In the name of God

Part 2. Complexity Theory

2.1. Complexity of Algorithms

Spring 2010

Instructor: Dr. Masoud Yaghini

Outline

- Algorithms
- Analyzing algorithms
- Order of Growth
- Complexity of Algorithms
- References

Algorithms

Algorithms

- **Algorithm**
  - An **algorithm** is any well-defined computational procedure that takes some values as **input** and produces some values as **output**.
- **Computational problem**
  - An algorithm is a tool for solving a well-specified **computational problem.**
- **Correct algorithm**
  - An algorithm is said to be **correct** if, for every input instance, it halts with the correct output.

Pseudocode

- **Pseudocode**
  - The algorithms are typically described as programs written in a **pseudocode** that is similar in many respects to C, Pascal, or Java.
- Difference between **pseudocode** and **real code**
  - **Pseudocode** employs an expressive method that is most clear and concise to specify a given algorithm
  - **Pseudocode** is not typically concerned with issues of software engineering, such as data abstraction, modularity, and error handling

Pseudocode

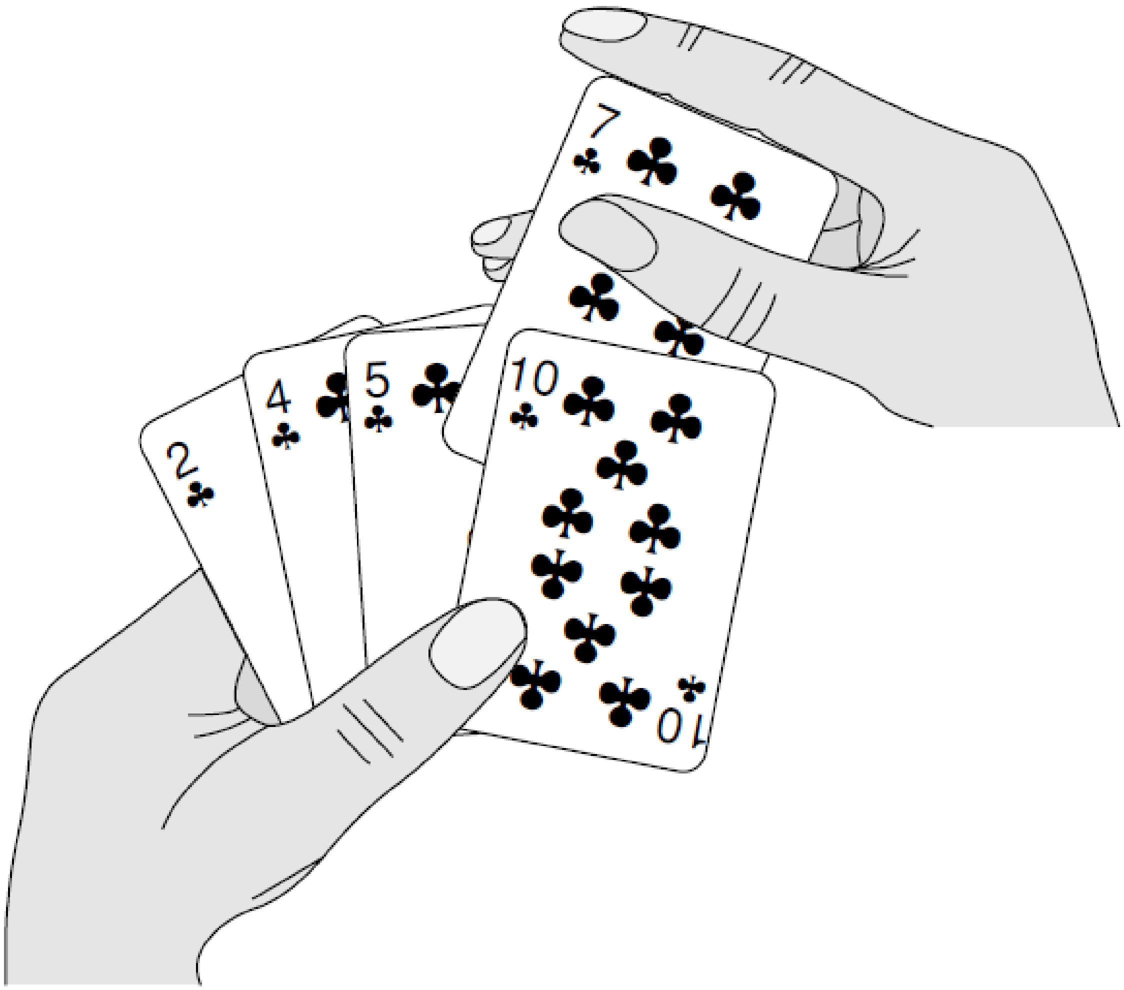
- Indentation indicates block structure.
- The looping constructs **while**, **for**, and **repeat** and
- The conditional constructs **if**, **then**, and **else**
- There is a symbol that indicates a comment.
- An assignment of the form $i \leftarrow e$ assigns variables i the value of expression e
- A multiple assignment of the form $i \leftarrow j \leftarrow e$ assigns to both variables i and j the value of expression e
- Array elements are accessed by specifying the array name followed by the index in square brackets

