CIS 263 Introduction to Data Structures and Algorithms

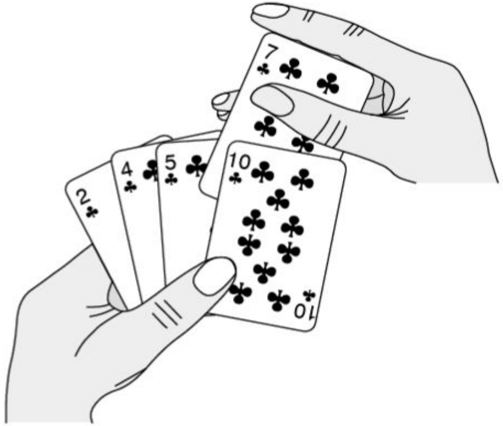
Designing Algorithms: Introduction to Complexity Analysis

Designing Algorithms

- **Incremental approach:** *Insertion Sort*
- **Divide and conquer:** *Merge Sort*

Designing Algorithms

- **Incremental approach:** Insertion Sort
- **Divide and conquer:** Merge Sort



Insertion sort

- Start with the 2nd element
- Find it's position among all those are before it
- If needed to change position (and as identified the position) move others to the right (right shift)

https://www.geeksforgeeks.org/insertion-sort/

Designing Algorithms

- **Incremental approach:** Insertion Sort
- **Divide and conquer:** Merge Sort

6 5 3 1 8 7 2 4

How Efficient an Algorithm is?

Insertion Sort

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

Complexity Analysis

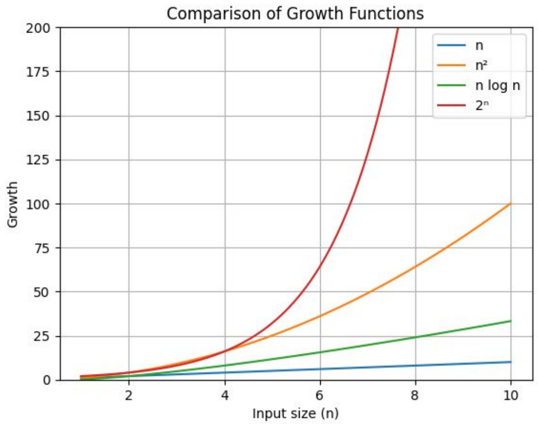
Insertion Sort

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

Complexity Analysis *- Concepts & Terminologies*

Growth functions describe how an algorithm’s **resource usage** (usually time or space)

- Increases as the **input size n** grows **asymptotic behavior** (what happens when *n* is large).
- They let us compare algorithms based on *scalability*, not on machine-dependent details.
- They ignore constants and low-order terms that don’t affect scalability



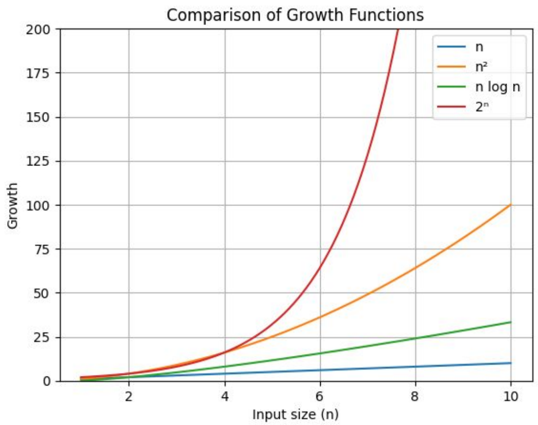
Complexity Analysis *- Concepts & Terminologies*

Growth functions describe how an algorithm's **resource usage** (usually time or space)

- Increases as the **input size n** grows **asymptotic behavior** (what happens when *n* is large).
- They let us compare algorithms based on *scalability*, not on machine-dependent details.
- They ignore constants and low-order terms that don't affect scalability

Example:

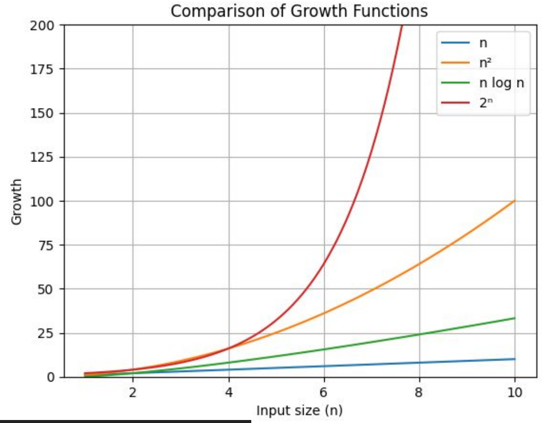
- $3n^2 + 5n + 20$ grows like n^2
- We say its growth function is **quadratic**



Complexity Analysis *- Concepts & Terminologies*

Growth functions describe how an algorithm's **resource usage** (usually time or space)

- Increases as the **input size n** grows **asymptotic behavior** (what happens when *n* is large).
- They let us compare algorithms based on *scalability*, not on machine-dependent details.
- They ignore constants and low-order terms that don't affect scalability



Example:

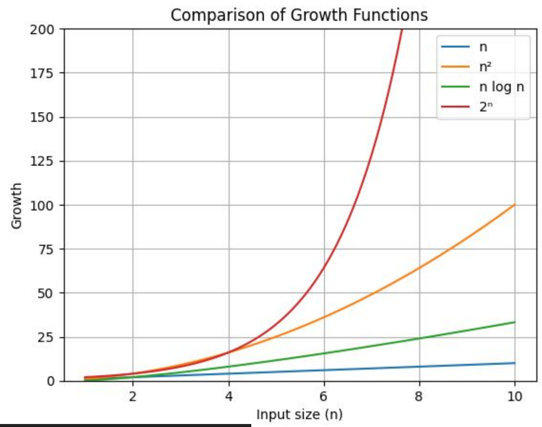
- $3n^2 + 5n + 20$ grows like n^2
- We say its growth function is **quadratic**

Big-O: upper bound on growth (e.g., $O(n^2)$)

Complexity Analysis - *Concepts & Terminologies*

Growth functions describe how an algorithm's **resource usage** (usually time or space)

- Increases as the **input size n** grows **asymptotic behavior** (what happens when *n* is large).
- They let us compare algorithms based on *scalability*, not on machine-dependent details.
- They ignore constants and low-order terms that don't affect scalability



Example:

- $3n^2 + 5n + 20$ grows like n^2
- We say its growth function is **quadratic**

Big-O: upper bound on growth (e.g., $O(n^2)$)

$$f(n) = 2n^2 + 3n + 1 \Rightarrow O(n^2)$$

Complexity Analysis *- Concepts & Terminologies*

Growth Function	Name	Example
$O(1)$	Constant	Array access
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Linear search
$O(n \log n)$	Linearithmic	Merge sort
$O(n^2)$	Quadratic	Bubble sort
$O(n^3)$	Cubic	Matrix multiplication (basic)
$O(2^n)$	Exponential	Subset generation
$O(n!)$	Factorial	Traveling Salesman (brute force)

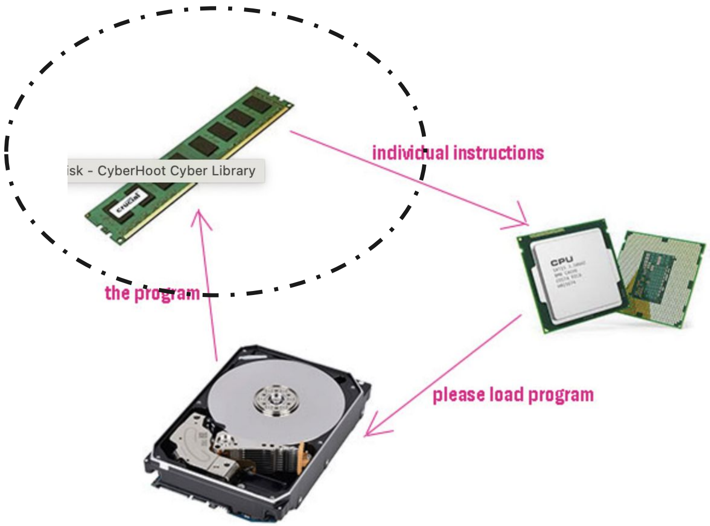
Example growth rates (from best to worst)

Complexity Analysis

- Space complexity
- Time Complexity

Space Complexity

- The **space Complexity of an algorithm** is the **total space taken by the algorithm** with **respect to the input size**.
- Space complexity includes **both Auxiliary space and space used by input**.



Space Complexity

- *Resource independent - Abstractions at the hardware (memory) level*

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

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	$n - 1$

Time Complexity or *Complexity (in General)*

- *Resource independent - Abstractions at the algorithm level*

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

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	$n - 1$

Time Complexity or *Complexity (in General)*

- *Resource independent - Abstractions at the algorithm level*

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

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	$n - 1$

Complexity Analysis – Insertion Sort

Complexity Analysis – Insertion Sort

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

Complexity Analysis – Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\boxed{key = A[i]}$	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	$n-1$

Complexity Analysis – Insertion Sort

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

Complexity Analysis – Insertion Sort

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

Complexity Analysis – Insertion Sort

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

Complexity Analysis – Insertion Sort

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

Complexity Analysis – Insertion Sort

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

Introduction to Algorithms, by Thomas Cormen et al.

- **Total cost**

Complexity Analysis – Insertion Sort

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

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

Introduction to Algorithms, by Thomas Cormen et al.

- **Best case**

$an + b$ for constants a and b

- **Linear function**

Complexity Analysis – Insertion Sort

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

Introduction to Algorithms, by Thomas Cormen et al.

- **Worst case**

  $an^2 + bn + c$ for constants a, b, and c

- **Quadratic function**

Complexity Analysis – Insertion Sort

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

$$\boxed{\begin{aligned} \sum_{k=1}^{n} k &= \frac{n(n+1)}{2} \\ &= \Theta(n^2) . \end{aligned}}$$

Introduction to Algorithms, by Thomas Cormen et al.

- **Worst case**

  $an^2 + bn + c$ for constants a, b, and c

- **Quadratic function**

Complexity Analysis – Insertion Sort

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

$$\boxed{\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2)\,.}$$

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

Introduction to Algorithms, by Thomas Cormen et al.

- **Worst case**

  $an^2 + bn + c$ for constants a, b, and c

- **Quadratic function**

Complexity Analysis – Insertion Sort

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

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

Introduction to Algorithms, by Thomas Cormen et al.

- **Worst case**

  $an^2 + bn + c$ for constants a, b, and c

- **Quadratic function**

Complexity Analysis – Insertion Sort

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

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

Upper Bound of the Algorithm Runtime

Introduction to Algorithms, by Thomas Cormen et al.

- **Worst case**

  $an^2 + bn + c$ for constants a, b, and c

- **Quadratic function**

Complexity Analysis – Insertion Sort

Introduction to Algorithms, by Thomas Cormen et al.

- Best case
- Worst case
- **Average case**

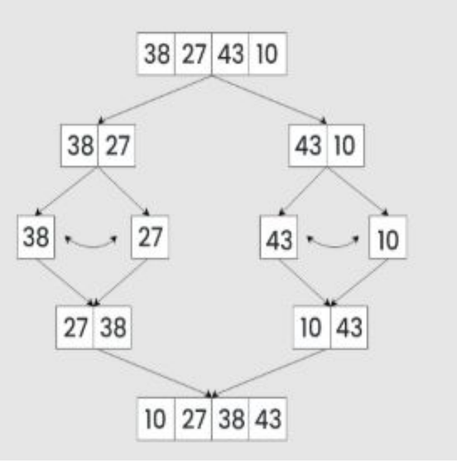
Sorting Algorithms – Worst-Case Time Complexity

Algorithm	Worst-Case Time	Notes
Bubble Sort	$O(n^2)$	Educational, very inefficient
Selection Sort	$O(n^2)$	Always $O(n^2)$
Insertion Sort	$O(n^2)$	Fast for small or nearly sorted data
Shell Sort	$O(n^2)$	Depends on gap sequence
Quick Sort	$O(n^2)$	Worst case with poor pivot selection
Tree Sort	$O(n^2)$	Occurs when tree becomes skewed
Bucket Sort	$O(n^2)$	Worst case when all items fall in one bucket

Algorithm	Worst-Case Time	Notes
Merge Sort	$O(n \log n)$	Stable; extra memory required
Heap Sort	$O(n \log n)$	In-place; not stable
Tim Sort	$O(n \log n)$	Python & Java default
Counting Sort	$O(n + k)$	Non-comparison; k = value range
Radix Sort	$O(nk)$	Non-comparison; k = digits
Bucket Sort	$O(n + k)$	Average case (worst can be quadratic)

Designing Algorithms

- **Incremental approach:** Insertion Sort
- **Divide and conquer:** Merge Sort



Recursion

Recursion

In mathematics, the **factorial** of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. The factorial of n also equals the product of n with the next smaller factorial:

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1 \\ &= n \times (n-1)! \end{aligned}$$

For example,

$$5! = 5 \times 4! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

Recursion

In mathematics, the **factorial** of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. The factorial of n also equals the product of n with the next smaller factorial:

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1 \\ &= n \times (n-1)! \end{aligned}$$

For example,

$$5! = 5 \times 4! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

Recursion

In mathematics, the **factorial** of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. The factorial of n also equals the product of n with the next smaller factorial:

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1 \\ &= n \times (n-1)! \end{aligned}$$

For example,

$$5! = 5 \times 4! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

Recursion

In mathematics, the **factorial** of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. The factorial of n also equals the product of n with the next smaller factorial:

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1 \\ &= n \times (n-1)! \end{aligned}$$

