Group 11

K Shreyas
Rajneesh Pandey
Satyarth Pandey

Asymptotic time and space complexity

- Compare 2 given sorting algorithms **=>** Compare raw timings? : bad idea because maybe different input/hardware/bad implementation/luck.
- The time/space complexity of an algorithm quantifies the amount of time/space taken by an algorithm to run as a function of the length of the input.
- Note that the complexity is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

Best, Worst, and Average Case Complexity

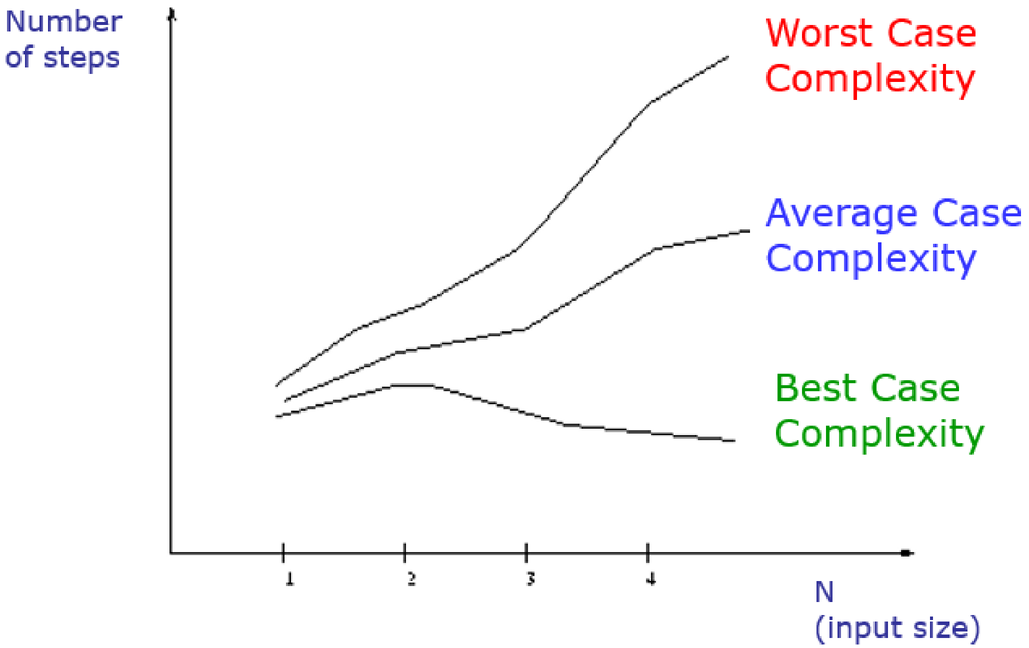
Suppose you are given a list L of some length len(L). Search an element E in this list.

- Worst Case Complexity:
  - The function defined by the *maximum* number of steps taken on any instance of size *n*.
  - Linear in length of list.

- Best Case Complexity:
  - The function defined by the *minimum* number of steps taken on any instance of size *n*.
  - Constant Time

- Average Case Complexity:
  - The function defined by the *average* number of steps taken on any instance of size *n*.
  - Linear in length of list.

Best, Worst, and Average Case Complexity



- **Goal**: to simplify analysis by getting rid of unneeded information (like “rounding” $1{,}000{,}001 \approx 1{,}000{,}000$)

- We want to say in a formal way $3n^2 \approx n^2$

Big-O Notation

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

- What does it mean!? :/

→ If $f(n) = O(n^2)$, then:

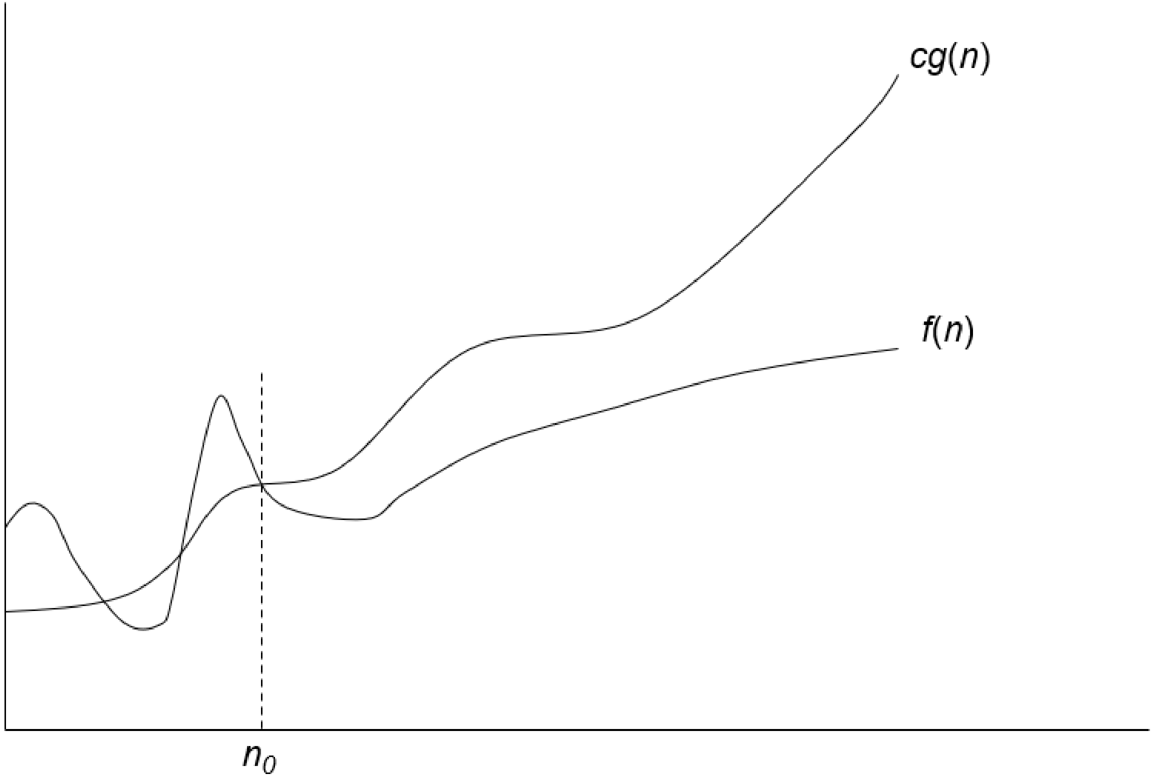
→ $f(n)$ can be larger than n^2 sometimes, **but...**

