# Algorithms & Data Structures

## Lecture 02
## Insertion sort & Asymptotic notation

Giovanni Bacci
giovbacci@cs.aau.dk

# Outline

- Insertion sort
- Loop invariants for proving correctness
- Asymptotic notation & Asymptotic analysis

# Intended Learning Goals

**KNOWLEDGE**

- Mathematical reasoning on concepts such as recursion, induction, concrete and abstract computational complexity
- Data structures, algorithm principles e.g., search trees, hash tables, dynamic programming, divide-and-conquer
- Graphs and graph algorithms e.g., graph exploration, shortest path, strongly connected components.

**SKILLS**

- Determine abstract complexity for specific algorithms
- Perform complexity and correctness analysis for simple algorithms
- Select and apply appropriate algorithms for standard tasks

**COMPETENCES**

- Ability to face a non-standard programming assignment
- Develop algorithms and data structures for solving specific tasks
- Analyse developed algorithms

# Sorting Problem

**The sorting problem:**

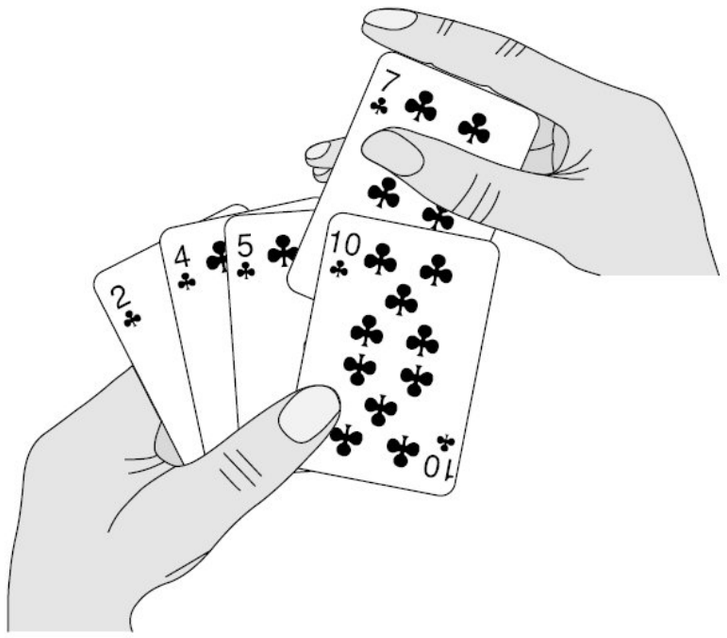
**Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

**Output:** A permutation (reordering) $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ of the input such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

- The sorting problem is present as sub-problem for many real-life problems — having a sorted sequences greatly simply many other algorithms
- There are many famous sorting algorithms

# Insertion sort

- Efficient algorithm for sorting a small number of elements
- Works the way many people sort a hand of playing cards
  - We start with an empty left hand and the cards face down on the table
  - Then we remove one card at a time from the table and insert it into the correct position in the left hand
  - To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left



# Insertion sort

The procedure takes an array $A[1..n]$ and sorts its content

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

- $A[1..j-1]$ are the cards in the hand
- $A[j..n]$ are the cards in the table

The algorithm sorts $A[1..n]$ **in place**: it rearranges $A$'s content by storing at most a constant number of elements outside the array at any time.

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

The loop does 3 tasks:

INSERTION-SORT$(A)$

```
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1 .. j − 1].
4       i = j − 1
5       while i > 0 and A[i] > key
6           A[i + 1] = A[i]
7           i = i − 1
8       A[i + 1] = key
```

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

The loop does 3 tasks:

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

Select the *key* to insert

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

The loop does 3 tasks:

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

Select the $key$ to insert

Make room in the sorted sub-array

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

The loop does 3 tasks:

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

Select the $key$ to insert

Make room in the sorted sub-array

Insert the $key$ in place

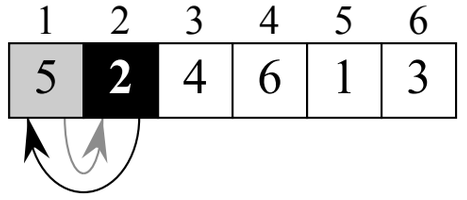
# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

