Analysis of Algorithms

Growth of Functions, Asymptotic Notation

Sorting Algorithms

Bubble Sort, Insertion Sort, Selection Sort, Runtime Analysis

Rashed Hassan Siam

Department of CSE

Eastern University

Growth of Functions

- The order of growth of an algorithm's running time gives a simple characterization of its efficiency and allows for comparison of the relative performance of different algorithms.

- When the input size 'n' is large enough, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

- We study the asymptotic efficiency of algorithms, which is how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

- An algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Measuring Input Size & Running Time

- Input Size: The input size of a problem is a measure of the number of items in the input. For example, the input size of a sorting algorithm is the number of items to be sorted.

- Running Time: The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.

- The running time of an algorithm can vary for different inputs of the same size. We are often interested in the worst-case running time, which is an upper bound on the running time for any input of a given size.

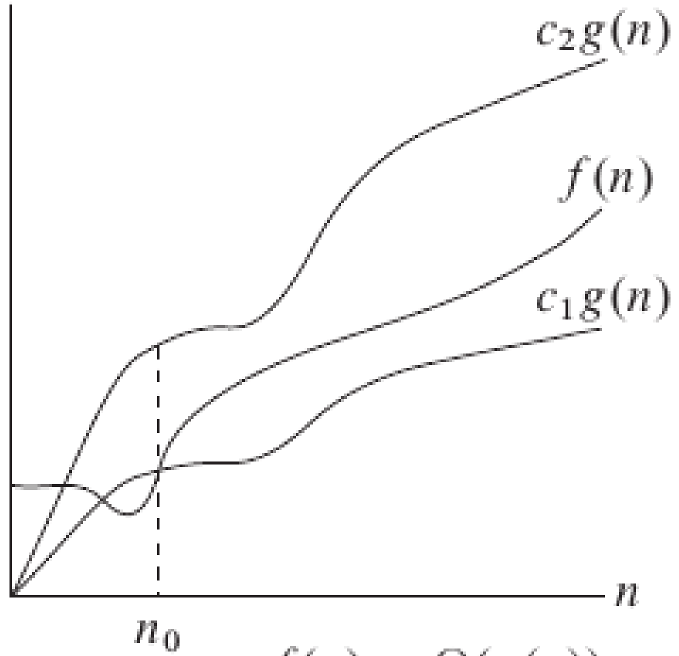
- The order of growth is the rate at which the running time increases as the input size grows. For example, an algorithm with a running time of $\Theta(n^2)$ is said to have a quadratic order of growth.

Asymptotic Notation

- Asymptotic notation is used to describe the asymptotic running time of an algorithm.
- It is defined in terms of functions whose domains are the set of natural numbers N = {0, 1, 2, ...}.
- It allows us to abstract away the details of the function and focus on the order of growth.
- Asymptotic notation can also be used to characterize other aspects of algorithms, such as the amount of space they use.

Θ-Notation (Theta-Notation)

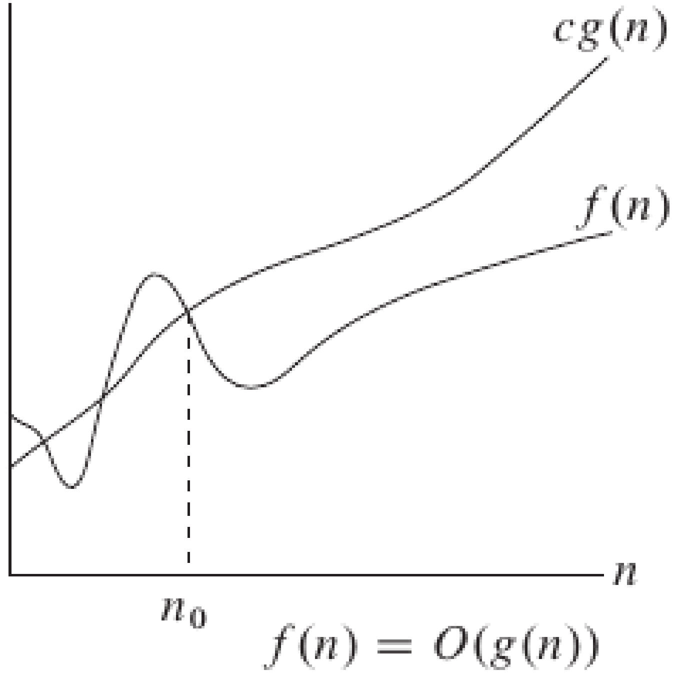
- Provides an asymptotically tight bound for a function.
- Definition: For a given function g(n), we denote by Θ(g(n)) the set of functions:
  - Θ(g(n))={f(n): there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n0$}.
- We write f(n) = Θ(g(n)) to indicate that f(n) is a member of Θ(g(n)).



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

O-Notation (Big-O Notation)

- Gives an asymptotic upper bound for a function.
- Definition: For a given function g(n), we denote by O(g(n)) the set of functions:
  - O(g(n))={f(n): there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$}.
- f(n) = Θ(g(n)) implies f(n) = O(g(n)).



