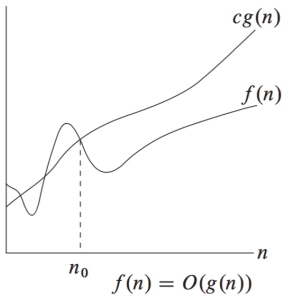
2. Asymptotic behavior of functions, asymptotic notation, further examples of simple algorithms.

1. What is asymptotic behavior of functions and asymptotic notation?

Asymptotic behavior of functions refers to how a function behaves as its input grows very large. Asymptotic notation is a way to describe this behavior in a concise and standardized way.

There are three commonly used asymptotic notations to analyze the **time complexity** of the algorithm:

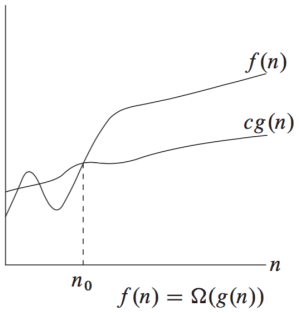
1.1 Big O notation (O) - Asymptotic upper bound



Big O notation is used to describe an ***tight*** **upper bound** on the growth rate of a function. It is used to describe how a function f(n) grows compared to another function g(n).

We say that **f(n) is O(g(n))** if there exist **constants c** and **n0** such that **f(n) ≤ c*g(n)** for all **n ≥ n0**.

1.2 Big Omega notation (Ω) - Asymptotic lower bound

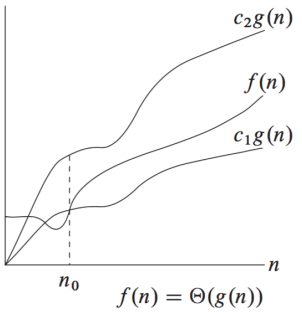


Big Omega notation is used to describe a **lower bound** on the growth rate of a function. It is used to describe how a function f(n) grows compared to another function g(n).

We say that **f(n) is Ω(g(n))** if there exist **constants c** and **n0** such that **f(n) ≥ c*g(n)**

In simpler terms, this means that f(n) grows at least as fast as g(n). For example, if f(n) = n^3 and g(n) = n^2, we can say that f(n) is Ω(g(n)), because n^3 grows at least as fast as n^2.

1.3 Big Theta notation (Θ) - Describe an exact bound on the growth rate of a function



Big-Theta notation is used to describe **the exact upper and lower bounds** on the growth rate of a function. When we compare a function f(n) to another function g(n), it describes the **relationship** between the **growth rate of f(n)** and the **growth rate of g(n)**.

We say that **f(n) is Θ(g(n))** if there exist **constants c1, c2**, and **n0** such that for all **n ≥ n0**, **0 ≤ c1g(n) ≤ f(n) ≤ c2g(n)**.

For any two functions f (n) and g(n), we have **f (n) = Θ(g(n))** if and only if **f (n) = O(g(n)) and f (n) = Ω(g(n))**.

2. Time complexity of an algorithm

The time complexity of an algorithm is a measure of how the running time of the algorithm increases as the size of the input grows. It is usually expressed as a function of the input size.

2.1 Examples of time complexity of common functions

The relationship between common time complexities (Important!):

$O(N^N) > O(N!) > O(2^N) > O(N^3) > O(N^2) > O(N \log N) > O(N) > O(\log N) > O(1)$

Note that log2N (base 2 logarithm) is often abbreviated as log N

- **Constant time complexity: O(1)**
  An algorithm has constant time complexity if its running time is **independent of the input size**. This means that no matter how large the input is, the running time of the algorithm remains the same.
  For example, **accessing an element in an array** by index is an operation has constant time complexity.

- **Logarithmic time complexity: O(log n)**
  An algorithm has logarithmic time complexity if its running time **grows at a logarithmic rate** with respect to the input size. This means that the running time of the algorithm grows much slower than the input size.
  For example, searching a sorted array using **binary search** has logarithmic time complexity.

- **Linear time complexity: O(n)**
  An algorithm has linear time complexity if its running time is **proportional** to the input size. This means that as the input size grows, the running time of the algorithm grows linearly.
  For example, **iterating through an array** of size n requires n operations, so it has linear time complexity.

- **n log(n) time complexity: O(n log(n))**
  An algorithm has n log(n) time complexity if its running time grows at a rate proportional to n multiplied by the logarithm of n. This means that the running time of the algorithm grows more slowly than n^2 but faster than n.
  For example, many efficient sorting algorithms, such as **quicksort** and **mergesort**, have n log(n) time complexity.

- **Quadratic time complexity: O(n^2)**
  An algorithm has quadratic time complexity if its running time grows at a quadratic rate with respect to the input size. This means that the running time of the algorithm grows much faster than the input size.
  For example, a **nested loop that iterates over an array of size n** has quadratic time complexity.