An Example: Insertion Sort

- Example: **sorting problem**
  - **Input**: A sequence of n numbers $\langle a_1, a_2, . . . , a_n \rangle$.
  - **Output**: A permutation (reordering) $\langle a'_1, a'_2, . . . , a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq . . . \leq a'_n \rangle$
- **Insertion sort algorithm**
  - **Insertion sort** is an efficient algorithm for sorting a small number of elements.
  - The numbers that we wish to sort are also known as the **keys**.
  - Insertion sort works the way many people sort a hand of playing cards.

An Example: Insertion Sort

- Sorting a hand of cards using insertion sort



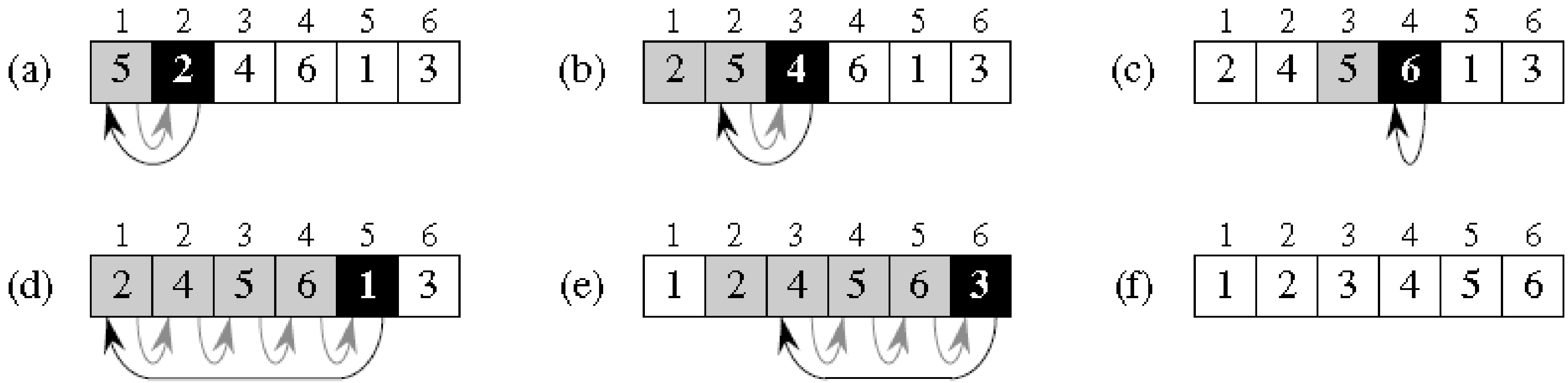
An Example: Insertion Sort

- **Input**: an array $A[1\ ..\ n]$ containing a sequence of length n that is to be sorted.

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

An Example: Insertion Sort

- The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$.



