UNIT 1:
Fundamentals of the Analysis of Algorithmic Efficiency…

Mathematical Analysis of Non-recursive Algorithms

O (Big-O)notation

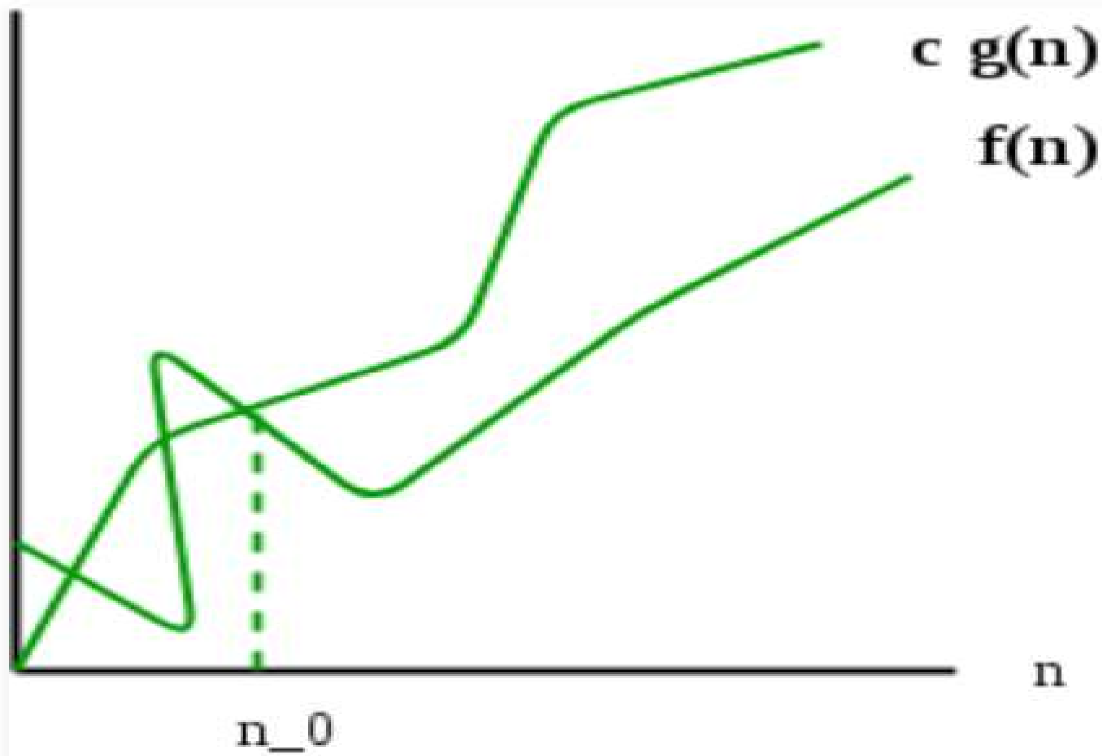
Definition:

A function f(n) is said to be in O(g(n)), denoted

$$f(n) \in O(g(n)),$$

if f(n) is **bounded above** by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some non-negative integer n_0 such that