→ We can choose some constant **c** and some value n_0 such that for **every** value of $\boldsymbol{n}$ larger than $\boldsymbol{n_0}$: $f(n) < cn^2$

→ That is, for values larger than n_0 , $f(n)$ is never more than a constant multiplier greater than n^2

→ Or, in other words, $f(n)$ does not grow more than a constant factor faster than n^2.

Visualization of $O(g(n))$



Examples

i) $2n^2 = O(n^3)$:

$2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$ and $n_0 = 2$

ii) $1000n^2 + 1000n = O(n^2)$:

$1000n^2 + 1000n \leq cn^2 \leq cn^2 + 1000n \Rightarrow c = 1001$ and $n_0 = 1$

- Note: Even though it is correct to say "$7n - 3$ is $O(n^3)$", a better statement is "$7n - 3$ is $O(n)$", that is, one should make the approximation as tight as possible

- Simple Rule: Drop lower order terms and constant factors

  $7n-3$ is $O(n)$

  $8n^2\log n + 5n^2 + n$ is $O(n^2\log n)$

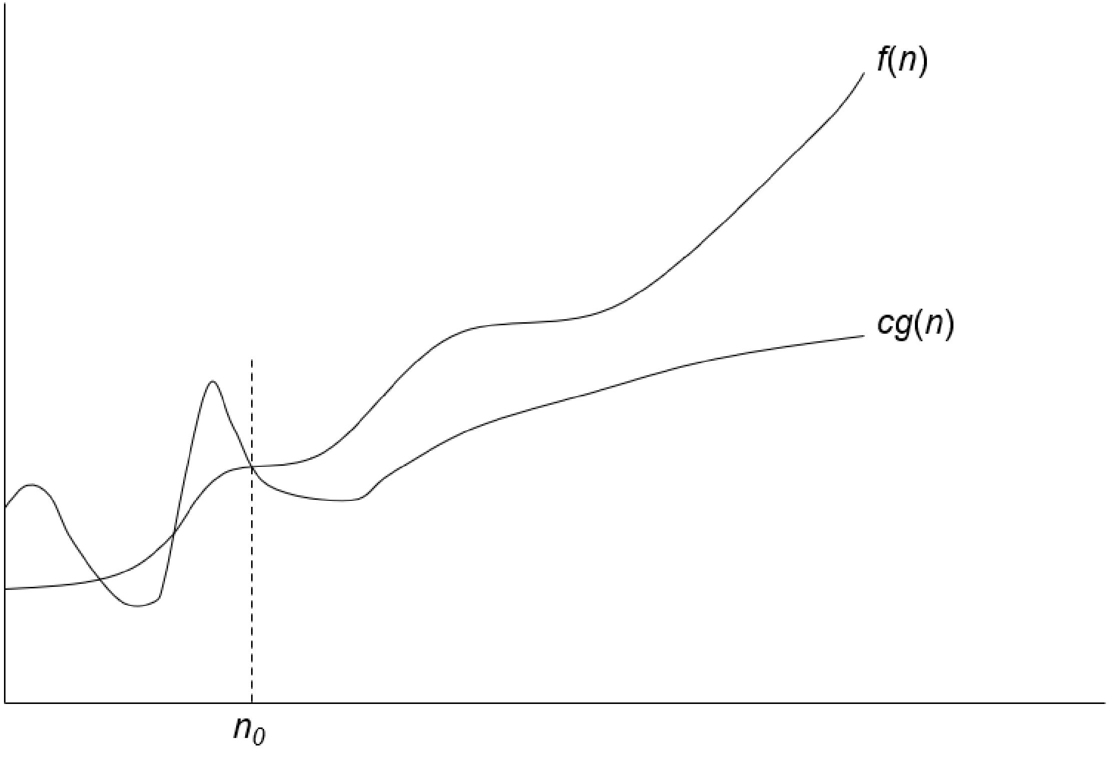
Big Omega Notation

$f(n) = \Omega(g(n))$: there exist positive constants c and n_0 such that

$$0 \le f(n) \ge cg(n) \text{ for all } n \ge n_0$$

- $\Omega()$ – A **lower** bound
- $N^2 = \Omega(N)$
- Let $c = 1$, $N_0 = 2$
- For all $N >= 2$, $N^2 > 1 \times N$

Visualization of $\Omega(g(n))$



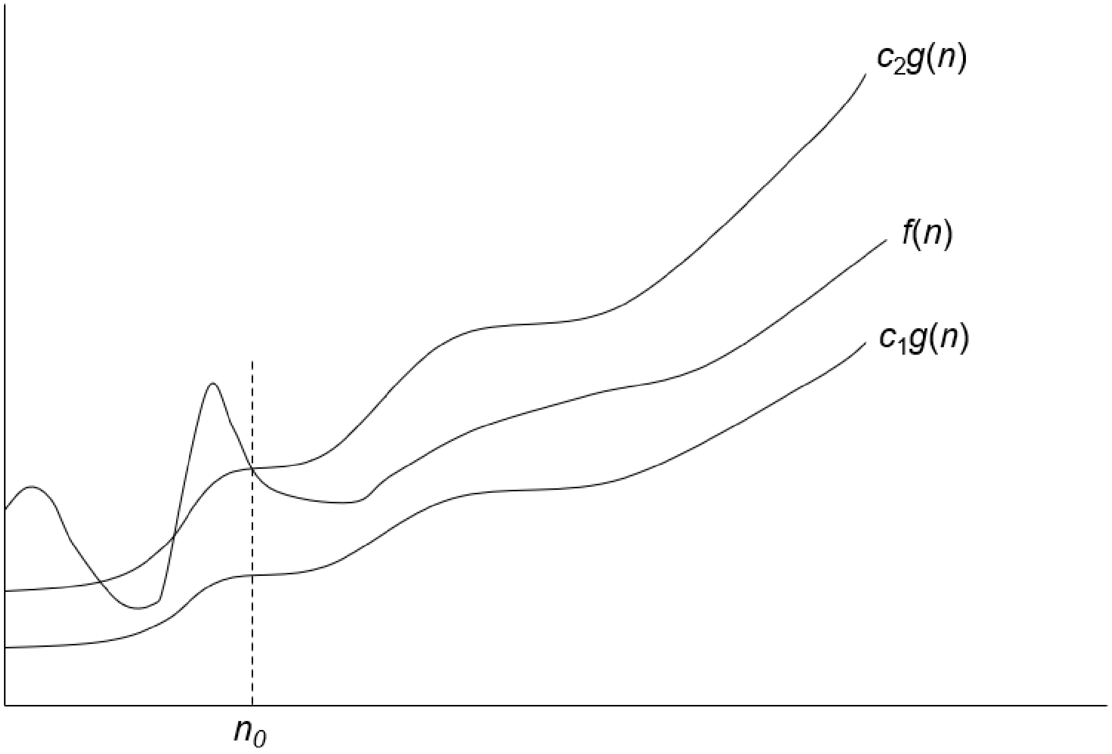
Theta - notation

$$f(n)=\Theta(g(n)): \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

- Big-*O* is not a tight upper bound. In other words $n = O(n^2)$
- Θ provides a tight bound
- In other words,

$$f(n)=\Theta(g(n)) \Rightarrow f(n)=O(g(n)) \text{ AND } f(n)=\Omega(g(n))$$

Visualization of $\Theta(g(n))$

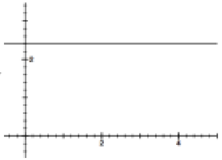


Few Examples

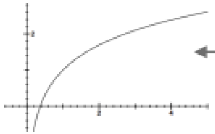
- $n = O(n^2) \neq \Theta(n^2)$
- $200n^2 = O(n^2) = \Theta(n^2)$
- $n^{2.5} \neq O(n^2) \neq \Theta(n^2)$

Popular Complexity Classes (order: low to high)

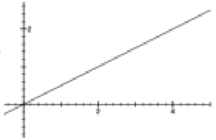
$O(1)$: constant



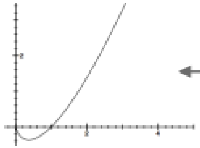
$O(\log n)$: logarithmic



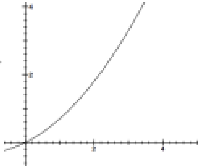
$O(n)$: linear



$O(n \log n)$: loglinear

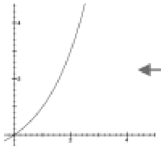


$O(n^c)$: polynomial



c is a constant

$O(c^n)$: exponential



GATE CSE 2006

Consider the following C-program fragment in which i, j and n are integer variables.

```
for (i = n, j = 0; i > 0; i /= 2, j += i);
```

let val (j) denote the value stored in the variable j after termination of the for loop. Which one of the following is true?

A. val (j) = θ (log n)

B. val (j) = $(\sqrt{n})$

C. val (j) = θ (n)

D. val (j) = θ (n log n)

GATE CSE 2006

Consider the following C-program fragment in which i, j and n are integer variables.

for (i = n, j = 0; i > 0; i /= 2, j += i);

let val (j) denote the value stored in the variable j after termination of the for loop. Which one of the following is true?

A val (j) = θ (log n)

B val (j) = $(\sqrt{n})$

C val (j) = θ (n) Correct Answer

D val (j) = θ (n log n)

Explanation:

n + n/2 + n/4 + n/8 + n/16 ... <= 2 * n

Geometric Progression! :)

GATE CSE 2015 Set 3

Consider the equality $\sum_{i=0}^{n} i^3 = X$ and the following choices for X

I. $\Theta\left(n^4\right)$

II. $\Theta\left(n^5\right)$

III. $O\left(n^5\right)$

IV. $\Omega\left(n^3\right)$

The equality above remains correct if X is replaced by

A. Only I

B. Only II

C. I or III or IV but not II

D. II or III or IV but not I

GATE CSE 2015 Set 3

Consider the equality $\sum_{i=0}^{n} i^3 = X$ and the following choices for X

I. $\Theta(n^4)$

II. $\Theta(n^5)$

III. $O(n^5)$

IV. $\Omega(n^3)$

The equality above remains correct if X is replaced by

A. Only I

B. Only II

C. I or III or IV but not II — Correct Answer

D. II or III or IV but not I

$$\sum_{i=1}^{n} i^3 = \left[\frac{n(n+1)}{2}\right]^2$$

Above we have a **$\Theta(N^4)$ Complexity**, which also further implies option III and IV

Recurrences

- A recurrence is an equation that describes a function in terms of its value on smaller inputs.

- For example: $T(n) = T(n-1) + T(n-2)$

Solving Recurrences

There are **3 methods** to solve these recurrences and obtain a running time of an algorithm in terms of Big-Oh (O) notation:

- **Substitution**: Guess a solution, verify by induction.
- **Recursion Tree**: Draw a tree representing the recurrence and sum computation at nodes.
- **Master Theorem**: A general formula to solve a large class of recurrences. More Reliable.

Master Theorem

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

- It's still really hard to tell what the Big-Θ is just by looking at it.
- But fancy mathematicians have a formula for us to use!

MASTER THEOREM

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If	$\log_b a < c$	then	$T(n) \in \Theta(n^c)$
If	$\log_b a = c$	then	$T(n) \in \Theta(n^c \log n)$
If	$\log_b a > c$	then	$T(n) \in \Theta\left(n^{\log_b a}\right)$

a=2 b=3 and c=1

$y = \log_b x$ *is equal to* $b^y = x$

$\log_3 2 \cong 0.63$

$\log_3 2 < 1$

We're in case 1

$T(n) \in \Theta(n)$

GATE CSE 2017 Set 2

Consider the recurrence function

$$T(n) = \begin{cases} 2T(\sqrt{n}) + 1, & n > 2 \\ 2, & 0 < n \end{cases}$$

Then T(N) in terms of θ notation is:

(A) θ (log log n)

(B) θ (log n)

(C) θ (sqrt(n))

(D) θ (n)

GATE CSE 2017 Set 2

Consider the recurrence function

$$T(n) = \begin{cases} 2T(\sqrt{n}) + 1, & n > 2 \\ 2, & 0 < n \end{cases}$$

Then T(N) in terms of θ notation is:

(A) θ (log log n)

(B) θ (log n) **Correct Answer**

(C) θ (sqrt(n))

(D) θ (n)

Explanation:

$T(n) = 2T(\sqrt{n}) + 1$

Let $n = 2^m$

$\implies T(2^m) = 2T(2^{m/2}) + 1$

Let $S(m) = T(2^m)$

$\implies S(m/2) = T(2^{m/2})$

Thus above equation will be :

$S(m) = 2S(m/2)+1$

Applying master's theorem $S(m) = \Theta(m)$

Thus : $T(n) = \Theta(\log(n))$ (since $n = 2^m$)

Searching, Sorting and Divide and Conquer Algorithms

Contents

Searching Algorithms:

Divide & Conquer Algorithms:

Sorting Algorithms:

Searching Algorithms

01. Linear (Sequential) Search:

It finds an item in an array by looking for it from the beginning till the end. Without any assumption about the array.

Worst case Analysis of Sequential Search:

Occurs when the search term is the last element in the array or not present.

The number of comparisons in worst case = size of the array = **n**.

Time complexity = **O(n)**

02. Binary search

Assumes the <u>list is already in order</u>.

In each step

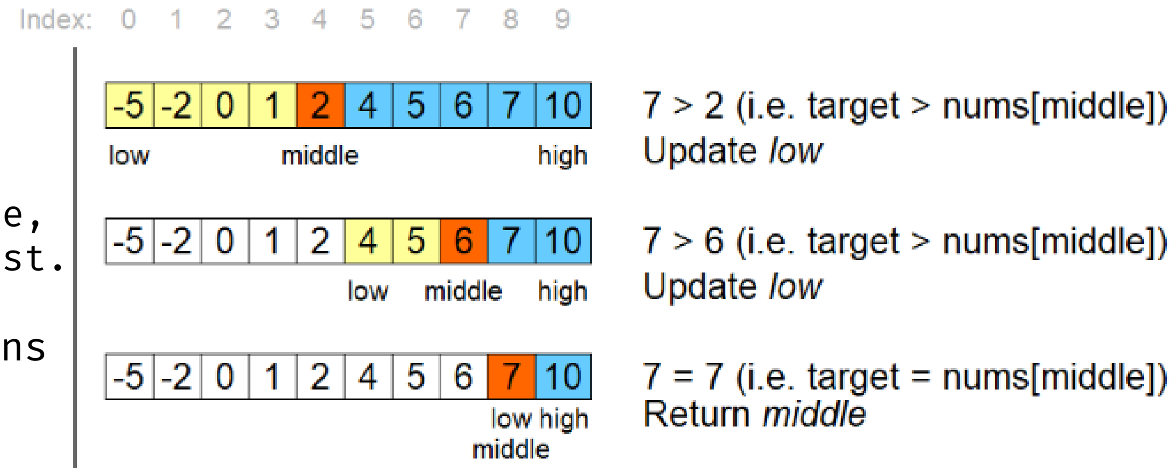
- Finds the Middle element
- Considers only left side or right side elements according to condition.

Worst-case analysis

When search term is
-not in the list, or
-one item away from the middle,
-first or last item in the list.

Maximum number of comparisons
(log(n))+1

Time complexity = **O(log(n))**



Sorting Algorithms

Types of sorting:

Based on Complexity, Stability, comparison, etc.

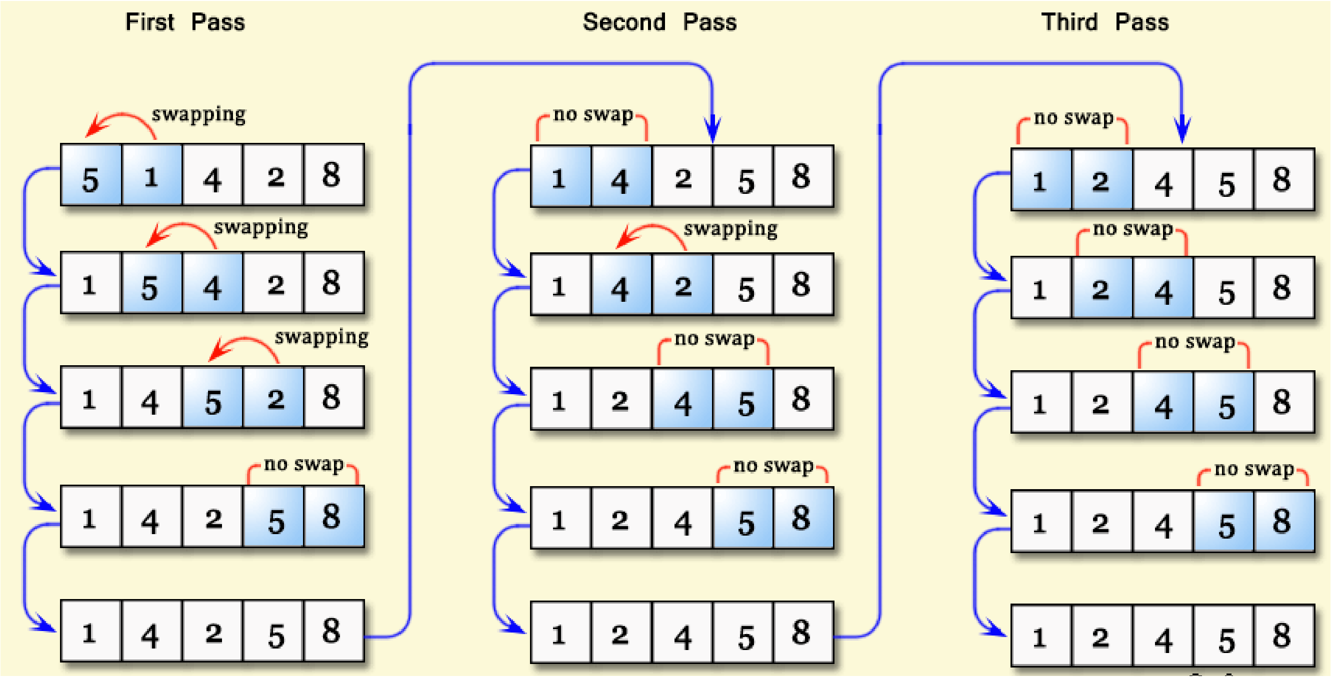
- **In-place Sorting:**
  The In-place sorting algorithm <u>does not use extra storage</u> to sort the elements in the given list.

- **Stable Sorting:**
  Stable sorting algorithm <u>maintain the relative order</u> of records with equal values during sorting the elements.

- **Comparison based sorting:**
  It determines that ,which of <u>two elements being compared</u> should occur first in the final sorted list.

Bubble Sort

Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order until the list get sorted.
It can be optimized by stopping the algorithm if the inner loop didn't cause any swap



Time Complexity:

Best case: **O(n)**
Average case: **O(n^2)**
Worst case : **O(n^2)**

Last **i** elements will be sorted in every **i** Pass.

Insertion Sort

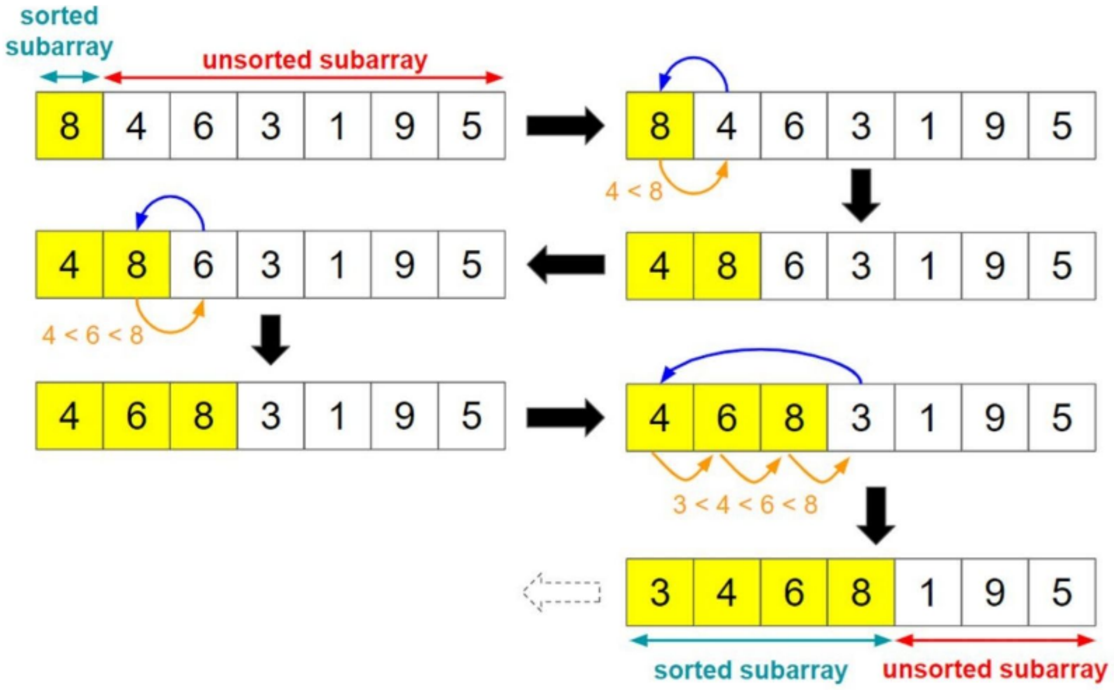
Iterates through an array and repeatedly inserts each element into its sorted position within a growing sorted subarray.

Time Complexity:

Best case: **O(n)** (sorted)
Average case: **O(n^2)**
Worst case : **O(n^2)**
(Reverse sorted)

Useful only for <u>small</u> files or very <u>nearly</u> <u>sorted</u> files.

It is <u>comparison</u> based, <u>in-place</u> and <u>stable</u> algorithm.



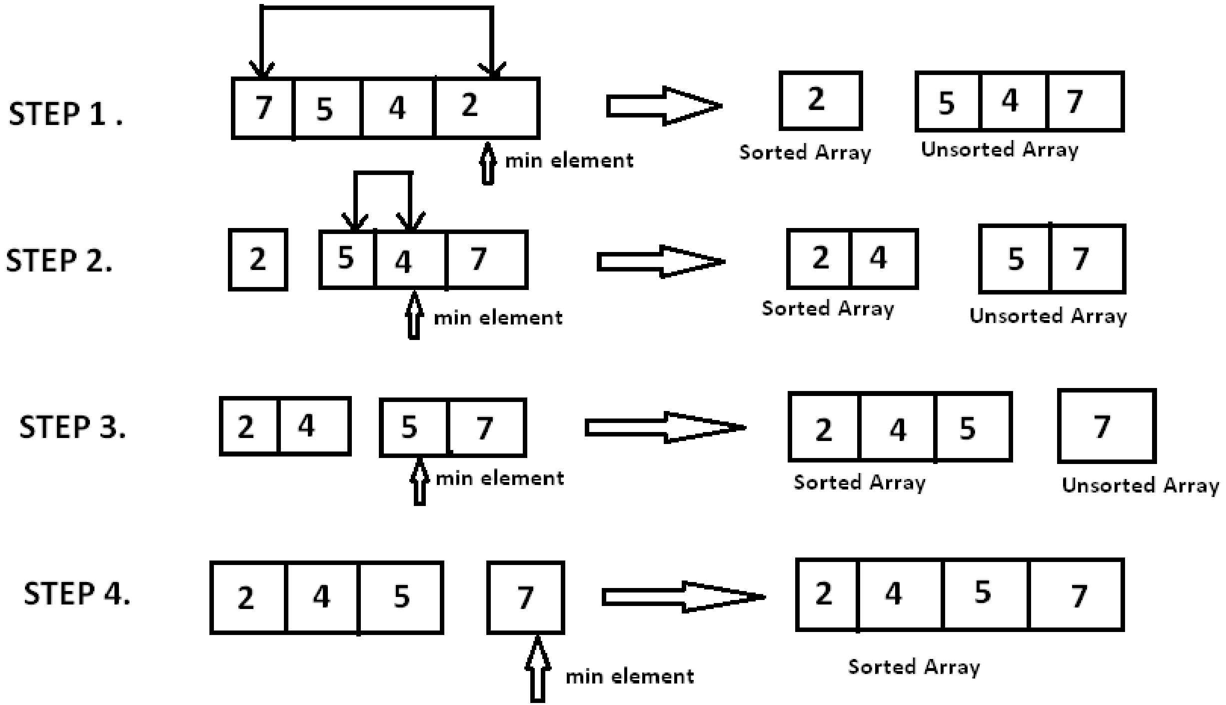
Selection Sort

Repeatedly selects the minimum element from an unsorted list and places it at the beginning of the list until the whole list is sorted.

Time Complexity:

Best case: **O(n^2)**
Average case: **O(n^2)**
Worst case : **O(n^2)**

It <u>in-place,</u> <u>comparison</u> sorting algorithms



Heap Sort

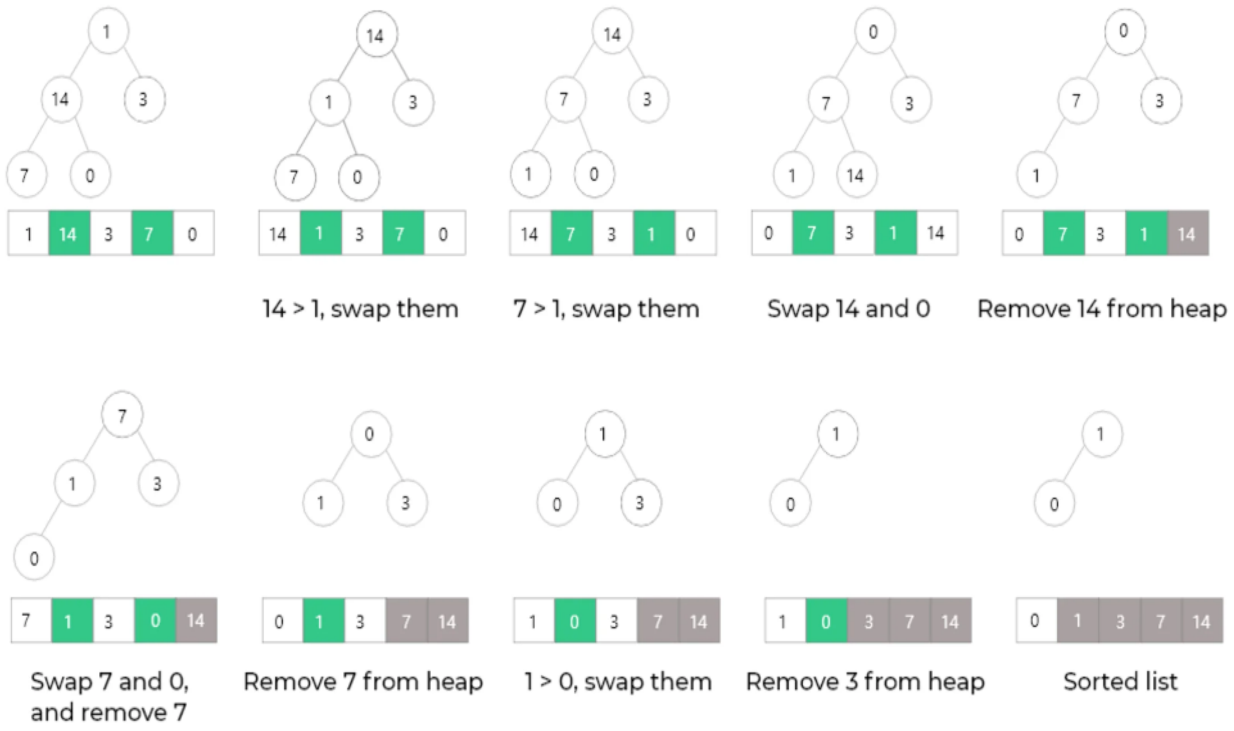
Comparison-based sorting algorithm that uses a binary heap data structure to sort elements in ascending or descending order

Max heap: parent>children.