- **Exponential time complexity: O(2^n)**
  An algorithm has exponential time complexity if its running time grows exponentially with respect to the input size. This means that the running time of the algorithm grows very fast and becomes impractical for large input sizes.
  For example, solving **traveling salesman problem using brute force** has exponential time complexity.

- **Factorial time complexity: O(n!)**
  An algorithm has factorial time complexity if its running time grows at a rate proportional to n factorial, which is n multiplied by every integer smaller than n. This means that the running time of the algorithm grows very quickly with respect to the input size and becomes impractical for all but the smallest inputs.
  For example, a **brute-force algorithm that generates all possible permutations of n elements** has factorial time complexity, as the number of possible permutations is n!.

2.2 Example of Binary search

Binary search, time complexity: O(log n)

Binary search is a more efficient algorithm than linear search for sorted arrays. The algorithm repeatedly divides the search interval in half until the target element is found or the interval is empty.

The time complexity of **binary search** is logarithmic in the size of the input, as the algorithm eliminates half of the remaining search space in each iteration.

c Copy code

```c
#include <stdio.h>

int binary_search(int arr[], int l, int r, int x) {
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x) {
            return mid;
        } else if (arr[mid] < x) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return -1;
}

int main() {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 9;
    int result = binary_search(arr, 0, n - 1, x);
    if (result == -1) {
        printf("Element not present in array\n");
    } else {
        printf("Element found at index %d\n", result);
    }
    return 0;
}
```

In this example, arr is the sorted array that we want to search, n is the length of the array, and x is the element we want to find. The binary_search function takes four arguments: the array, the left index l, the right index r, and the element we want to find x. The function returns the index of the element if it is found in the array, or -1 if it is not present.

2.3 Example of Insertion sort

Insertion sort, time complexity: O(n^2)

Insertion sort is a simple sorting algorithm that builds the final sorted array one element at a time. The algorithm iterates through the input array and inserts each element into its correct position in the sorted array.

The time complexity of **insertion sort** is quadratic in the size of the input, as the algorithm needs to compare each element with each other element on average.

c Copy code

```c
#include <stdio.h>

void insertion_sort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void print_array(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    print_array(arr, n);
    insertion_sort(arr, n);
    printf("Sorted array: ");
    print_array(arr, n);
    return 0;
}
```

In this example, arr is the array that we want to sort and n is the length of the array. The insertion_sort function takes two arguments: the array and its length. The function sorts the array using Insertion Sort algorithm.

3.1 Exercise for proving asymptotic behaviors of a given function

Find the asymptotic upper bound, asymptotic lower bound, and big Theta notation of the function:

i. f(n) = 2n^3 - 5n^2 + 7n
ii. f(n) = 4n^2 + 3n log n
iii. f(n) = 3^n + n^3

3.2 Exercise for calculating time complexity of a given function

Set 1:

i. f(n) = 12
ii. f(n) = 2n + 3
iii. f(n) = 3n^2 + 2n + 1
iv. f(n) = 5 log n + 20
v. f(n) = 2n + 3n log n + 19
vi. f(n) = 6n^3 + 2n^2 + 3n + 4
vii. f(n) = 2^n + n^2+ 1

Set 2:

i.

```
for (int i = 1; i <= n; i++) {
    x++;
}
```

ii.

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        x++;
    }
}
```

iii.

```
for (int i = 1; i <= n; i++) {
    x++;
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        x++;
    }
}
```

Set 3:

i.

```
int i = n * n;
while (i != 1) {
    i = i / 2;
}
```

ii.

```
int x = 0;
int n;

while (n > (x + 1) * (x + 1)) {
    x++;
}
```

iii.

```
int m = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= 2 * i; j++) {
        m++;
    }
}
```

iv.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        a[i][j] = 0;
    }
}
```

Set 4 (Shown with python code):

i.

```
def sum_of_squares(n):
    sum = 0
    for i in range(n):
        sum += i * i
    return sum
```

ii.

```
def is_palindrome(s):
    for i in range(len(s) // 2):
        if s[i] != s[-i - 1]:
            return False
    return True
```

Note: In Python, the "//" symbol is the integer division operator. This operator is used to calculate the integer part of the result of dividing two integers and return the integer value. For example, the result of 5//2 is 2.

iii.

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

Note: In Python, the double asterisk "**" is an operator that is used to represent exponentiation, i.e., raising a number to a power. For example, 2 ** 3 would return 8, because it means "2 raised to the power of 3" which is equal to 2 * 2 * 2 = 8.

iv.

```
def contains_duplicates(arr):
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j]:
                return True
    return False
```

v.

```
def count_pairs(arr, k):
    count = 0
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] + arr[j] == k:
                count += 1
    return count
```