$$\mathbf{f(n) \le cg(n) \text{ for all } n \ge n_0}$$



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

Example: O notation

- Let $f(n) = 10n + 5$,

  $10n + 5 \le 11n$

  Here $g(n) = 11n$, then $f(n) \in O(n)$ and $c=11$

  and for $n \ge 6$

Ω (Omega) notation

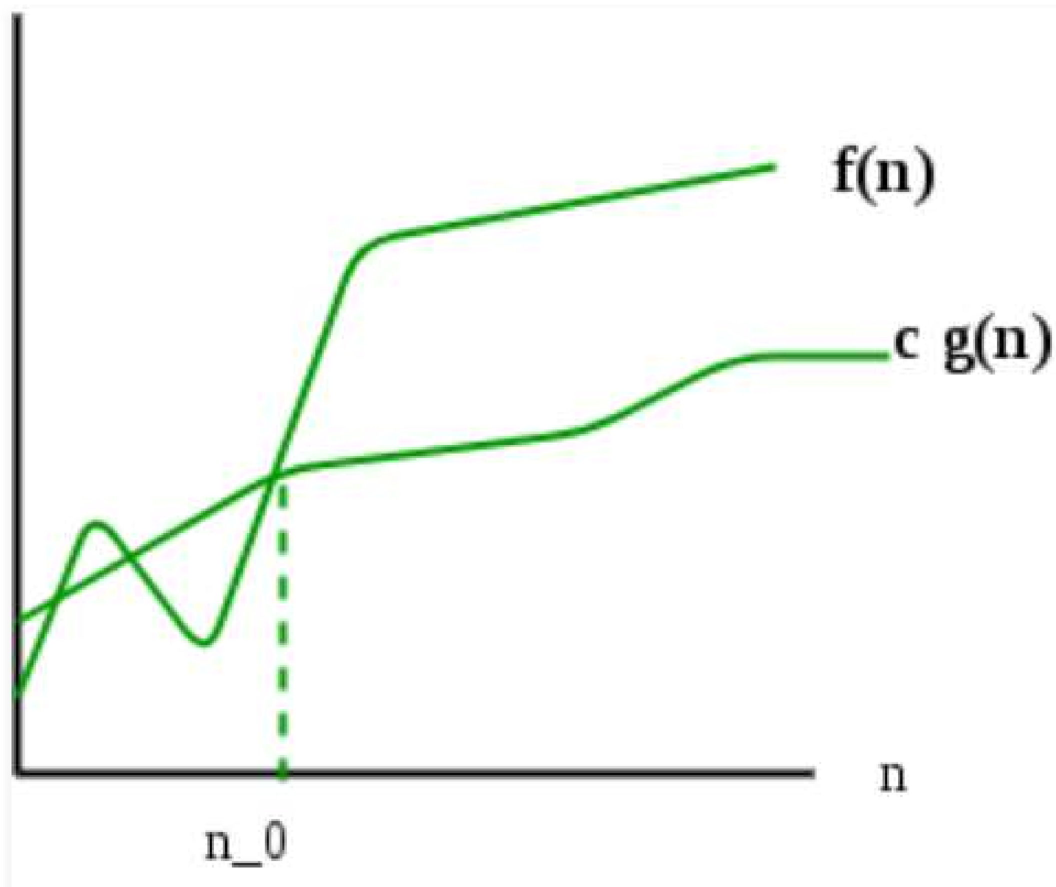
Definition:

A function $f(n)$ is said to be in $\Omega(g(n))$, denoted

$$f(n) \in \Omega(g(n)),$$

if $f(n)$ is **bounded below** by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some non-negative integer n_0 such that

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



f(n) = Omega(g(n))

Θ (Theta) notation

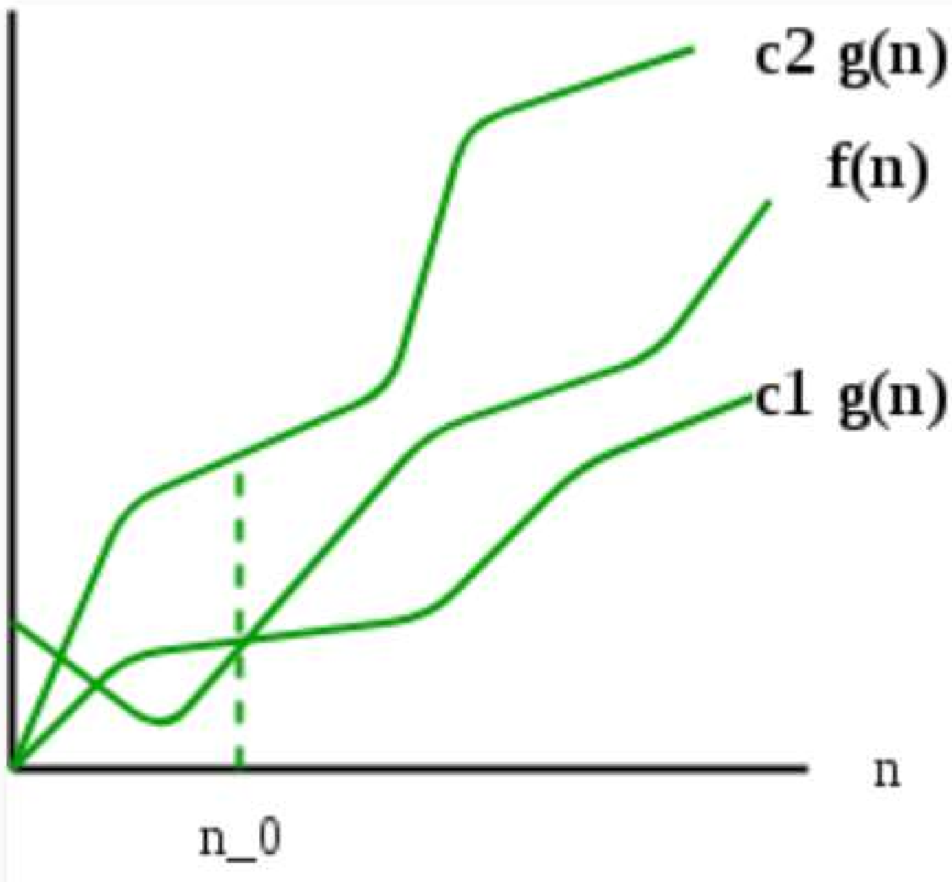
Definition:

A function f(n) is said to be in Θ(g(n)), denoted

$$f(n) \in \Theta(g(n)),$$

if f(n) is **bounded both above and below** by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constant c_1 and c_2 and some non-negative integer n_0 such that

$$\mathbf{c_2 g(n) \le f(n) \le c_1 g(n) \text{ for all } n \ge n_0}$$

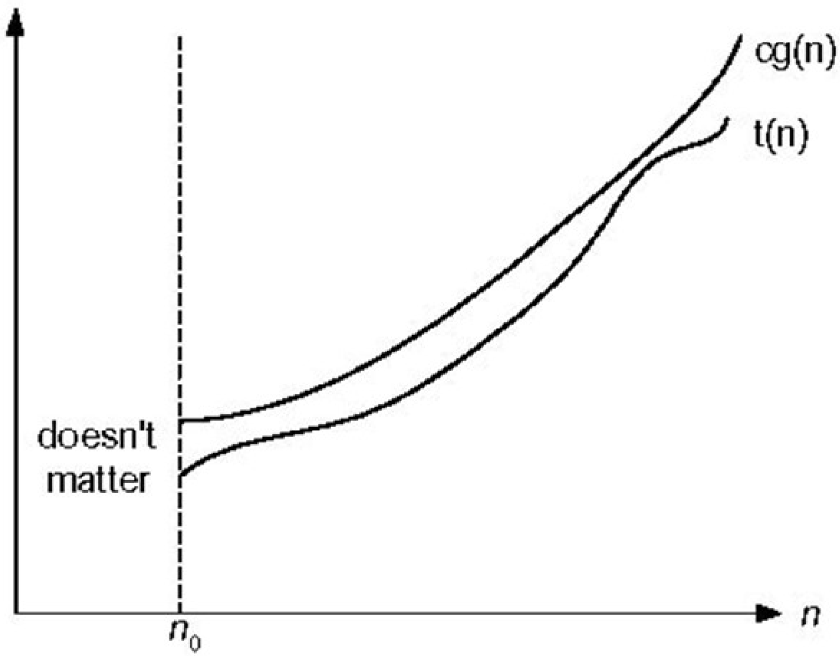


f(n) = theta(g(n))

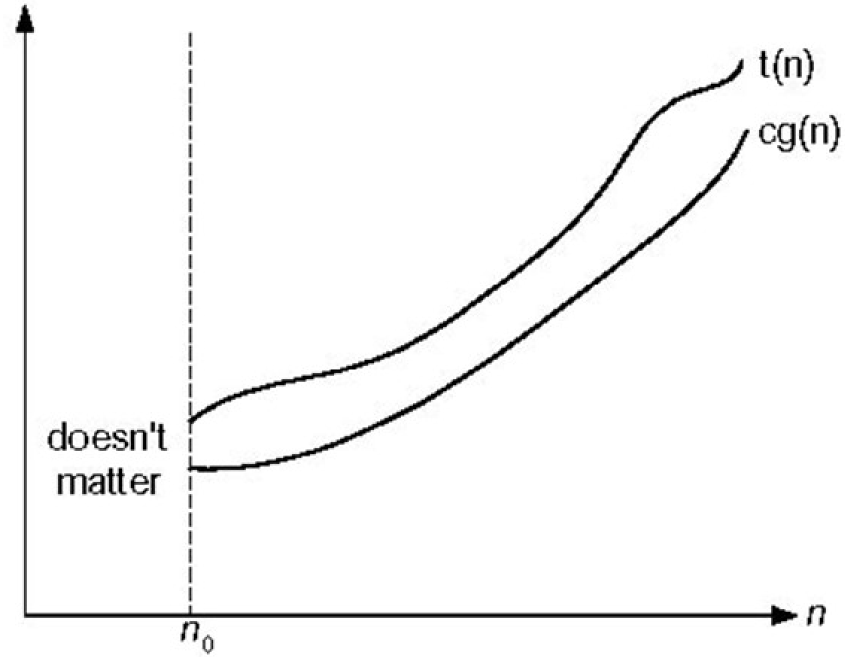
Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

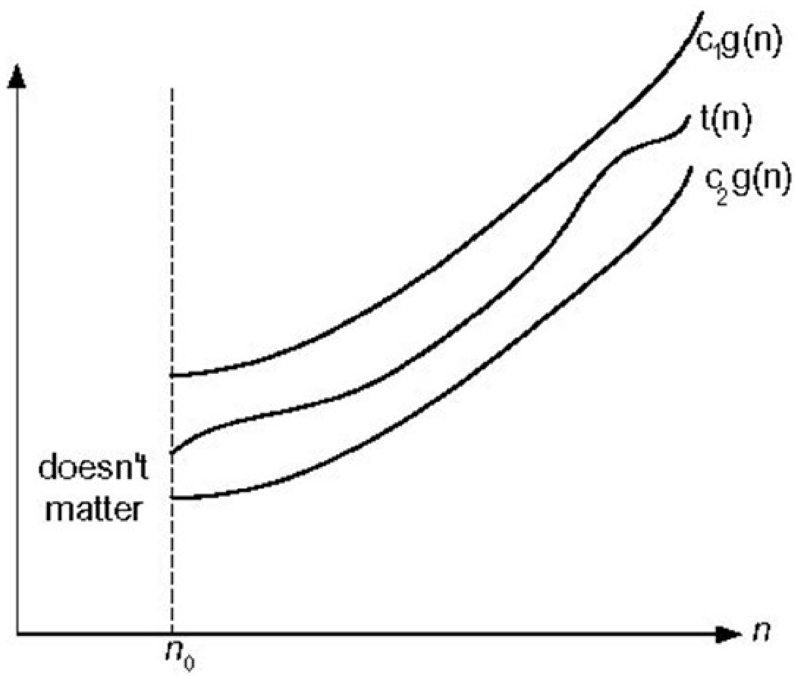
- O(g(n)): class of functions f(n) that grow no faster than g(n)
- Θ(g(n)): class of functions f(n) that grow at same rate as g(n)
- Ω(g(n)): class of functions f(n) that grow at least as fast as g(n)



Big-oh notation: $t(n) \in O(g(n))$



Big-omega notation: $t(n) \in \Omega(g(n))$



Big-theta notation: $t(n) \in \Theta(g(n))$

General plan for analyzing efficiency of non-recursive algorithms

1. Decide on parameter/s ***n*** indicating input's size
2. Identify algorithm's basic operation
3. Check/Determine worst, average, and best cases for input of size n
4. Set up a sum for the number of times the basic operation is executed
5. Simplify the sum using standard formulas and rules

Example 1: Maximum element

Finding the value of the largest element in a list of n numbers.

Example 1: Maximum element

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

Example 1: Maximum element
Algorithm analysis

1. input's size: **n – number of elements in the array**
2. basic operation: **comparison** and assignment

$$
\begin{aligned}
&\textbf{if } A[i] > maxval\\
&\qquad maxval \leftarrow A[i]
\end{aligned}
$$

3. **No worst, average, and best cases**
4. Let C(n) = number of times the basic operation is executed.