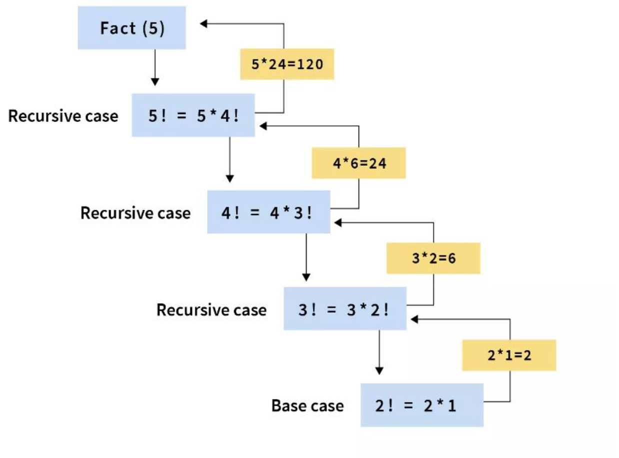
For example,

$$5! = 5 \times 4! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

Recursion

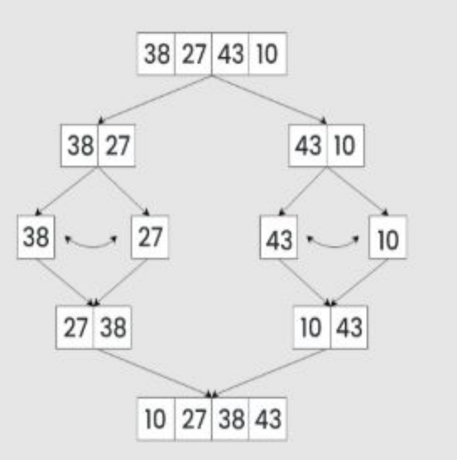
Recursion:

- The factorial of ***n*** also equals the product of ***n*** with the next smaller factorial



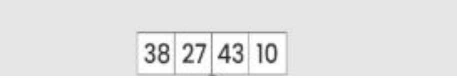
Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.



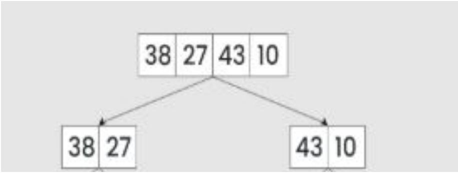
Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.



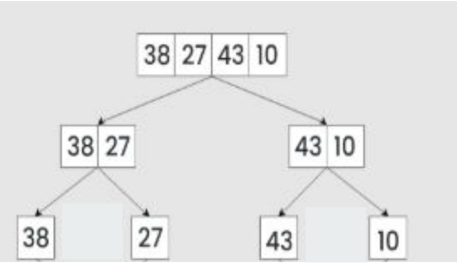
Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.



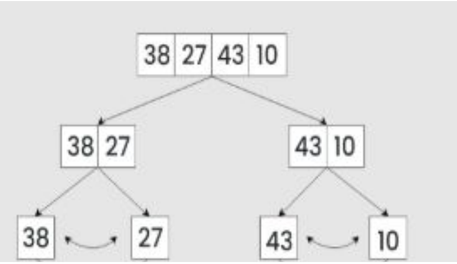
Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.



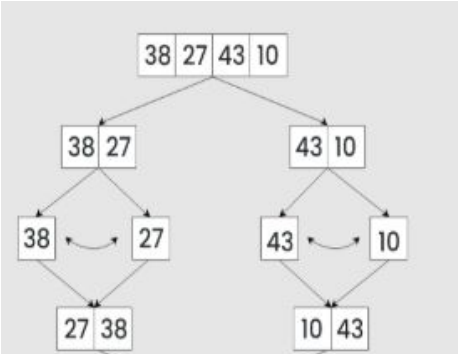
Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.



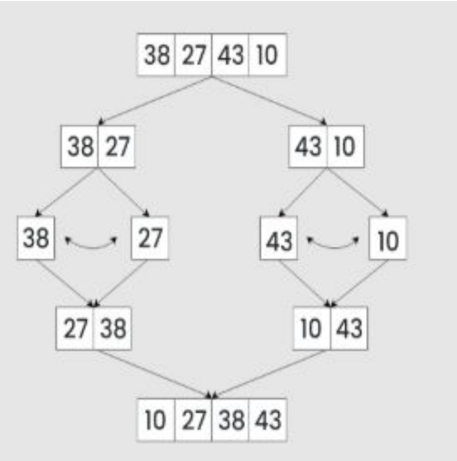
Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.



Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.



Merge Sort

Complexity

Worst Case Time Complexity [Big-O]: **O(n*log n)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Comparing complexities

1K data points	Insertion sort: O(1K**2) ~ 1M Bubble sort: O(1K**2) ~ 1M Merge sort: O(1K * log (1K)) ~ 7K

Comparing complexities

1K data points	Insertion sort: O(1K**2) ~ 1M Bubble sort: O(1K**2) ~ 1M Merge sort: O(1K * log (1K)) ~ 7K
1000K (1M) data points	Insertion sort: O(1M**2) ~ 1 Trillion Bubble sort: O(1M**2) ~ 1 Trillion Merge sort: O(1M * log (1M)) ~ 14M