An Example: Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = ∅ j = ∅ key = ∅

A[j] = ∅ A[j+1] = ∅

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = 2 j = 1 key = 10
A[j] = 30 A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

1	2	3	4
30	30	40	20

i = 2 j = 1 key = 10
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
       }
       A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 1   key = 10
A[j] = 30   A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2 j = 0 key = 10
A[j] = ∅ A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 0   key = 10

A[j] = ∅   A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
  for i = 2 to n {
       key = A[i]
       j = i - 1;
       while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
       }
⇒      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 0 key = 10

A[j] = ∅ A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 0 key = 40
A[j] = ∅ A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3   j = 0   key = 40
A[j] = ∅   A[j+1] = 10

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3   j = 2   key = 40
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3   j = 2   key = 40
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

1	2	3	4
10	30	40	20

i = 4   j = 2   key = 20
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

1	2	3	4
10	30	40	20

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4   j = 3   key = 20
A[j] = 40   A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

1	2	3	4
10	30	40	40

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4   j = 2   key = 20
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
  }
}
```

An Example: Insertion Sort

1	2	3	4
10	30	30	40

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4	j = 1	key = 20
A[j] = 10		A[j+1] = 20

```
InsertionSort(A, n) {
  for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
  }
}
```

Done!

Analyzing algorithms

Analyzing algorithms

- **Analyzing an algorithm**
  - Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
  - Main resources are **computational time** and **memory**
  - Most often it is computational time that we want to measure.
  - By analyzing several candidate algorithms for a problem, a most efficient one can be easily identified.

Analyzing algorithms

- The time taken by the INSERTION-SORT procedure depends on
  - The size of the input: sorting a thousand numbers takes longer than sorting three numbers.
  - How the numbers nearly sorted they already are.
- The time taken by an algorithm grows with **the size of the input**
- It is traditional to describe the **running time of a program** as a function of **the size of its input**.

Analyzing algorithms

- **Input size**