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

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

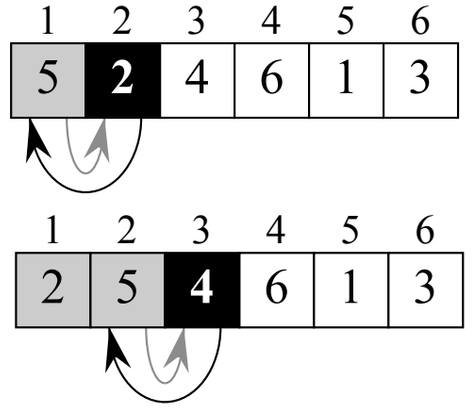



INSERTION-SORT$(A)$

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

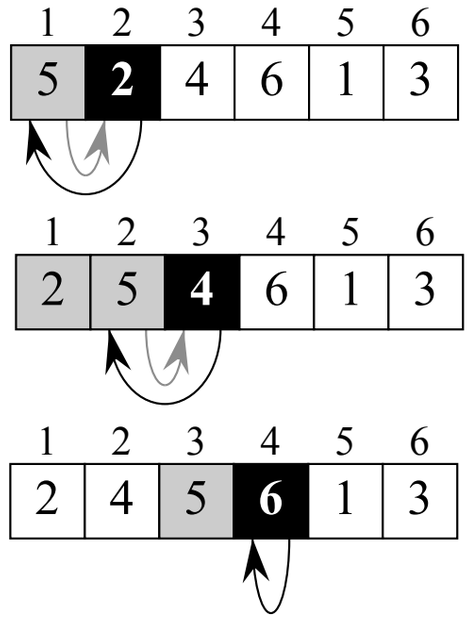



INSERTION-SORT($A$)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

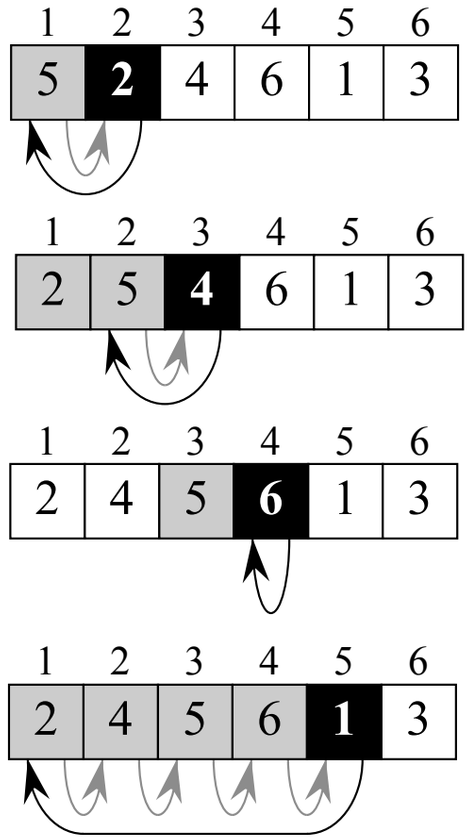



INSERTION-SORT($A$)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

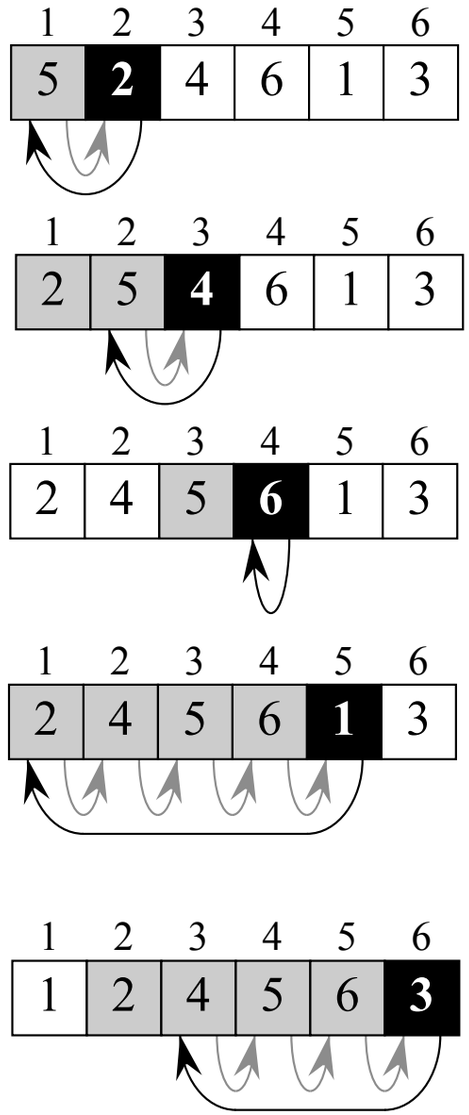



INSERTION-SORT($A$)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$

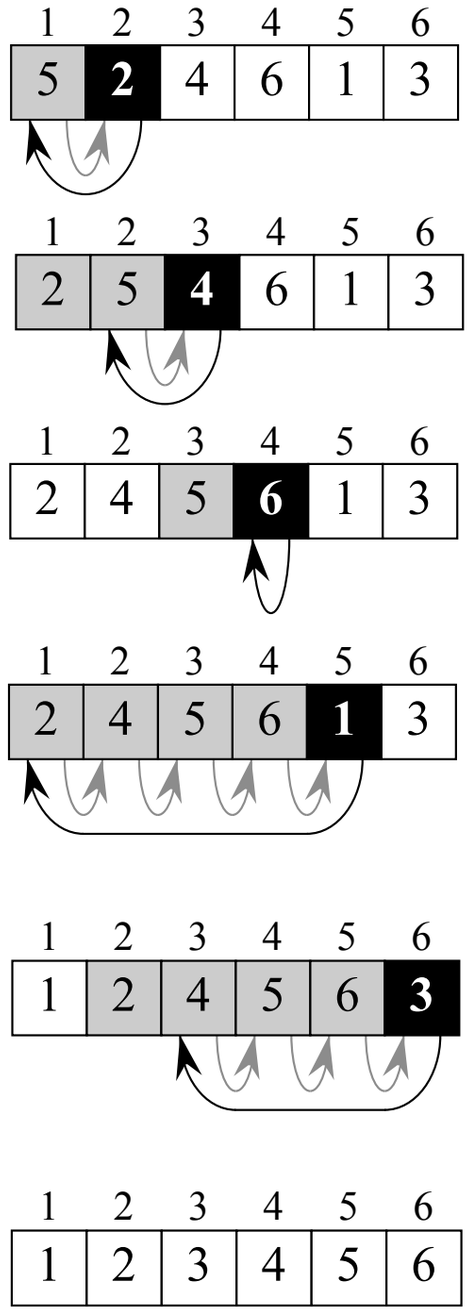



INSERTION-SORT($A$)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

# Insertion sort: example

Consider the execution of the insertion sort in the array $A = [5,2,4,6,3,1]$




INSERTION-SORT($A$)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

# What about correctness?

- The algorithm iterates the three main steps in a for-loop indexed by $j$
- The sub-array $A[1..j-1]$ constitutes the currently **sorted hand**
- The sub-array $A[j..n]$ corresponds to the pile of cards still on the table — *they still are in the original configuration*
- In fact, elements in $A[1..j-1]$ are the elements **originally** in positions $1$ through $j-1$, but now sorted.

**Loop Invariant**

*At the start of each iteration of the for-loop, the sub-array $A[1..j-1]$ consists of the element originally in $A[1..j-1]$, but in sorted order.*

# Loop Invariants

- We use loop invariants to help us understand why an algorithm is correct

- We must prove three things about a loop invariant:
  - **Initialisation:** it holds true prior to the first iteration of the loop
  - **Maintenance:** if it is true before an iteration of the loop, then it remains true before the next iteration
  - **Termination:** when the loop terminates, the invariant gives a useful property that helps show that the algorithm is correct

# Insertion Sort: initialisation

**Loop Invariant**

*At the start of each iteration of the for-loop, the sub-array $A[1..j-1]$ consists of the element originally in $A[1..j-1]$, but in sorted order.*

We have to show that the loop invariant holds before the first loop iteration:

- Before the first loop iteration $j = 2$,
- Therefore $A[1..j-1]$ consists of just a single element $A[1]$ which is in fact the original element in $A[1]$.
- Moreover, the sub-array $A[1..j-1] = A[1]$ is trivially sorted

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

# Insertion Sort: maintenance

**Loop Invariant**

*At the start of each iteration of the for-loop, the sub-array $A[1..j-1]$ consists of the element originally in $A[1..j-1]$, but in sorted order.*

We have to show that each iteration maintains the loop invariant:

- Informally*, the inner while-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 $A[j]$ among the elements in $A[1..j-1]$ (lines 4–7) and places it (line 8)
- At this point the sub-array $A[1..j]$ is sorted and contains the elements originally in $A[1..j]$.
- Incrementing $j$ for the next iteration of the for-loop then preserves the loop invariant.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

(*) A more formal treatment of the while-loop of lines 5—7 would require to define and check a loop invariant.

# Insertion Sort: termination

**Loop Invariant**

*At the start of each iteration of the for-loop, the sub-array $A[1..j-1]$ consists of the element originally in $A[1..j-1]$, but in sorted order.*

We examine what happens when the loop terminates:

- The for-loop terminates when $j > A.length$, that is $j = n + 1$.
- By substituting $n + 1$ for $j$ in the wording of the loop invariant we have that: “the sub-array $A[1..n]$ consists of the element originally in $A[1..n]$, but in sorted order”
- Since $A[1..n]$ is the entire array, we conclude that the array $A$ is sorted. Hence the algorithm is **correct**.

INSERTION-SORT($A$)

```
1  for j = 2 to A.length
2      key = A[j]
3
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

# Runtime analysis

For each $j = 2,3,\ldots,n$, we let $t_j$ denote the number of times the while-loop test in line 5 is executed for that value of $j$. Note that when a for or while loop exits, the test is executed one time more than the loop body.

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

# Runtime analysis

For each $j = 2,3,\ldots,n$, we let $t_j$ denote the number of times the while-loop test in line 5 is executed for that value of $j$. Note that when a for or while loop exits, the test is executed one time more than the loop body.

| INSERTION-SORT$(A)$ | | *cost* | *times* |
|---|---|---|---|
| 1 | **for** $j = 2$ **to** $A.length$ | $c_1$ | $n$ |
| 2 | $key = A[j]$ | $c_2$ | $n-1$ |
| 3 | // Insert $A[j]$ into the sorted sequence $A[1..j-1]$. | 0 | $n-1$ |
| 4 | $i = j-1$ | $c_4$ | $n-1$ |
| 5 | **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 | $A[i+1] = A[i]$ | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 7 | $i = i-1$ | $c_7$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 8 | $A[i+1] = key$ | $c_8$ | $n-1$ |

# Runtime analysis

For each $j = 2,3,\ldots,n$, we let $t_j$ denote the number of times the while-loop test in line 5 is executed for that value of $j$. Note that when a for or while loop exits, the test is executed one time more than the loop body.

| INSERTION-SORT$(A)$ | *cost* | *times* |
|---|---|---|
| 1 **for** $j = 2$ **to** $A.length$ | $c_1$ | $n$ |
| 2 $key = A[j]$ | $c_2$ | $n-1$ |
| 3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$. | 0 | $n-1$ |
| 4 $i = j-1$ | $c_4$ | $n-1$ |
| 5 **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 $A[i+1] = A[i]$ | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 7 $i = i-1$ | $c_7$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 8 $A[i+1] = key$ | $c_8$ | $n-1$ |

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

# Best-case analysis

The best case occurs when $t_j$ is minimal for each $j = 2,3,\ldots,n$. That is when the array $A$ is already sorted and $t_j = 1$ for all $j = 2,3,\ldots,n$.

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

# Best-case analysis

The best case occurs when $t_j$ is minimal for each $j = 2,3,\ldots,n$. That is when the array $A$ is already sorted and $t_j = 1$ for all $j = 2,3,\ldots,n$.

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

It's a **linear function** of $n$.

Of the form $an + b$ for some constants $a$ and $b$ that depend on $c_i$ (for $i = 1,\ldots,8$)

# Worst-case analysis

The worst case occurs when $t_j$ is maximal for each $j = 2,3,\ldots,n$. That is when the array $A$ is in reverse sorted order (i.e., in decreasing order).
In this case we have to compare $A[j]$ with the entire sub-array $A[1..j-1]$. Thus having $t_j = j$ for all $j = 2,3,\ldots,n$.

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

(*) Arithmetic series. Look CLRS pp. 1146

# Worst-case analysis

The worst case occurs when $t_j$ is maximal for each $j = 2,3,\ldots,n$. That is when the array $A$ is in reverse sorted order (i.e., in decreasing order).
In this case we have to compare $A[j]$ with the entire sub-array $A[1..j-1]$. Thus having $t_j = j$ for all $j = 2,3,\ldots,n$.

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

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

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

It's a **quadratic function** of $n$.

Of the form $an^2 + bn + c$ for some constants $a, b$ and $c$ that depend on $c_i$ (for $i = 1,\ldots,8$)

(*) Arithmetic series. Look CLRS pp. 1146

# Order of Growth

- Often, it is not worth determining the exact running time
  - Useful to abstract from computer architecture details

- For **large enough inputs**, the multiplicative constants (coefficients) and the lower order terms are dominated by the effect of the input size itself

