* **Asymptotic analysis**:

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don’t depend on machine-specific constants and don’t require algorithms to be implemented and time taken by programs to be compared.

* **Asymptotic notations:**

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

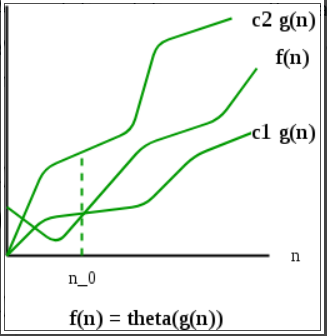
* The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

1. **Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. **It is used to determine the average case.**

**Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that**

**n0 <= c1\*g(n) <= f(n) <= c2\*g(n) for all n >= n0}.**

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1\*g(n) and c2\*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

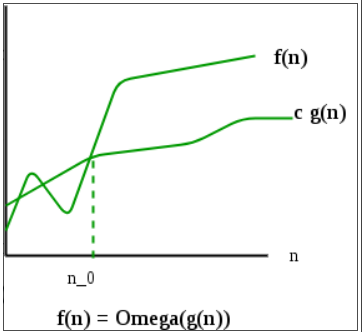


***Note: Theta provides exact bound***

1. **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. **It is used to determine the worst case.**

**O(g(n)) = { f(n): there exist positive constants c and n0 such that**

**n0 <= f(n) <= c\*g(n) for all n >= n0}**

****

***Note: Big O provides exact or upper bound***

1. **Ω Notation:** Ω notation provides an asymptotic lower bound. **It is used to determine the best case.**

**Ω (g(n)) = {f(n): there exist positive constants c and n0 such that**

**n0 <= c\*g(n) <= f(n) for all n >= n0}**

***Note: Omega provides lower bound. Omega notation is the least used notation among all three.***

P.S.:

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Result** | **Description** |
| Best case | Ω (1) |  |
| Average case | Θ (n+1/2) |  |
| Worst case | O (n) | Max time which our code might take |

**Order of time complexity:** O(nn) > O(n!) > O(n3) > O(n2) > O(n.log(n)) > O(n.log(log(n))) > O(n) > O(sqrt(n)) > O(log(n)) > O(1)

* **Stack and Heap:**
* In Java, to store primitives (system configured) and non-primitives (user-defined), there are two memories: Stack and Heap
* Heap size is greater Stack
* In Heap, we create the actual non-primitive
* In Stack, we store the address of the non-primitives, created in the Heap.
* **Equals vs == :**
* == compares the logical addresses of the objects.
* .equals compares the actual content of the objects.
* For non-primitives, it’s advisable to use .equals.
* **Strings:**
* In Java, strings are treated as objects.
* In Heap, there’s a storage area for Strings- **String Pool**
* If there’s a new request for a string, to optimize the storage, Java looks in String pool for a String having same content and returns its reference rather than creating a new String.
* Even though reference of an existing string is shared, the changes in one doesn’t reflect another because as and when we’re modifying one string, we are actually creating a new String, which will eventually create a new String in String pool.
* Strings are immutable: Once created, we can’t change the content of it. Any change in the string will eventually create a new String in String pool.
* If a variable is containing “Hello” value, and another variable holds the same value, java will provide the reference of former variable to the latter, if new keyword is not used for initializing 2nd variable.
* **Substring (startIndex** (inclusive)**, endIndex** (exclusive)**).**
* **Bit Manipulation:**
* Position**:** It’s read from right to left, with rightmost bit is at 0 index and index increases subsequently.
* **GetBit**: Used to find the value of the bit at a position in a given number

Here we perform two operations:

1. Bit Mask i.e., Left shift the bits of 1(in binary) by the bits given in position. E.g., we need to Getbit for position 2, we’ll left shift 1 by 2 bits.
2. Perform logical AND of the number with the Masked number(result of a)
3. If the result of b. is a non-zero number, result of GetBit will be 1 else 0

* **SetBit**: Used to set the value of a bit to 1, at a position in a given number

Here we perform two operations:

* 1. Bit Mask i.e., Left shift the bits of 1(in binary) by the bits given in position. E.g., we need to SetBit for position 2, we’ll left shift 1 by 2 bits.
  2. Perform logical OR of the number with the Masked number(result of a)
  3. The result of b is the required result of SetBit
* **ClearBit**: Used to reset the value of a bit at a position in a given number to zero

Here we perform following operations:

1. Bit Mask i.e., Left shift the bits of 1(in binary) by the bits given in position. E.g., we need to ClearBit for position 2, we’ll left shift 1 by 2 bits.
2. Perform logical NOT on the BitMask, result of a.
3. Perform logical AND of the number with the ~(BitMask, result of b)
4. The result of c is the required result of ClearBit

* **UpdateBit**: Used to set the value of a bit at a position in a given number to zero/one

Here we perform the operation, depending upon what should be the updated value:

* 1. Bit Mask i.e., Left shift the bits of 1(in binary) by the bits given in position. E.g., we need to ClearBit for position 2, we’ll left shift 1 by 2 bits.
  2. If we need to set a bit to 0: Perform ClearBit operation
  3. If we need to set a bit to 1: Perform SetBit operation
* **Sorting:**

1. **Bubble Sort:**

**Principle:**

Here, we bring largest element to the end of the array.

In every iteration, we compare each element with its next and swap if current element is greater than the next.

We do so until we find there are no swaps performed or end of array is encountered.

1. **Selection Sort:**

**Principle:**

Here, we bring the smallest element to the starting of the array.

In every iteration, we consider 1st element to be smallest, and store its index thereby comparing it with every other element. If an element smaller than the smallest is found, we update the smallest index and at the end of iteration place it at starting.

1. **Insertion Sort:**

**Principle:**

Here, we divide the array into two parts, sorted and unsorted.

In every iteration, we take a part of array and considering it to be sorted. We compare 1st element of unsorted array and compare it with every element of sorted array starting from end. Then, if this element is smaller than any of them, we create a gap for this element by shifting all other elements of sorted part (larger elements than this) to the right. Then, like this every element from unsorted part is placed at its appropriate position in sorted part.

1. **Quick Sort:**

**Principle:**

Since we can break up the original array into equal subarrays log N times, and for each time we break them up, we must run partitions on all N cells from the original array, we end up with about N \* log N steps.

When it comes to Quicksort*,*

1. *Best-case scenario:*

When the pivot always ends up smack in the middle of the subarray after the partition.

1. *Worst-case scenario:*

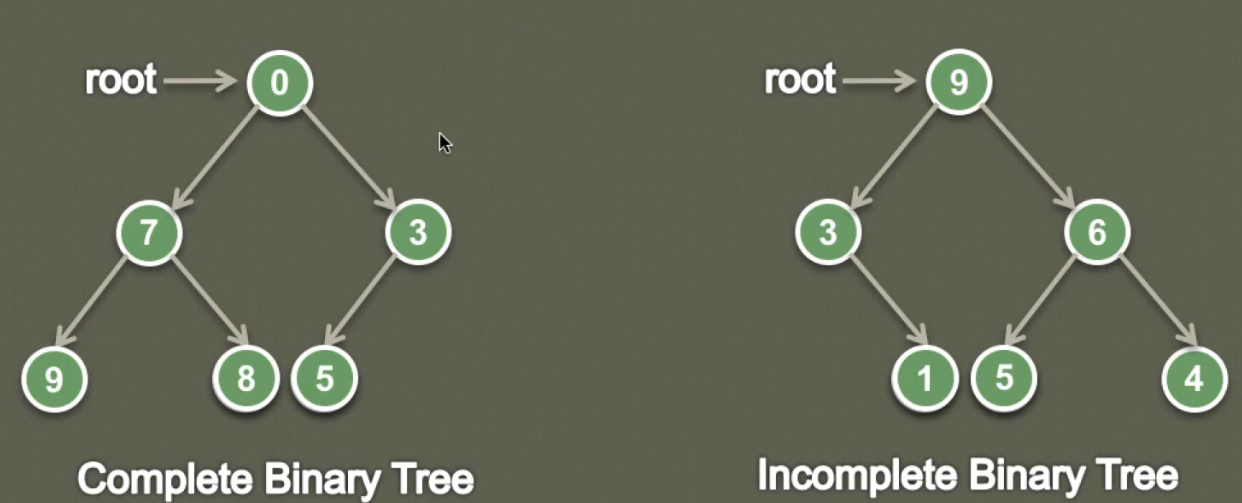
When the pivot always ends up on one side of the subarray instead of the middle. This can happen in several cases, including where the array is in perfect ascending or descending order

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Best case** | **Average case** | **Worst case** |
| **Insertion sort** | O(N) | O(N2) | O(N2) |
| **Quick sort** | O(N log N) | O(N log N) | O(N2) |

* **Complete Binary Tree:**

It’s a binary tree with the following properties:

1. All levels are complete except the last one
2. The last level is filled in a manner such that left child is never empty.



* **Binary Heap:**

It’s a complete binary tree.

It’s a DS that helps us implementing Priority Queue operations efficiently.

1. Min Heap

Each node has value less than or equal to its child node.

1. Max Heap

Each node has value greater than or equal to its child node.

* **Implementation:**

1. Implemented using array, having 0th element as null.
2. Values are stored in array by traversing the tree level by level from left to right.
3. In array, 1th element is the Max and last element is min.
4. **Detecting child by the parent’s index**: If a parent node is at kth level, it’ll have children as: 2\*k and 2\*k+1.
5. **Detecting parent by the child’s index**: If a child node is at kth level, it’ll have parent at: k/2th level (for odd k, round off the result).