Time Complexity: **O(nlogn)**

Build-Heap: O(n) time.
Heapify: O(logn) time.
Top to bottom
Extract-Max: O(logn) time.

It is comparison based, in-place and non-stable algorithm.



Merge Sort

It's a divide-and-conquer algorithm that recursively divides a list into two halves, sorts them independently, and then merges them back together into a sorted list.

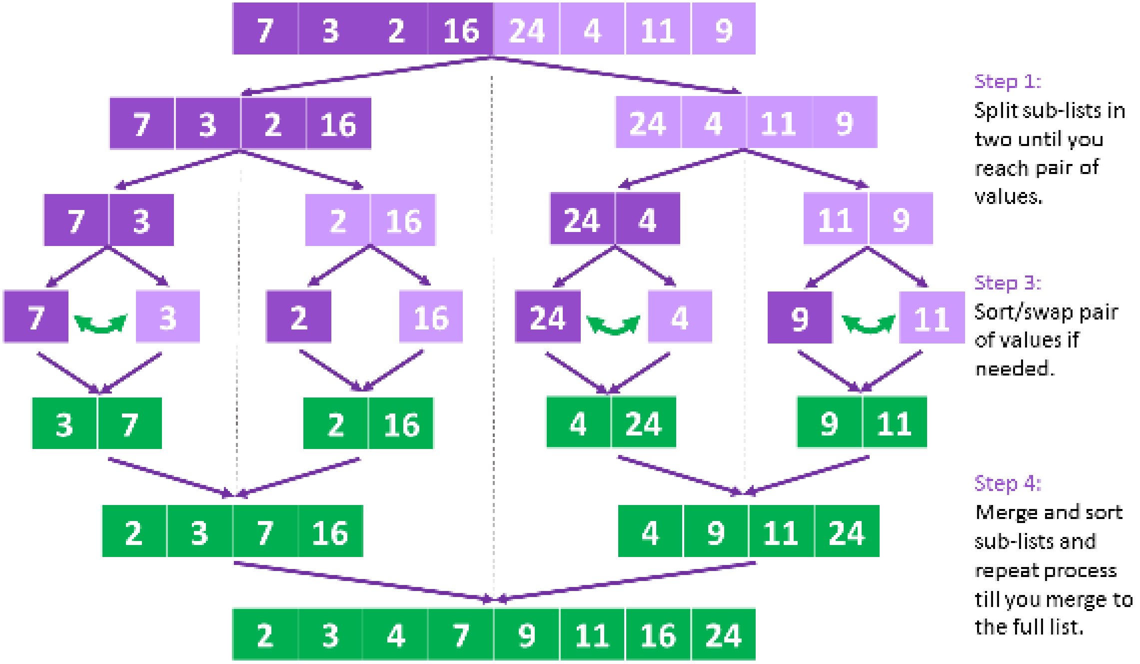
Time Complexity:

$T(n) = 2T(n/2) + O(n)$

Best Case: **O(nlogn)**
Average case: **O(nlogn)**
Worst Case: **O(nlogn)**

It is <u>comparison</u> based, and <u>stable</u> algorithm.

Application:
Count Inversion in array



Quick Sort (Partition exchange sort.)

Comparison-based algo. that uses a divide-and-conquer approach by selecting a pivot element and partitioning the array into smaller sub-arrays.

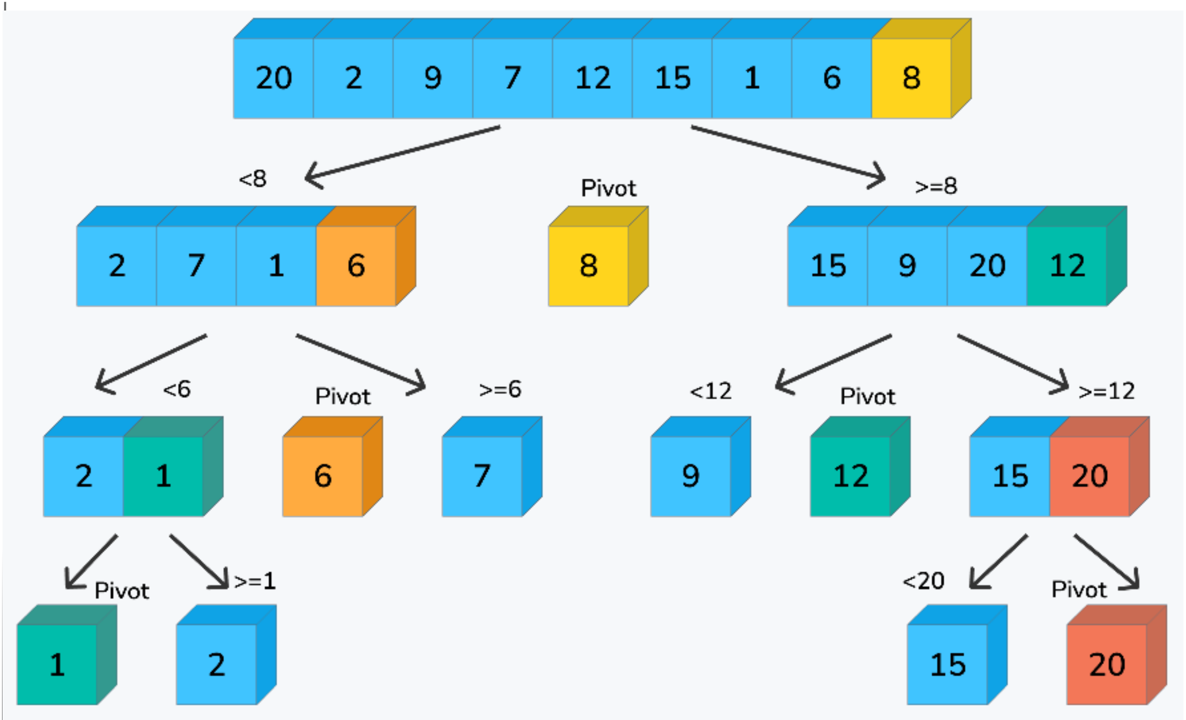
Time Complexity:

T(n) = T(n-1)+T(1)+cn

Best-Average case: **O(nlogn)**
Worst case : **O(n^2)**
(sorted/Reverse-sorted/all equal)

It is <u>in-place</u> sorting algo

Randomized quick sort is a variant of this that selects a random pivot element to avoid worst-case scenarios and improve average-case performance.