- We can study the **asymptotic efficiency**

# Asymptotic Analysis

- GOAL: to simplify analysis of running time by getting rid of “details”, which may be affected by specific implementation and hardware

- Capture the essence of the running time $T(n)$:
  - Defined as functions whose domain are the set of natural numbers $\mathbb{N} = \{0,1,2,\ldots\}$
  - Use a notation that is convenient for describing worst-case running time (e.g., Insertion sort runs in quadratic time $\Theta(n^2)$)

# $\Theta$-notation

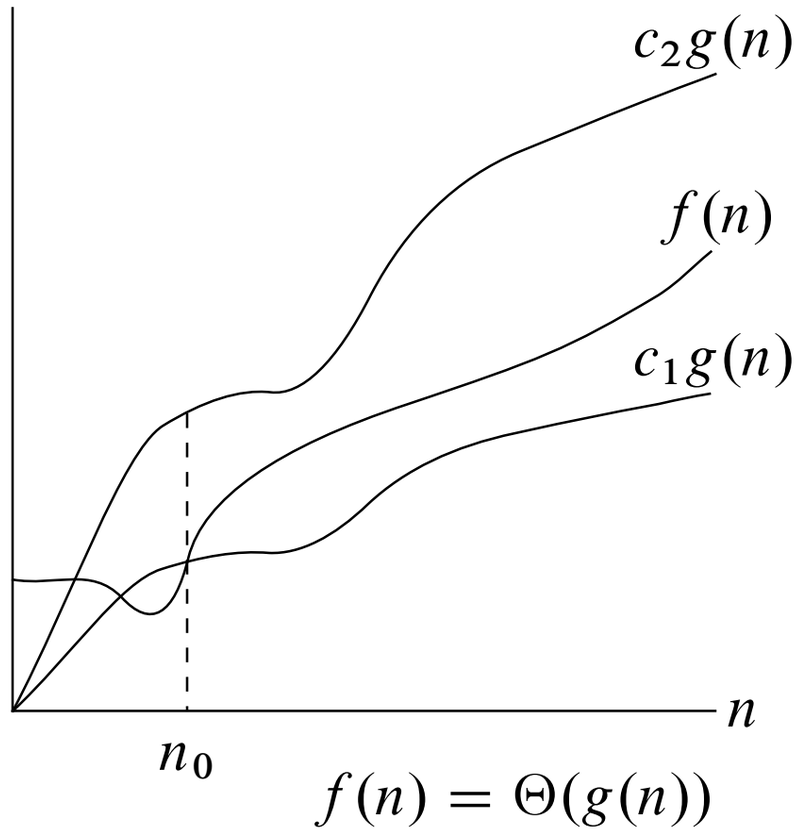
**Definition:** For a function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

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

- $f(n) = \Theta(g(n))$ means $f(n) \in \Theta(g(n))$
- Asymptotically **tight bound**
- You may read as "$f(n)$ and $g(n)$ have equal asymptotic growth"

Examples

- $T(n) = 23n^3 - 2n = \Theta(n^3)$
- $T(n) = n \log n + 2n = \Theta(n \log n)$
- $T(n) = 35 \cdot 2^n + n^{30} = \Theta(2^n)$
- $T(n) = 100 = \Theta(1)$



# Proof Example

We want to show that $23n^3 - 2n = \Theta(n^3)$

1. **Lower Bound:** we have to find $c_1, n_1 > 0$ such that $0 \leq c_1 n^3$ and $c_1 n^3 \leq 23n^3 - 2n$ for all $n \geq n_1$
   - For all $n \geq n_1 > 0$, dividing $n^3$ on both sides of the inequality $c_1 n^3 \leq 23n^3 - 2n$ yields $c_1 \leq 23 - \frac{2}{n^2}$.
   - For any value $n \geq 1$ the above inequality holds for any constant $c_1 \leq 21$
   - We can choose $n_1 = 1$ and $c_1 = 21$

2. **Upper Bound:** we have to find $c_2, n_2 > 0$ and $n_2$ such that $23n^3 - 2n \leq c_2 n^3$ for all $n \geq n_2$
   - For any $n \geq 1$ we have $23n^3 - 2n \leq 23n^3 \leq c_2 n^3$
   - Thus we can choose $n_2 = 1$ and $c_2 = 23$

3. Let $n_0 = \max(n_1, n_2) = 1$

# Quiz

Show that $23n^3 + 2n = \Theta(n^3)$

# Quiz: answer

Show that $23n^3 + 2n = \Theta(n^3)$

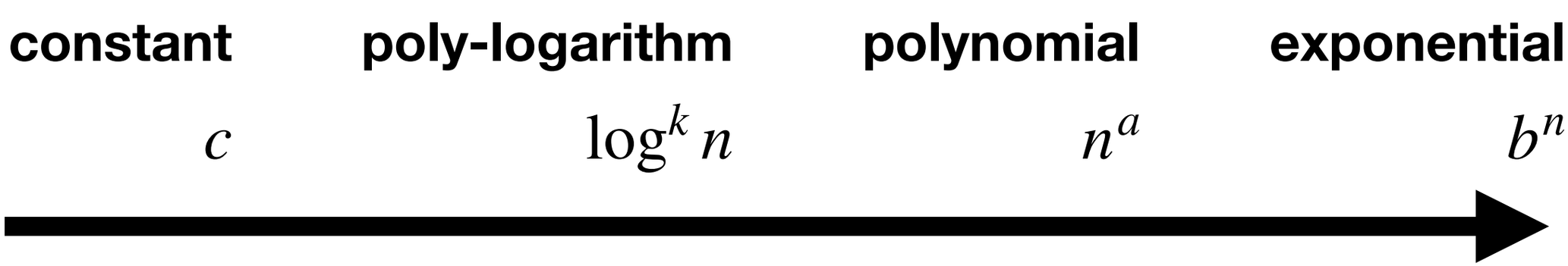
1. **Lower Bound:** we have to find $c_1, n_1 > 0$ such that $0 \leq c_1 n^3$ and $c_1 n^3 \leq 23n^3 + 2n$ for all $n \geq n_1$
   - For all $n \geq n_1 > 0$, dividing $n^3$ on both sides of the inequality $c_1 n^3 \leq 23n^3 + 2n$ yields $c_1 \leq 23 + \frac{2}{n^2}$.
   - For any value $n \geq 1$ the above inequality holds for any constant $c_1 \leq 23$
   - We can choose $n_1 = 1$ and $c_1 = 23$