  - For many problems, the most natural measure is the number of items in the input—for example, the array size n for sorting.
  - Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one.
    - For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph.

Analyzing algorithms

- **Running time**
  - The **running time** of an algorithm on a particular input is the number of primitive operations or “steps” executed.
  - It is machine-independent.

Analyzing Insertion Sort

- We start by presenting the time cost of each statement and the number of times each statement is executed.

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

Analyzing Insertion Sort

- Where,
  - c_i : the time cost of *i*th the statement
  - $j = 2, 3, \ldots, n$, where $n = length[A]$
  - t_j : the number of times the **while** loop test in line 5 is executed for that value of j.
  - $T(n)$: the running time of algorithm

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

- Even for inputs of a given size, an algorithm's running time may depend on **which** input of that size is given.

Analyzing Insertion Sort

- **Best case**
  - The **best case** occurs if the array is already sorted,
  - $t_j = 1$ for $j = 2, 3, \ldots, n$, inner loop body never executed
  - The best-case running time is:

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

  - *T(n)* can be expressed as ***an* + *b*** for constants *a* and *b* that depend on the statement costs c_i
  - It is thus a ***linear function*** of *n.*

Analyzing Insertion Sort

- **Worst case**
  - $t_j = j$ for $j = 2, 3, \ldots, n$, inner loop body executed for all previous elements

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

  - $T(n)$ can be expressed as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i ;
  - It is thus a **quadratic function** of n.

Analyzing algorithms

- In analyzing algorithm, we usually concentrate on finding only the **worst-case running time**, that is the longest running time for **any** input of size *n*.
- The reasons for using **worst-case running time**:
  - The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer.
  - For some algorithms, the worst case occurs frequently.

Order of Growth

Order of Growth

- **Asymptotic performance**
  - How does algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
- **Order of growth / rate of growth**
  - is the interesting measure

O-Notation

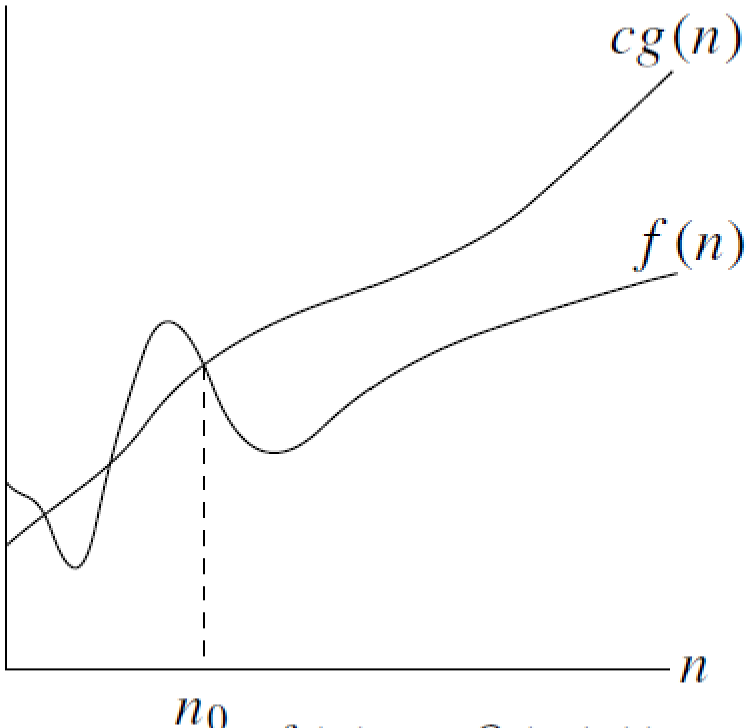
- ***O*-notation** provides an **asymptotic upper bound.**
- When we use *O*-notation to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input.
- **Definition of Big-*O* Notation**
  - $O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ $\}$
- We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$.

O-Notation

- The worst-case running time of INSERTION-SORT is $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i
