 + 100n + 500$
- After the simplifying abstractions, we are left with n^2 which is the factor by which $T(n)$ will increase for large values of n.

Order of growth: Simplifying abstractions

- We will make simplifying abstractions because we are only interested in the rate of growth (order of growth) of the running time for large values of n.
- Suppose $T(n) = 5n^2 + 100n + 500$
- After the simplifying abstractions, we are left with n^2 which is the factor by which $T(n)$ will increase for large values of n.
- We express the above by saying that insertion sort has a worst-case running time of $\Theta(n^2)$.

Order of growth: Simplifying abstractions

- We will make simplifying abstractions because we are only interested in the rate of growth (order of growth) of the running time for large values of n.
- Suppose $T(n) = 5n^2 + 100n + 500$
- After the simplifying abstractions, we are left with n^2 which is the factor by which $T(n)$ will increase for large values of n.
- We express the above by saying that insertion sort has a worst-case running time of $\Theta(n^2)$.
- Is order of growth of Insertion-sort dependent on how it is implemented?

Problem

2.2-1

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

Insertion sort

- Main idea : follow *Incremental* approach

Insertion sort

- Main idea : follow *Incremental* approach

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	$n-1$

The divide-and-conquer approach

- The **merge sort** algorithm follows the divide-and-conquer paradigm

The divide-and-conquer approach

- The **merge sort** algorithm follows the divide-and-conquer paradigm

  Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.

The divide-and-conquer approach

- The **merge sort** algorithm follows the divide-and-conquer paradigm

  Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.

  Conquer: Sort the two subsequences recursively using merge sort.

The divide-and-conquer approach

- The **merge sort** algorithm follows the divide-and-conquer paradigm

  Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.

  Conquer: Sort the two subsequences recursively using merge sort.

  Combine: Merge the two sorted subsequences to produce the sorted answer.

Combine step in Merge sort

- $MERGE(A, p, q, r)$ procedure.

Combine step in Merge sort

- $MERGE(A, p, q, r)$ procedure.
- The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted.

Combine step in Merge sort

- $MERGE(A, p, q, r)$ procedure.
- The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted.
- It combines the sorted subarrays such that the combined array $A[p..r]$ is also sorted.

Combine step in Merge sort

- $MERGE(A, p, q, r)$ procedure.
- The procedure assumes that the subarrays $A[p..q]$ and $A[q + 1..r]$ are sorted.
- It combines the sorted subarrays such that the combined array $A[p..r]$ is also sorted.
- Main idea :

Combine step in Merge sort

- $MERGE(A, p, q, r)$ procedure.
- The procedure assumes that the subarrays $A[p..q]$ and $A[q + 1..r]$ are sorted.
- It combines the sorted subarrays such that the combined array $A[p..r]$ is also sorted.
- Main idea :

L :	2	4	5	9
R :	3	6	7	8

Combine step in Merge sort

Combine step in Merge sort

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Merge procedure

- Running time of *MERGE* procedure is $\Theta(n)$

Merge sort

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textsc{Merge-Sort}(A, p, r)} \\
1 & \textbf{if } p < r \\
2 & \quad q = \lfloor (p + r)/2 \rfloor \\
3 & \quad \textsc{Merge-Sort}(A, p, q) \\
4 & \quad \textsc{Merge-Sort}(A, q + 1, r) \\
5 & \quad \textsc{Merge}(A, p, q, r)
\end{array}
$$

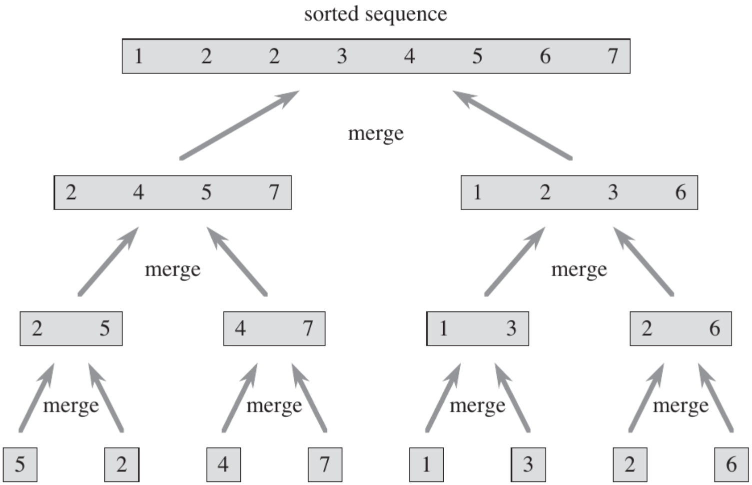
- Initial call $MERGE\text{-}SORT(A, 1, A.length)$

Merge sort operations

$A = [\ 5,\ 2,\ 4,\ 7,\ 1,\ 3,\ 6,\ 2\ ]$

Merge sort operations

A = [5, 2, 4, 7, 1, 3, 6, 2]



Analyzing divide-and-conquer algorithms

- Recurrence equation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analyzing the merge sort algorithm

- Recurrence for worst-case running time of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \, , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \, . \end{cases}$$

Analyzing the merge sort algorithm

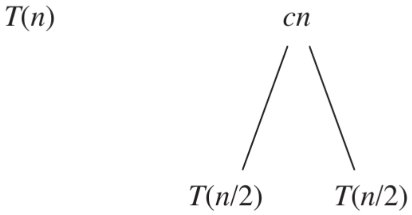
- Recurrence for worst-case running time of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

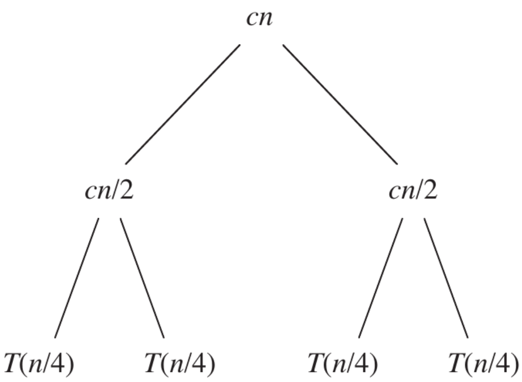
- Recurrence for worst-case running time of merge sort:

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

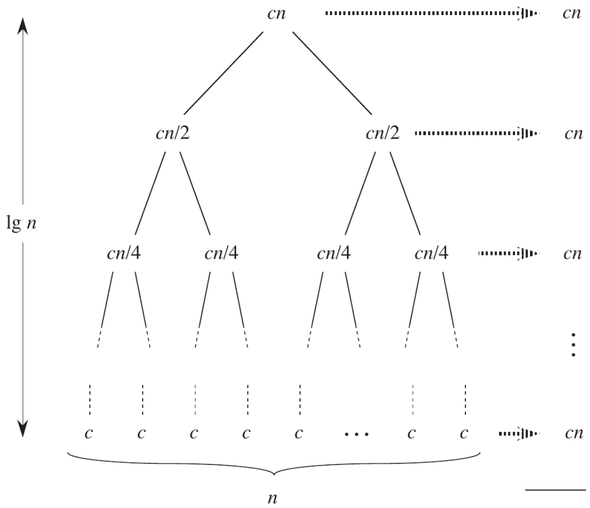
Recursion tree for merge sort



Recursion tree for merge sort



Recursion tree for merge sort



Analyzing the merge sort algorithm

- The recursion tree will have $\lg n + 1$ levels.