2. **Upper Bound:** we have to find $c_2, n_2 > 0$ and $n_2$ such that $23n^3 + 2n \leq c_2 n^3$ for all $n \geq n_2$
   - For any $n \geq 1$ we have $23n^3 + 2n \leq 23n^3 + 2n^3 = 25n^3 \leq c_2 n^3$
   - Thus we can choose $n_2 = 1$ and $c_2 = 25$

3. Let $n_0 = \max(n_1, n_2) = 1$

# $\Theta$-notation

- The "engineering way" of manipulating $\Theta$ notation:
  1. **Ignore leading constant coefficients**
     - Example: $T(n) = 100n^2 = \Theta(n^2)$
  2. **Drop lower order terms**
     - Example: $T(n) = n^5 + n^3 + \log n = \Theta(n^5)$
- How to identify lower order terms?



# (big-O) $O$-notation

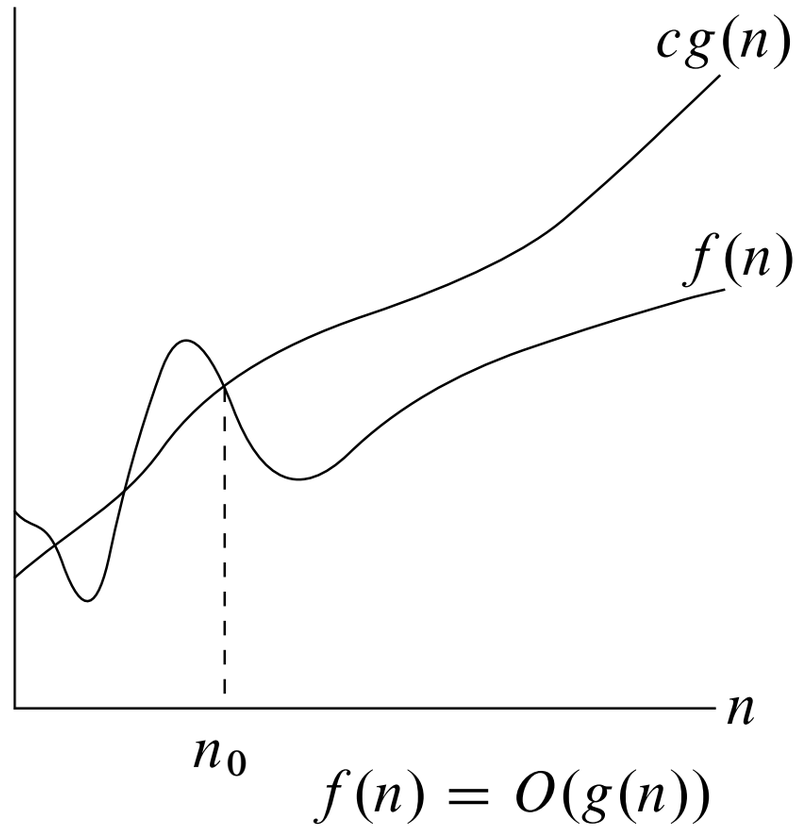
**Definition:** For a function $g(n)$, we denote by $O(g(n))$ the set of functions

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

- $f(n) = O(g(n))$ means $f(n) \in O(g(n))$
- Asymptotically **upper bound**
- You may read as "$f(n)$ grows asymptotically slower than $g(n)$"

Examples

- $T(n) = 23n^3 - 2n = O(n^4)$
- $T(n) = 23n^3 - 2n = O(n^3)$
- $T(n) = 35 \cdot 2^n + n^{30} = O(2^n)$
- $T(n) = 100 = O(\log n)$



# (big-Omega) $\Omega$-notation

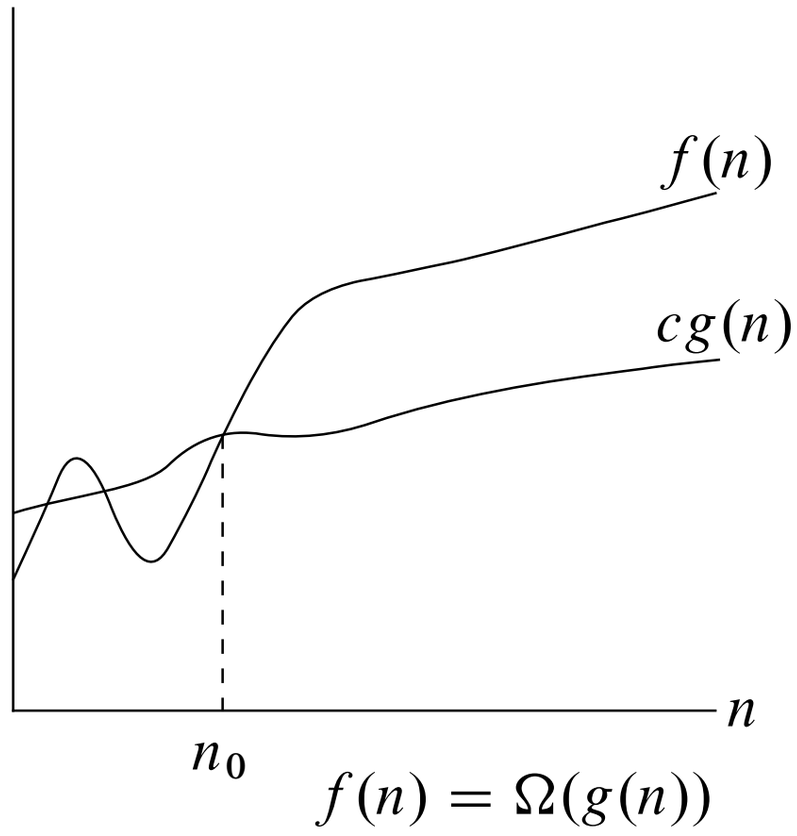
**Definition:** For a function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

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

- $f(n) = O(g(n))$ means $f(n) \in O(g(n))$
- Asymptotically **lower bound**
- You may read as "$f(n)$ grows asymptotically faster than $g(n)$"

Examples

- $T(n) = 23n^3 - 2n = \Omega(n^3)$
- $T(n) = 23n^3 - 2n = \Omega(n)$
- $T(n) = 2^n + n^{30} = \Omega(n^{100})$
- $T(n) = 100 = \Omega(1)$



# Asymptotic Comparison

**Theorem:** For any two functions $f(n)$ and $g(n)$, we have that

$$f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

- **Transitivity:**

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n))$$

- **Reflexivity:**

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

- **Symmetry:**

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

- **Transposed Symmetry:**

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

# Using Asymptotic notation

Consider the worst-case running time for insertion sort seen before

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

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

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