- For simplicity we ignore constant factors because they are less significant than the rate of growth in determining computational efficiency for large inputs
- Therefore we consider only $\boldsymbol{n^2}$ as **rate of growth** or **order of growth**
- Thus, we say Insertion-Sort's (worst-case) running time is $\boldsymbol{O(n^2)}$
  - Properly we should say run time is ***in*** $\boldsymbol{O(n^2)}$

O-Notation

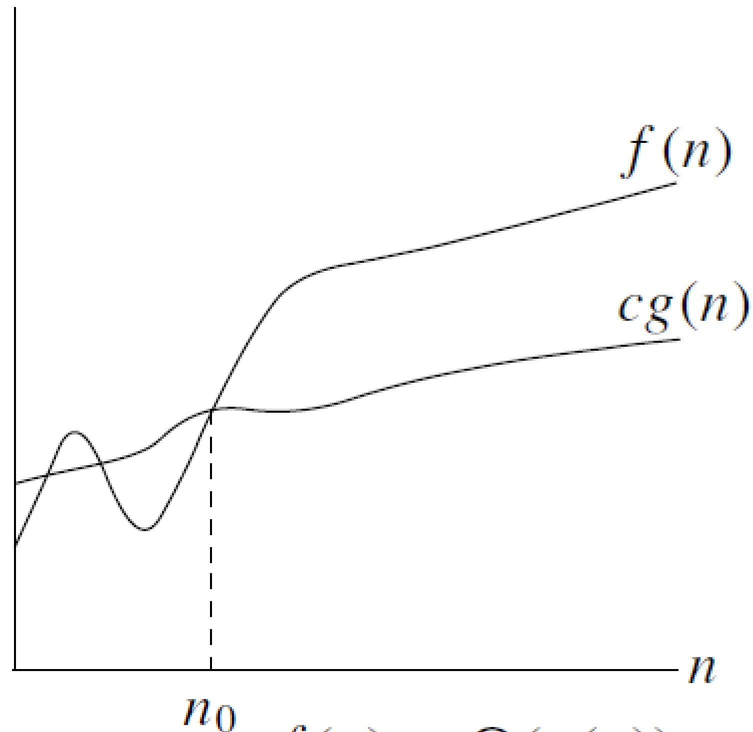
- The value of $f(n)$ always lies on or below $cg(n)$.



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

Ω-Notation

- **Ω-notation** provides an ***asymptotic lower bound.***
- **Definition of Big-Ω Notation**
  - $\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n)$ for all $n \ge n_0 \}$

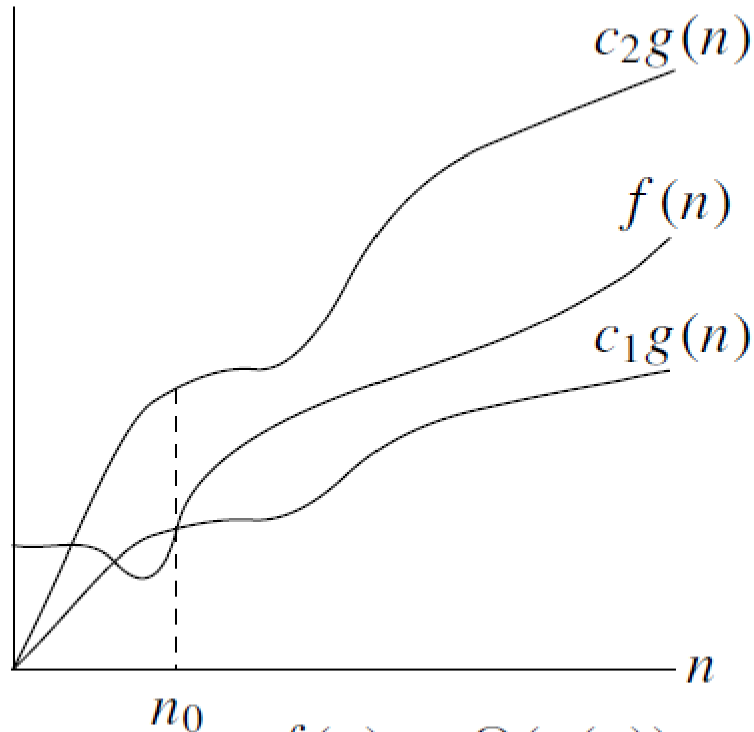


$$f(n) = \Omega(g(n))$$

Θ-Notation

- **Definition of Big-Θ Notation**
  - $\Theta(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0 \}$
- function $f(n)$ belongs to the set $(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$, for sufficiently large n.
- Because $(g(n))$ is a set, we could write $f(n) \in (g(n))$ to indicate that $f(n)$ is a member of $(g(n))$.
- Instead, we will usually write $f(n) = (g(n))$ to express the same notion.

Θ-Notation



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

Complexity of Algorithms

Complexity of Algorithms

- **Polynomial-time algorithm**
  - An algorithm is a polynomial-time algorithm if its complexity is $O(g(n))$, where $g(n)$ is a polynomial function of n.
- A polynomial function of degree k can be defined as follows:

$$p(n) = a_k \cdot n^k + \cdots + a_j \cdot n^j + \cdots + a_1 \cdot n + a_0$$

- where $a_k > 0$ and $a_j \geq 0, \forall 1 \leq j \leq k - 1$.
- The corresponding algorithm has a polynomial complexity of $O(n^k)$.

Complexity of Algorithms

- **Exponential-time algorithm**
  - An algorithm is an exponential-time algorithm if its complexity is $O(c^n)$, where c is a real constant strictly superior to 1.

Complexity of Algorithms

- Search time of an algorithm as a function of the problem size using different complexities

Complexity	Size = 10	Size = 20	Size = 30	Size = 40	Size = 50
$O(x)$	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s
$O(x^2)$	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s
$O(x^5)$	0.1 s	0.32 s	24.3 s	1.7 mn	5.2 mn
$O(2^x)$	0.001 s	1.0 s	17.9 mn	12.7 days	35.7 years
$O(3^x)$	0.059 s	58.0 mn	6.5 years	3855 centuries	2×10^8 centuries

References

References

- Thomas H. Cormen et al., **Introduction to Algorithms**, Second Edition, The MIT Press, 2001. (Chapter 1-3)
- El-Ghazali Talbi, **Metaheuristics : From Design to Implementation**, John Wiley & Sons, 2009. (Chapter 1)

The End