Problems

Q1. In a binary max heap containing n numbers, the smallest element can be found in time?

A. O(n)
B. O(logn)
C. O(1)
D. O(log(logn))

Correct ans: **A**

In max heap, the smallest element should not have any children so it will be present in the leaf node. And to find the smallest element you have to go to the leaf and find minimum among them.
In a heap leaf node starts from **floor(n/2)+1** and end at **n**. So no of leaf nodes present in the heap is **O(n/2)**. So total no of comparison needed in order to find the smallest element is n and the time taken for that will be **O(n)**.

Q2. There are two sorted list each of length n. An element to be searched in the both the lists. The lists are mutually exclusive. The maximum number of comparisons required using binary search and find its time complexity?

A. O(n)
B. O(logn)
C. O(n*n)
D. O(log(logn))

Correct ans: **B**
Suppose searching element=X

First apply the BINARY search in array 1, if element found then well and good, TC will be O(logn). But, If element not found in array 1,then apply the Binary search in array2,again in will take logn time ,but Now,overall time must be 2logn but asymptotically the TC WILL BE = o(logn)

Q.3. An element in an array X is called a leader if it is greater than all elements to the right of it in X. The best algorithm to find all leaders in an array.

A. Solves it in linear time using a left to right pass of the array
B. Solves it in linear time using a right to left pass of the array
C. Solves it using divide and conquer in time Theta(nlogn)
D. Solves it in time Theta(n^2)

Correct ans: **B**

Let max = arr[n-1], then
Traverse from right to left and add arr[i] to Leader if (arr[i]>max) and update max=arr[i]

Q.4 Give the correct matching for the following pairs:

Group - 1	Group - 2
A. O(logn)	(P) Selection
B. O(n)	(Q) Insertion sort
C. O(nlogn)	(R) Binary search
D. O(n*n)	(S) Merge sort

Correct Ans: **A – R , B – Q, C – S, D – P**

Best case time complexity
Selection sort : O(n*n)
Merge sort : O(nlogn)
Binary search : O(logn)
Insertion sort : O(n)

Hashing

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. Use of a hash function to index a hash table is called hashing.

A good hash function should have following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys
3. Should minimize collisions.
4. Should have a low load factor.

Load factor = (number of keys) / (number of slots in the table)

Collision Resolution by Separate Chaining

If multiple keys index to the same slot using a hash function, maintain a data structure at each slot of the table to contain multiple elements, such as a linked list, dynamic array or a balanced BST.

Collision Resolution by Open Addressing

- **Linear Probing:** If a key hashes to an occupied slot in the table, put it in the next available slot by traversing through the table indices **in steps of 1 ($n + 1$, $n + 2$, $n + 3$, etc)**.
- **Quadratic Probing:** If a key hashes to an occupied slot in the table, put it in the next available slot by traversing through the table indices **in steps of square numbers ($n + 1$, $n + 4$, $n + 9$, etc)**.
- **Double Hashing:** If a key hashes to an occupied slot in the table, put it in the next available slot by traversing through the table indices **in steps of *a secondary hash function* ($n + h_2(x)$, $n + 2h_2(x)$, $n + 3h_2(x)$, etc)**.

GATE Questions

1. Suppose we are given n keys, m has table slots, and two simple uniform hash functions h1 and h2. Further suppose our hashing scheme uses h1 for the odd keys and h2 for the even keys. What is the expected number of keys in a slot?

GATE Questions

1. Suppose we are given n keys, m has table slots, and two simple uniform hash functions h1 and h2. Further suppose our hashing scheme uses h1 for the odd keys and h2 for the even keys. What is the expected number of keys in a slot?

Ans: **n / m**

GATE Questions

2. Consider a double hashing scheme in which the primary hash function is h1(k)= k mod 23, and the secondary hash function is h2(k) = 1+(k mod 19). Assume that the table size is 23. Then the address returned by probe 1 in the probe sequence (assume that the probe sequence begins at probe 0) for key value k=90 is ______.

GATE Questions

2. Consider a double hashing scheme in which the primary hash function is h1(k)= k mod 23, and the secondary hash function is h2(k) = 1+(k mod 19). Assume that the table size is 23. Then the address returned by probe 1 in the probe sequence (assume that the probe sequence begins at probe 0) for key value k=90 is ______.

Ans:

h1(90) = 90 mod 23 = 21. We assume that 21 is occupied, hence probe 1 is needed.

h2(90) = 1 + (90 mod 19)= 15.

So the next probe location would be at (21 + 15) mod 23 = 36 mod 23 = **13**.

GATE Questions

3. A hash table contains 10 buckets and uses linear probing to resolve collisions. The key values are integers and the hash function used is key % 10. If the values 43, 165, 62, 123, 142 are inserted in the table, in what location would the key value 142 be inserted?

GATE Questions

3. A hash table contains 10 buckets and uses linear probing to resolve collisions. The key values are integers and the hash function used is key % 10. If the values 43, 165, 62, 123, 142 are inserted in the table, in what location would the key value 142 be inserted?

Ans:

43 ->3; 165 -> 5; 62 -> 2; 123 -> ~~3~~ -> 4

142 -> ~~2 -> 3 -> 4 -> 5 ->~~ **6**