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

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

We remove clutter and simplify the analysis!

# Using Asymptotic notation

Consider the worst-case running time for insertion sort seen before

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

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

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

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

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

We remove clutter and simplify the analysis!

**Remark:** it is not the usual algebra!

# Quiz

**Question 1.**
Identifying asymptotic notation. (Note: lg means logarithm in base 2)

(1.1) [5 Pts] Mark **ALL** the correct answers. $\lg n^2 + \lg 8^n + \sqrt{n^2}$ is

☐ **a)** $\Theta(\lg n)$ ☐ **b)** $\Theta(n)$ ☐ **c)** $\Theta(\sqrt{n})$ ☐ **d)** $\Theta(n^2)$ ☐ **e)** $\Theta(2^n)$

(1.2) [5 Pts] Mark **ALL** the correct answers. $n \lg n^2 + \lg 32^n + 100n$ is

☐ **a)** $O(\lg n)$ ☐ **b)** $O(n)$ ☐ **c)** $O(n \lg n)$ ☐ **d)** $O(n^2)$ ☐ **e)** $O(2^n)$

(1.3) [5 Pts] Mark **ALL** the correct answers. $n \lg n^2 + \lg 32^n + 100n$ is

☐ **a)** $\Omega(\lg n)$ ☐ **b)** $\Omega(n)$ ☐ **c)** $\Omega(n \lg n)$ ☐ **d)** $\Omega(n^2)$ ☐ **e)** $\Omega(2^n)$

**Question 1.**
Identifying asymptotic notation. (Note: lg means logarithm in base 2)

(1.1) [3 Pts] Mark **ALL** the correct answers. $\lg n^{100} + \sqrt{n} + \lg 8^n$ is

☐ **a)** $\Theta(\lg n)$ ☐ **b)** $\Theta(n)$ ☐ **c)** $\Theta(\sqrt{n})$ ☐ **d)** $\Theta(n^2)$ ☐ **e)** $\Theta(2^n)$

(1.2) [3 Pts] Mark **ALL** the correct answers. $n \lg n^2 + \lg 32^n + 100n^2$ is

☐ **a)** $O(\lg n)$ ☐ **b)** $O(n)$ ☐ **c)** $O(n \lg n)$ ☐ **d)** $O(n^2)$ ☐ **e)** $O(2^n)$

(1.3) [3 Pts] Mark **ALL** the correct answers. $n \lg n^2 + \lg 5^n + 100n^4$ is

☐ **a)** $\Omega(\lg n)$ ☐ **b)** $\Omega(n)$ ☐ **c)** $\Omega(n \lg n)$ ☐ **d)** $\Omega(n^2)$ ☐ **e)** $\Omega(2^n)$

# Learned Today

- Insertion sort
  - Runtime: $\Theta(n^2)$ and $\Omega(n)$
  - Proof of correctness
- Use of loop invariants for proving correctness

- Formal concepts of $\Theta$, $O$, and $\Omega$ notation
  - How to use them for running time analysis
  - How to easily classify functions w.r.t. their asymptotic growth