Algorithm makes one comparison on each iteration of loop, which runs for i=1 to (n-1). Therefore,

$$C(n) = \sum_{i=1}^{n-1} 1$$

Simplifying the sum using standard formulas we get:

$$C(n) = \sum_{i=1}^{n-1} 1$$

$$\sum_{i=1}^{n} 1 = n - 1 + 1$$

upper bound minus the lower bound plus one

= n-1 ∈ Θ(n)

Let's check our understanding…

```
ALGORITHM Mystery(n)
//Input: A nonnegative integer n
S ← 0
for i ←1 to n do
  S ← S + i * i
return S
```

a. What does this algorithm compute?
b. What is its basic operation?
c. How many times is the basic operation executed?
d. What is the efficiency class of this algorithm?

Let's check our understanding…

Algorithm $Mystery(n)$
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  $S \leftarrow S + i * i$
return S

What does this algorithm compute?

A. n^2

B. $\sum_{i=1}^{n} i$

C. $\sum_{i=1}^{n} i^2$

D. $\sum_{i=1}^{n} 2i$

computes the sum of 'squares of numbers' from 1 to n, i.e.,
S = 1*1 + 2*2 + 3*3 ....+ n*n

Example 2: Unique elements

Check whether all the element in a given list are distinct.

Example 2: Unique elements

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise

for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Example 2: Unique elements Algorithm analysis

1. input's size: **n – array size**
2. basic operation: **comparison**

$$\textbf{if } A[i] = A[j]$$

3. **Worst, average, and best cases may exists (depends on implementation)**
4. Let C(n) = number of times the basic operation is executed.

Example 2: Unique elements

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise

for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**

return true

Flag = 1;

Example 2: Unique elements
Algorithm analysis

1. input's size: **n – array size**
2. basic operation: **comparison**

$$\textbf{if } A[i] = A[j]$$

3. **No worst, average, and best cases**
4. Let C(n) = number of times the basic operation is executed.

Algorithm makes one comparison on each iteration of inner for loop, which runs for j=i+1 to (n-1); which in turn runs for each iteration of outer for loop with i=0 to n-2. Therefore,

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Simplifying the sum using standard formulas we get:

$$C(n) = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$= n(n-1)/2 \in \Theta(n^2)$

$$C(n) = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} n - 1 - i$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$= (n-1)\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$= (n-1)(n-2-0+1) \; - \sum_{i=0}^{n-2} i$$

$$= (n-1)^2 \; - \sum_{i=0}^{n-2} i$$

$$= (n-1)^2 \; - \frac{(n-2)(n-1)}{2}$$

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

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)\left((n-1) - \frac{(n-2)}{2}\right)$$

$$= (n-1)\left(\frac{2n-2-n+2}{2}\right)$$

$$= (n-1)\left(\frac{n}{2}\right) = \frac{n(n-1)}{2} = \frac{1}{2}n^2$$

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

Example 3: Matrix multiplication

Example 2: Matrix multiplication

ALGORITHM $MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Example 3: Matrix multiplication Algorithm analysis

1. input's size: **n – matrix order**
2. basic operation: **multiplication, addition, assignment**

$$C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$$

3. **No worst, average, and best cases**
4. Let C(n) = number of times the basic operation is executed.

Algorithm makes one multiplication on each iteration of innermost for loop, which runs for k=0 to n-1; which in turn runs for each iteration of j=0 to n-1; which in turn runs for each iteration of outer for loop with i=0 to n-1. Therefore,

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Simplifying the sum using standard formulas we get:

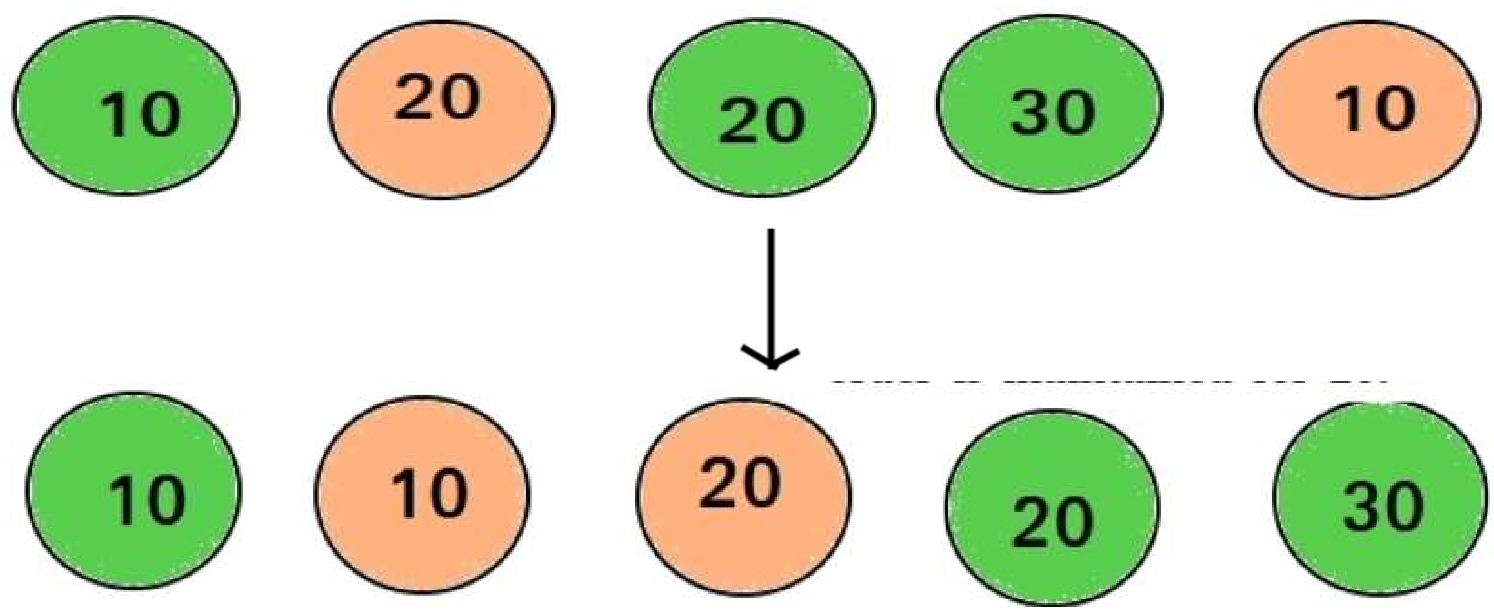
$= n^3 \in \Theta(n^3)$

Brute force

A straightforward approach to solving a problem, usually directly based on the problem’s statement and definitions of the concepts involved.

Stable sorting algorithm

A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.



Example: Insertion Sort, Merge Sort, Bubble Sort

In-place sorting algorithm

Algorithm that do not use extra space for manipulating the input but may require a small though non constant extra space for its operation

Example:

Bubble sort, Selection sort, Insertion sort, Heapsort, Shell sort

Selection sort

- in-place comparison-based algorithm
- performs well on a small list
- works by repeatedly going through the list of items, each time selecting an item according to its ordering and placing it in the correct position in the sequence.

$$A_0 \le A_1 \le \cdots \le A_{i-1} \quad \Big| \quad A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions $\qquad$ the last $n - i$ elements

Selection sort

ALGORITHM *SelectionSort*$(A[0..n-1])$

//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order

for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$
            $min \leftarrow j$
    swap $A[i]$ and $A[min]$

Visualization tools

https://visualgo.net/bn/sorting

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Selection sort: Algorithm analysis

1. input's size: **n –** Number of elements to be sorted
2. basic operation: key **comparison, swapping**

$$\textbf{if } A[j] < A[min] \quad min \leftarrow j$$

$$\text{swap } A[i] \text{ and } A[min]$$

3. **No worst, average, and best cases**
4. Let C(n) = number of times the basic operation is executed.

Algorithm makes one key comparison on each iteration of innermost for loop, which runs for j=i+1 to n-1; which in turn runs for each iteration of outer for loop with i=0 to n-2. Therefore,

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Simplifying the sum using standard formulas we get:

$$= \frac{(n-1)n}{2} \in \Theta(n^2)$$

Selection sort

Disadvantage:

- poor efficiency when dealing with a huge list of items
- its performance is easily influenced by the initial ordering of the items before the sorting process. Thus, it is only suitable for a list of few elements that are in random order.

Let's check our understanding…

ALGORITHM $BubbleSort(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$
            swap $A[j]$ and $A[j+1]$

Next session…

Fundamentals of the Analysis of Algorithmic Efficiency…

- Mathematical Analysis of Recursive algorithms