Analyzing the merge sort algorithm

- The recursion tree will have $\lg n + 1$ levels.
- Worst case running time of merge sort:

$$T(n) = cn \lg n + cn$$

Analyzing the merge sort algorithm

- The recursion tree will have $\lg n + 1$ levels.
- Worst case running time of merge sort:

$$\begin{aligned} T(n) &= cn \lg n + cn \\ &= \Theta(n \lg n) \end{aligned}$$

- Once input size n is large enough, merge sort ($\Theta(n \lg n)$) will beat insertion sort ($\Theta(n^2)$).

- Once input size n is large enough, merge sort ($\Theta(n \lg n)$) will beat insertion sort ($\Theta(n^2)$).
- We are interested in studying the **asymptotic** efficiency of algorithms (i.e. order of growth when n tends to infinity).

- Once input size n is large enough, merge sort ($\Theta(n \lg n)$) will beat insertion sort ($\Theta(n^2)$).
- We are interested in studying the **asymptotic** efficiency of algorithms (i.e. order of growth when n tends to infinity).
- We use different asymptotic notations (e.g. Θ notation) for describing the efficiency of algrithms.

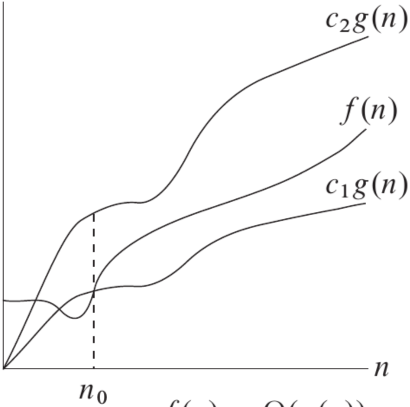
- Once input size n is large enough, merge sort ($\Theta(n \lg n)$) will beat insertion sort ($\Theta(n^2)$).
- We are interested in studying the **asymptotic** efficiency of algorithms (i.e. order of growth when n tends to infinity).
- We use different asymptotic notations (e.g. Θ notation) for describing the efficiency of algrithms.
- When we say running time $T(n) = \Theta(n^2)$, we mean $T(n)$ is a function in the set $\Theta(n^2)$.

Θ notation

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

Θ notation

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



$f(n) = \Theta(g(n))$

Θ notation

- Is $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$?

Θ notation

- Is $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$?
- To check the above we need to find positive constants c_1, c_2 and n_0 such that:

$$0 < c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \text{ , For } n \geq n_0 \tag{1}$$

Θ notation

- Is $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$?
- To check the above we need to find positive constants c_1, c_2 and n_0 such that:

$$0 < c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \text{ , For } n \geq n_0 \tag{1}$$

$$0 < c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Θ notation

- Is $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$?
- To check the above we need to find positive constants c_1, c_2 and n_0 such that:

$$0 < c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \text{ , For } n \geq n_0 \tag{1}$$

$$0 < c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- Eqn. 1 will be true for $c_1 = 1/14$, $c_2 = 1/2$ and $n_0 = 7$. (Other choices were also possible.)

Θ notation

- Is $3n^2 + 4n - 100 \in \Theta(n^2)$?

Θ notation

- Is $3n^2 + 4n - 100 \in \Theta(n^2)$?
- Is $3n + 2 \in \Theta(n^2)$?

Θ notation

- Is $3n^2 + 4n - 100 \in \Theta(n^2)$?
- Is $3n + 2 \in \Theta(n^2)$?
- Is $2n^3 \in \Theta(n^2)$?

Θ notation

- Is $3n^2 + 4n - 100 \in \Theta(n^2)$?
- Is $3n + 2 \in \Theta(n^2)$?
- Is $2n^3 \in \Theta(n^2)$?
- In general, if $p(n)$ is a degree d polynomial, then $p(n) \in \Theta(n^d)$.

Θ notation

- Is $3n^2 + 4n - 100 \in \Theta(n^2)$?
- Is $3n + 2 \in \Theta(n^2)$?
- Is $2n^3 \in \Theta(n^2)$?
- In general, if $p(n)$ is a degree d polynomial, then $p(n) \in \Theta(n^d)$.

Caveat: The coefficient of the highest degree term must be positive.

Θ notation

- $2n^2 + \Theta(n) = \Theta(n^2)$

O notation (Big-oh)

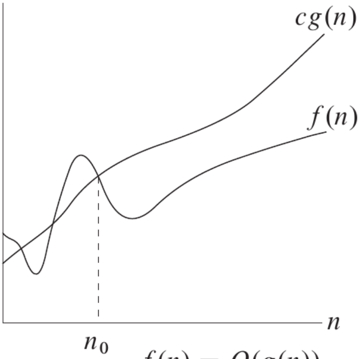
- We use O notation to express asymptotic upper bound.

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

O notation (Big-oh)

- We use O notation to express asymptotic upper bound.

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



$f(n) = O(g(n))$

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?
- Is $2n^3 \in O(n^2)$?

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?
- Is $2n^3 \in O(n^2)$?
- Is $2n + 2 \in O(n)$?

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?
- Is $2n^3 \in O(n^2)$?
- Is $2n + 2 \in O(n)$?
- Is the following statement true?
  If $f(n) = O(g(n))$ then $f(n) = \Theta(g(n))$.

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?
- Is $2n^3 \in O(n^2)$?
- Is $2n + 2 \in O(n)$?
- Is the following statement true?
  If $f(n) = O(g(n))$ then $f(n) = \Theta(g(n))$.
- O notation helps to express the running time on *every* input.

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?
- Is $2n^3 \in O(n^2)$?
- Is $2n + 2 \in O(n)$?
- Is the following statement true?
  If $f(n) = O(g(n))$ then $f(n) = \Theta(g(n))$.
- O notation helps to express the running time on *every* input.
- For example, the running time of insertion sort on *every* input is $O(n^2)$.

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?
- Is $2n^3 \in O(n^2)$?
- Is $2n + 2 \in O(n)$?
- Is the following statement true?
  If $f(n) = O(g(n))$ then $f(n) = \Theta(g(n))$.
- O notation helps to express the running time on *every* input.
- For example, the running time of insertion sort on *every* input is $O(n^2)$.
- Can we say, "running time of insertion sort on *every* input is $O(n^3)$"?

O notation (Big-oh)

- Is $3n + 2 \in O(n^2)$?
- Is $2n^3 \in O(n^2)$?
- Is $2n + 2 \in O(n)$?
- Is the following statement true?
  If $f(n) = O(g(n))$ then $f(n) = \Theta(g(n))$.
- O notation helps to express the running time on *every* input.
- For example, the running time of insertion sort on *every* input is $O(n^2)$.
- Can we say, "running time of insertion sort on *every* input is $O(n^3)$"?
- Is $3n^2 = O(n^2 - 10n - 20)$?

Ω-notation

- Provides an asymptotic lower bound.

Ω-notation

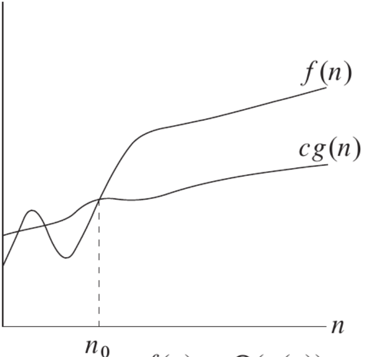
- Provides an asymptotic lower bound.

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

Ω-notation

- Provides an asymptotic lower bound.

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



$f(n) = \Omega(g(n))$

Ω-notation

- Is $3n^2 + 2 \in \Omega(n)$?

Ω-notation

- Is $3n^2 + 2 \in \Omega(n)$?
- Is $3n^2 + 2 \in \Omega(n^3)$?

Ω-notation

- It helps us express the best case running time for *any* input.

Ω-notation

- It helps us express the best case running time for *any* input.
- The running time of insertion sort is $\Omega(n)$ for *any* input.

Ω-notation

- It helps us express the best case running time for *any* input.
- The running time of insertion sort is $\Omega(n)$ for *any* input.

Theorem 3.1

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 $f(n) = \Omega(g(n))$. ■

o-notation

- $o(g(n))$ "little-oh of g of n"

o-notation

- $o(g(n))$ “little-oh of g of n”
- We use this notation to denote an upperbound which is not asymptotically tight

o-notation

- $o(g(n))$ "little-oh of g of n"
- We use this notation to denote an upperbound which is not asymptotically tight
- Tight bound:$2n^2 = O(n^2)$

o-notation

- $o(g(n))$ "little-oh of g of n"
- We use this notation to denote an upperbound which is not asymptotically tight
- Tight bound:$2n^2 = O(n^2)$ Loose bound:$2n = O(n^2)$.

o-notation

- $o(g(n))$ "little-oh of g of n"
- We use this notation to denote an upperbound which is not asymptotically tight
- Tight bound: $2n^2 = O(n^2)$ Loose bound: $2n = O(n^2)$.

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}.$$

o-notation

- $o(g(n))$ "little-oh of g of n"
- We use this notation to denote an upperbound which is not asymptotically tight
- Tight bound:$2n^2 = O(n^2)$ Loose bound:$2n = O(n^2)$.

$$o(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\} .$$

- Is $2n^2 + 1 = o(n^2)$?

o-notation

- $o(g(n))$ "little-oh of g of n"
- We use this notation to denote an upperbound which is not asymptotically tight
- Tight bound:$2n^2 = O(n^2)$ Loose bound:$2n = O(n^2)$.

$$o(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

- Is $2n^2 + 1 = o(n^2)$?
- Is $2n^2 + 1 = o(n^3)$?

ω-notation

- $\omega(g(n))$ "little-omega of g of n"

ω-notation

- $\omega(g(n))$ "little-omega of g of n"
- We use this notation to denote a lowerbound which is not asymptotically tight

ω-notation

- $\omega(g(n))$ "little-omega of g of n"
- We use this notation to denote a lowerbound which is not asymptotically tight
- Tight bound:$2n^2 = \Omega(n^2)$

ω-notation

- $\omega(g(n))$ "little-omega of g of n"
- We use this notation to denote a lowerbound which is not asymptotically tight
- Tight bound:$2n^2 = \Omega(n^2)$ Loose bound:$2n^2 = \Omega(n)$.

ω-notation

- $\omega(g(n))$ "little-omega of g of n"
- We use this notation to denote a lowerbound which is not asymptotically tight
- Tight bound: $2n^2 = \Omega(n^2)$ Loose bound: $2n^2 = \Omega(n)$.

$$\omega(g(n)) = \{ f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \} .$$

ω-notation

- $\omega(g(n))$ "little-omega of g of n"
- We use this notation to denote a lowerbound which is not asymptotically tight
- Tight bound:$2n^2 = \Omega(n^2)$ Loose bound:$2n^2 = \Omega(n)$.

$$\omega(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \} .$$

- Is $2n^2 + 1 = \omega(n^2)$?

ω-notation

- $\omega(g(n))$ "little-omega of g of n"
- We use this notation to denote a lowerbound which is not asymptotically tight
- Tight bound: $2n^2 = \Omega(n^2)$ Loose bound: $2n^2 = \Omega(n)$.

$$\omega(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \} .$$

- Is $2n^2 + 1 = \omega(n^2)$?
- Is $2n^2 + 1 = \omega(n)$?

Comparing Functions

- Is the following True?

  $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Common Mathematical Functions

- $a^{\log_c b} = b^{\log_c a}$, where $a > 0$ and $b > 0$

  Let $k = \log_b a$, then $a = b^k$

Common Mathematical Functions

- $a^{\log_c b} = b^{\log_c a}$, where $a > 0$ and $b > 0$
  Let $k = \log_b a$, then $a = b^k$
- Go through section 3.2 of the textbook.