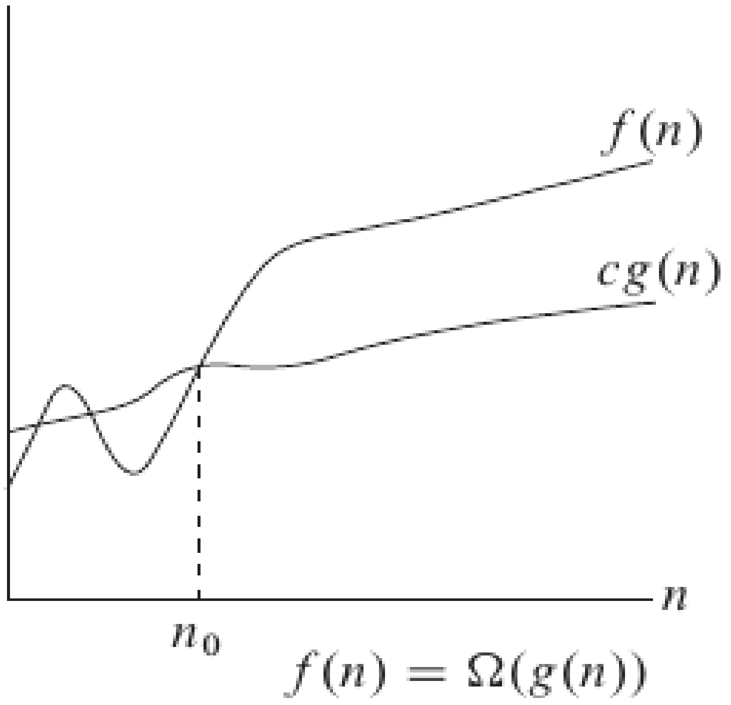
Ω-Notation (Big-Omega Notation)

- Provides an asymptotic lower bound on a function.

- Definition: For a given function g(n), we denote by Ω(g(n)) the set of functions:

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

- 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 (little-oh) and ω-notation (little-omega)

- o-notation: Denotes an upper bound that is not asymptotically tight

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

$$lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- ω-notation: Denotes a lower bound that is not asymptotically tight.

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

$$lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Sorting An Array

- Ascending order:
  - Given array - 64, 34, 25, 12, 22, 11, 90
  - Sorted array – 11, 12, 22, 25, 34, 64, 90
- Descending order:
  - Given array - 64, 34, 25, 12, 22, 11, 90
  - Sorted array – 90, 64, 34, 25, 22, 12, 11

Sorting An Array

Bubble Sort:

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The largest elements "bubble" to the top.

- Use Case: Primarily for educational purposes due to its poor performance.

- Time Complexity: O(n^2) in the average and worst case. Best case is O(n).

- Space Complexity: O(1)

```
BUBBLESORT($A$)
for $i = 1$ to $A.length - 1$
    for $j = A.length$ downto $i + 1$
        if $A[j] < A[j-1]$
            exchange $A[j]$ with $A[j-1]$
```

Sorting An Array

Insertion Sort:

- Insertion Sort builds the final sorted array one item at a time. It iterates through an input array and removes one element per iteration, finds the place the element belongs in the sorted part, and inserts it there. It's like sorting cards in your hand.

- Use Case: Efficient for small datasets or datasets that are already substantially sorted.

- Time Complexity: O(n^2) in the average and worst case. Best case is O(n).

- Space Complexity: O(1)

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

Sorting An Array

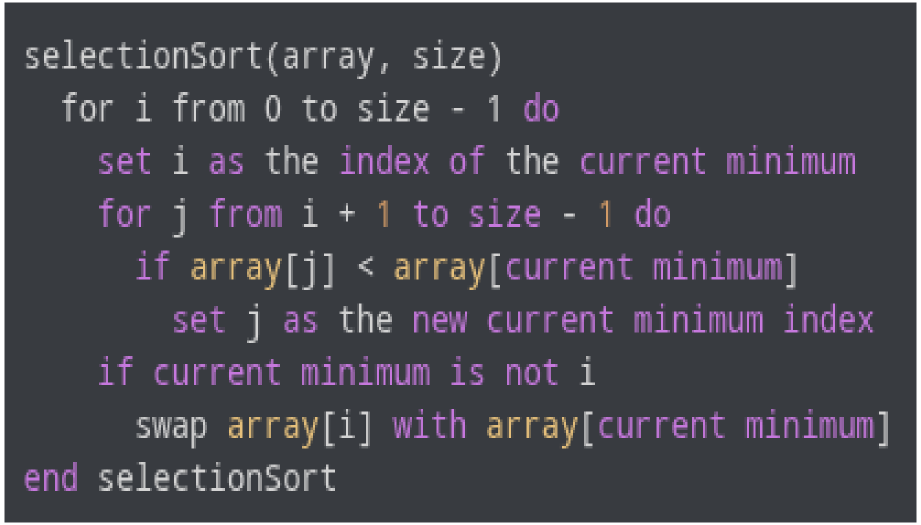
Selection Sort:

- Selection Sort divides the input list into two parts: a sorted sublist which is built up from left to right, and a sublist of the remaining unsorted items. It proceeds by finding the smallest element from the unsorted sublist and swapping it with the leftmost unsorted element.

- Use Case: Useful when memory write operations are costly, as it minimizes the number of swaps.

- Time Complexity: O(n^2) in all cases.

- Space Complexity: O(1)



```
selectionSort(array, size)
  for i from 0 to size - 1 do
    set i as the index of the current minimum
    for j from i + 1 to size - 1 do
      if array[j] < array[current minimum]
        set j as the new current minimum index
    if current minimum is not i
      swap array[i] with array[current minimum]
end selectionSort
```

Reference

- ***Introduction to Algorithms, Third Edition***
  - ***Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein***