Tribhuvan University

Institute of Science and Technology

2080

Bachelor Level / Fifth Semester / Science

Computer Science and Information Technology (CSC314)

Design and Analysis of Algorithms

Full Marks: 60 + 20 + 20 Pass Marks: 24 + 8 + 8 Time: 3 Hours

Candidates are required to give their answers in their own words as far as practicable.

The figures in the margin indicate full marks.

Section A
Attempt any two questions.

What is a recurrence relation? How can it be solved? Show that the time complexity of the recurrence relation T(n) = 2T(n/2) + 1 is O(n) using the substitution method. (10)

Recurrence relation - Wikipedia

→ **If an algorithm is designed so that it will break a problem into smaller subproblems (divide and conquer), its running time is described by a recurrence relation.**

A simple example is the time an algorithm takes to find an element in an ordered vector with n elements, in the worst case.

A naive algorithm will search from left to right, one element at a time. The worst possible scenario is when the required element is the last, so the number of comparisons is n.

A better algorithm is called binary search. However, it requires a sorted vector. It will first check if the element is at the middle of the vector. If not, then it will check if the middle element is greater or lesser than the sought element. At this point, half of the vector can be discarded, and the algorithm can be run again on the other half. The number of comparisons will be given by

$c1=1$

$cn=1+cn/2$

the time complexity of which will be

$O(\log 2(n))$

In mathematics, a recurrence relation is an equation according to which the nth term of a sequence of numbers is equal to some combination of the previous terms. Often, only k previous terms of the sequence appear in the equation, for a parameter k that is independent of n; this number k is called the order of the relation. If the values of the first k numbers in the sequence have been given, the rest of the sequence can be calculated by repeatedly applying the equation.

In linear recurrences, the nth term is equated to a linear function of the k previous terms. A famous example is the recurrence of the Fibonacci numbers,

Fn = Fn-1 + Fn-2

where the order k is two and the linear function merely adds the two previous terms. This example is a linear recurrence with constant coefficients, because the coefficients of the linear function (1 and 1) are constants that do not depend on n. For these recurrences, one can express the general term of the sequence as a closed-form expression of n. As well, linear recurrences with polynomial coefficients depending on n are also important, because many common elementary and special functions have a Taylor series whose coefficients satisfy such a recurrence relation (see holonomic function).Solving a recurrence relation means obtaining a closed-form solution: a non-recursive function of n.

The concept of a recurrence relation can be extended to multidimensional arrays, that is, indexed families that are indexed by tuples of natural numbers.

Factorial

The factorial is defined by the recurrence relation

$n!=n(n-1)!\text{for}n>0,$

and the initial condition

$0!=1.$

This is an example of a linear recurrence with polynomial coefficients of order 1, with the simple polynomial

$f(n)=n$

as its only coefficient.

Fibonacci numbers

The recurrence of order two satisfied by the Fibonacci numbers is the canonical example of a homogeneous linear recurrence relation with constant coefficients (see below). The Fibonacci sequence is defined using the recurrence

$Fn=Fn-1+Fn-2$

with initial conditions

$F0=0$

$F1=1.$

Explicitly, the recurrence yields the equations

$F2=F1+F0$

$F3=F2+F1$

$F4=F3+F2$

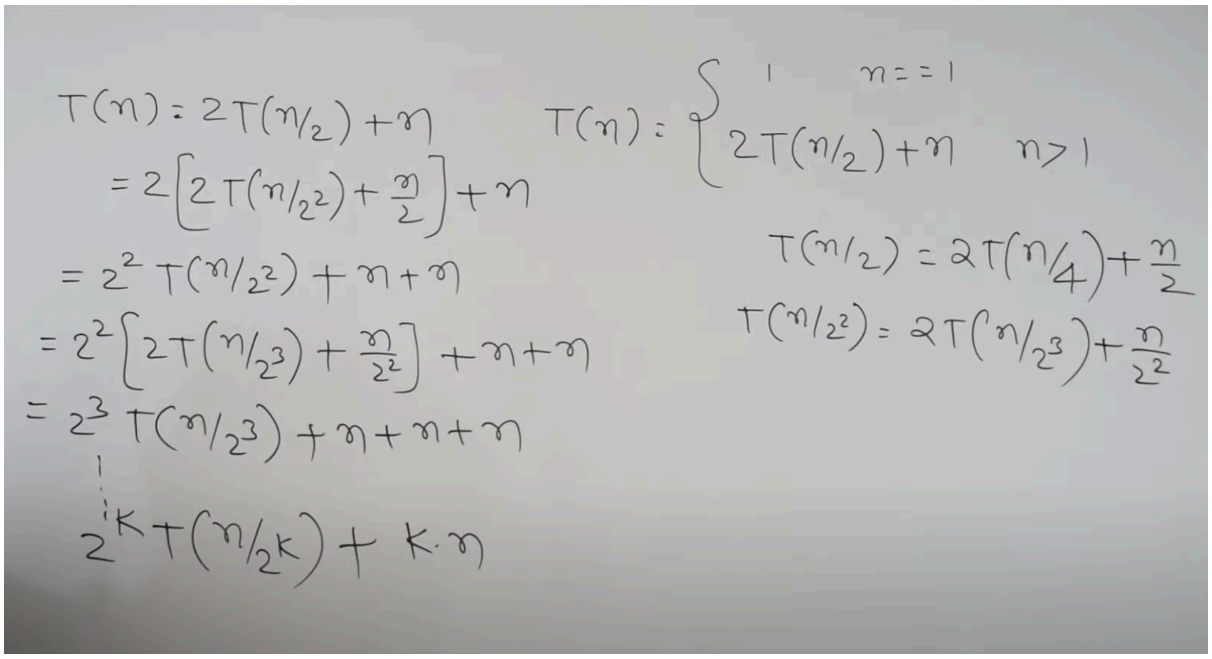
etc.

We obtain the sequence of Fibonacci numbers, which begins

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The recurrence can be solved by methods described below yielding Binet's formula, which involves powers of the two roots of the characteristic polynomial $t2=t+1$; the generating function of the sequence is the rational function

t **/ 1−t−t2.**

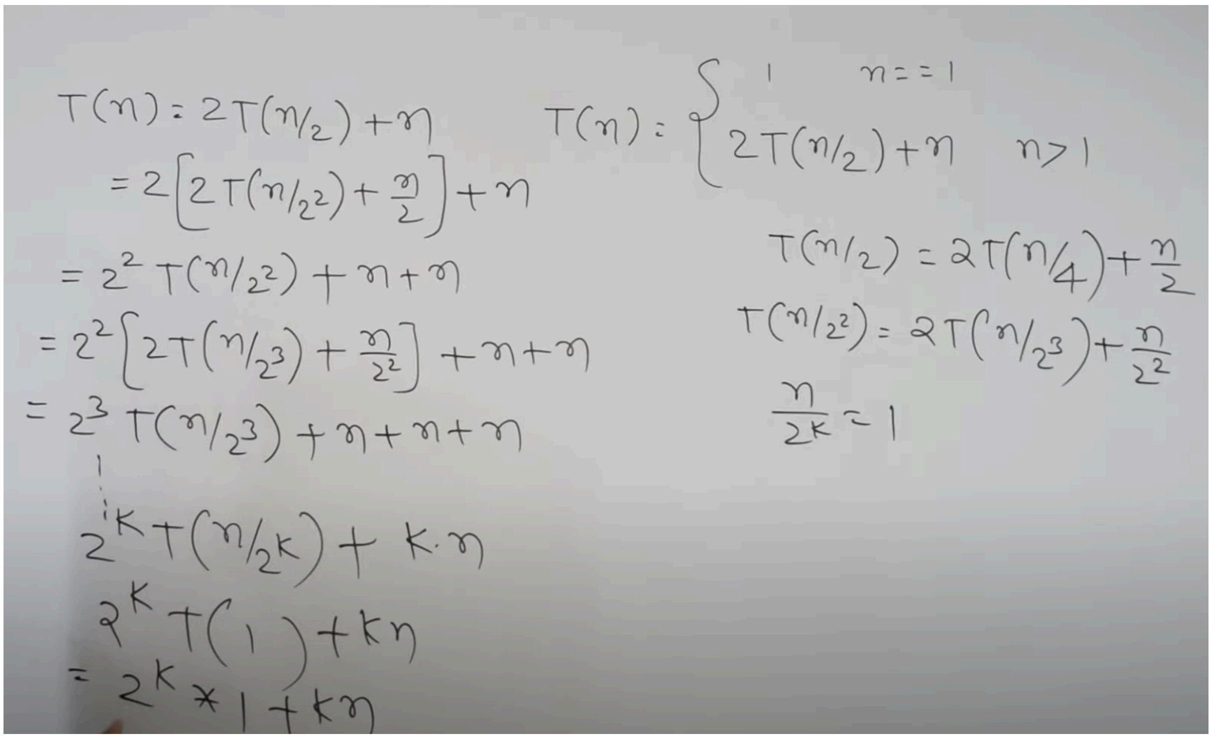


$$T(n) = \begin{cases} 1 & n==1 \\ 2T(n/2)+n & n>1 \end{cases}$$

$$\begin{aligned}
T(n) &= 2T(n/2)+n \\
&= 2\left[2T(n/2^2)+\frac{n}{2}\right]+n \\
&= 2^2\,T(n/2^2)+n+n \\
&= 2^2\left[2T(n/2^3)+\frac{n}{2^2}\right]+n+n \\
&= 2^3\,T(n/2^3)+n+n+n \\
&\vdots \\
&\quad 2^k\,T(n/2^k)+k\cdot n
\end{aligned}$$

$$T(n/2) = 2T(n/4)+\frac{n}{2}$$

$$T(n/2^2) = 2T(n/2^3)+\frac{n}{2^2}$$



$$T(n) = 2T(n/2) + n \qquad T(n) = \begin{cases} 1 & n == 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

$$= 2\left[2T(n/2^2) + \frac{n}{2}\right] + n$$

$$= 2^2\, T(n/2^2) + n + n$$

$$= 2^2\left[2T(n/2^3) + \frac{n}{2^2}\right] + n + n$$

$$= 2^3\, T(n/2^3) + n + n + n$$

$$\vdots$$

$$2^k\, T(n/2^k) + k \cdot n$$

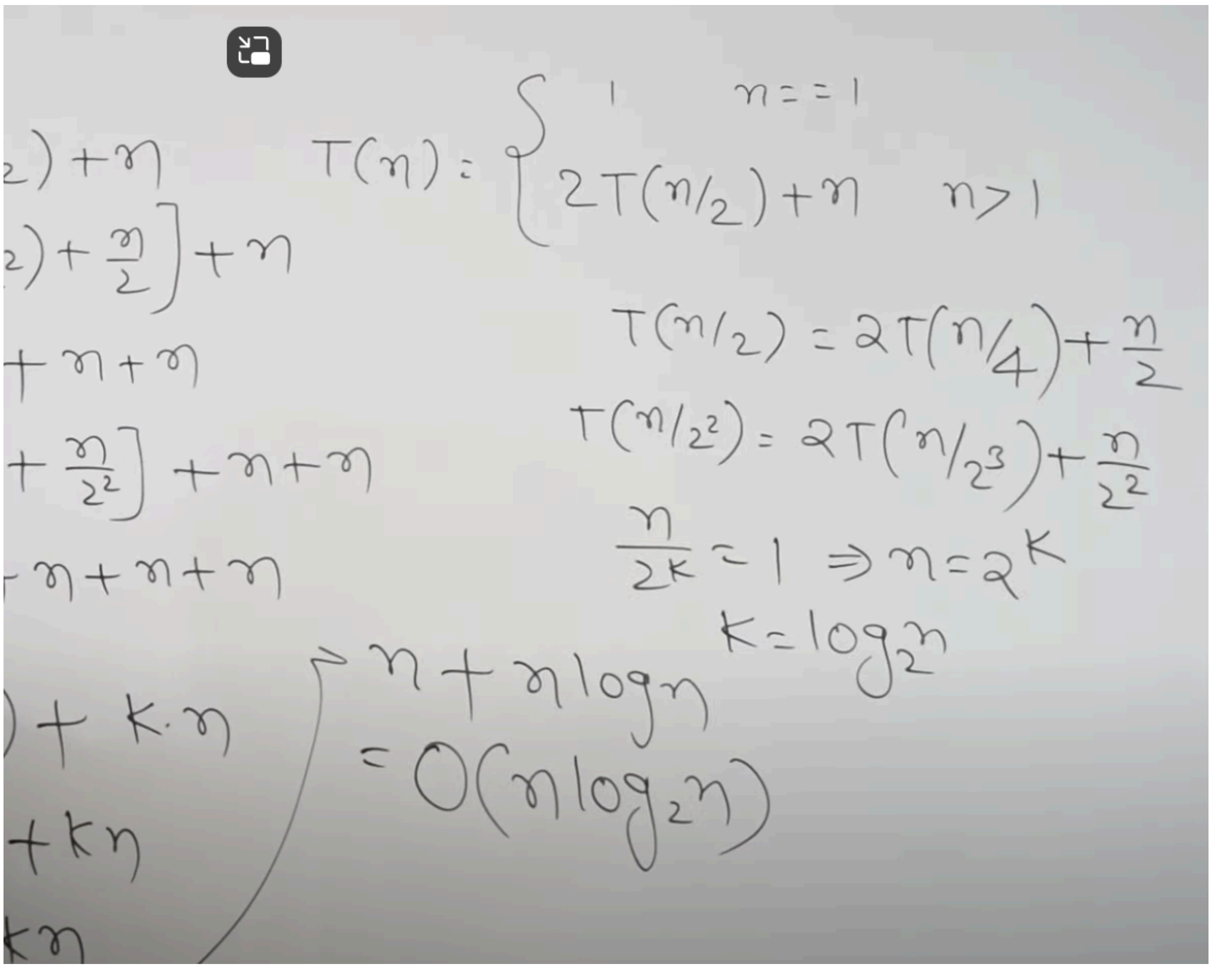
$$2^k\, T(1) + kn$$

$$= 2^k * 1 + kn$$

$$T(n/2) = 2T(n/4) + \frac{n}{2}$$

$$T(n/2^2) = 2T(n/2^3) + \frac{n}{2^2}$$

$$\frac{n}{2^k} = 1$$



$$T(n) = \begin{cases} 1 & n == 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

$$T(n/2) = 2T(n/4) + \frac{n}{2}$$

$$T(n/2^2) = 2T(n/2^3) + \frac{n}{2^2}$$

$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k$$

$$k = \log_2 n$$

$$\Rightarrow n + n\log n$$

$$= O(n \log_2 n)$$

In mathematics, a recurrence relation is an equation according to which the nth term of a sequence of numbers is equal to some combination of the previous terms. Often, only k previous terms of the sequence appear in the equation, for a parameter k that is independent of n;

this number k is called the *order* of the relation.

If the values of the first k numbers in the sequence have been given, the rest of the sequence can be calculated by repeatedly applying the equation.

A *recurrence relation* is an equation that expresses each element of a sequence as a function of the preceding ones.

Factorial

The factorial is defined by the recurrence relation

$$n! = n(n-1)!$$

For $n>0$, and the initial condition

$$0!=1.$$

This is an example of a *linear recurrence with polynomial coefficients* of order 1, with the simple polynomial

$$f(n) = n$$

as its only coefficient.

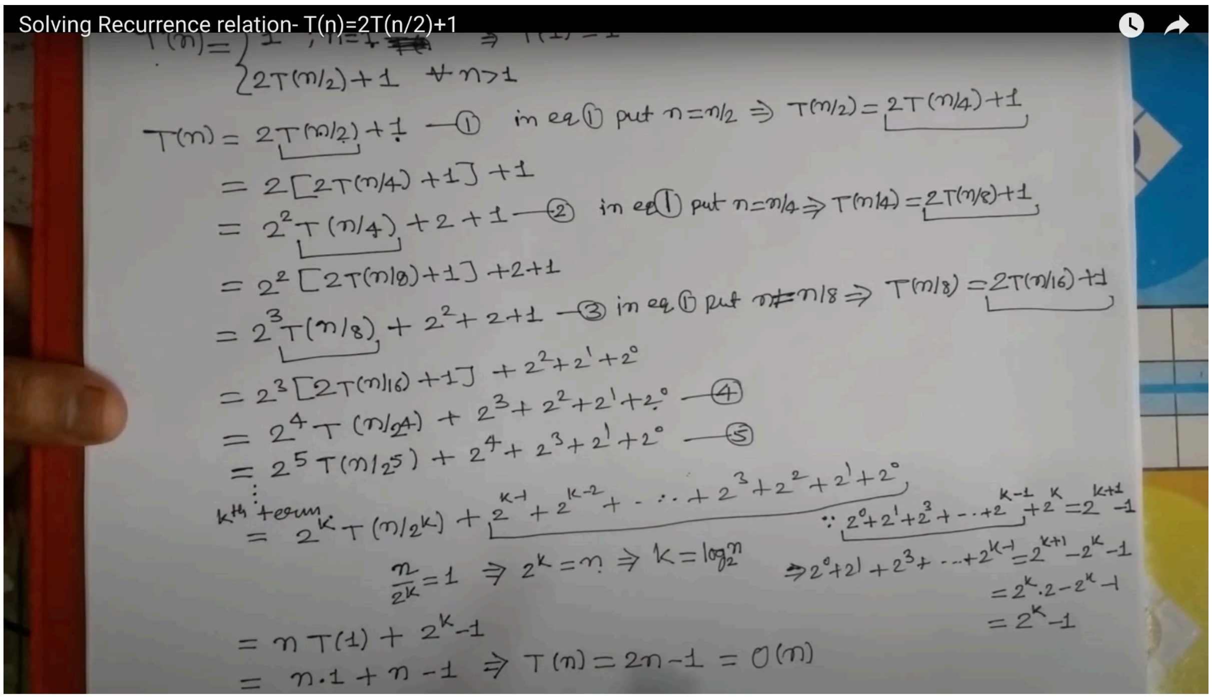
A recurrence relation is an equation that d**efines a sequence** by **relating its terms to one or more preceding terms**. **It tells you how to find the next term in the sequence based on the previous terms.**

There are several methods to solve recurrence relations, depending on the specific type of relation. Here are some common approaches:

1. **Iterative method:** This is a straightforward approach where you simply write out the first few terms of the sequence using the given recurrence relation and initial conditions. By observing the pattern, you might be able to guess the closed form for the nth term.
2. **Characteristic equation method:** This method works well for linear homogeneous recurrence relations with constant coefficients. You assume a solution of the form `a^n` (where a is a constant) and substitute it into the recurrence relation. This leads

to a characteristic equation, which you solve to find the values of a. The general solution will then be a linear combination of terms of the form a^n where a are the roots of the characteristic equation.

3. **Generating functions method:** This method involves representing the sequence as a power series, where the coefficient of the nth term is the nth term of the sequence. By manipulating the generating function equation, you can arrive at a closed form for the sequence.


Solving Recurrence relation- T(n)=2T(n/2)+1

$$T(n) = \begin{cases} 1 & ; n = 1 \Rightarrow T(1) = 1 \\ 2T(n/2) + 1 & \forall n > 1 \end{cases}$$

$T(n) = 2\underbrace{T(n/2)} + 1$ — ① in eq ① put $n = n/2 \Rightarrow T(n/2) = \underbrace{2T(n/4)+1}$

$= 2[2T(n/4) + 1] + 1$

$= 2^2 \underbrace{T(n/4)} + 2 + 1$ — ② in eq ① put $n = n/4 \Rightarrow T(n/4) = \underbrace{2T(n/8)+1}$

$= 2^2 [2T(n/8) + 1] + 2 + 1$

$= 2^3 \underbrace{T(n/8)} + 2^2 + 2 + 1$ — ③ in eq ① put $n = n/8 \Rightarrow T(n/8) = \underbrace{2T(n/16)+1}$

$= 2^3 [2T(n/16) + 1] + 2^2 + 2^1 + 2^0$

$= 2^4 T(n/2^4) + 2^3 + 2^2 + 2^1 + 2^0$ — ④

$= 2^5 T(n/2^5) + 2^4 + 2^3 + 2^1 + 2^0$ — ⑤

$\vdots$

kth term

$= 2^k T(n/2^k) + \underbrace{2^{k-1} + 2^{k-2} + \cdots + 2^3 + 2^2 + 2^1 + 2^0}$

$\because \underbrace{2^0 + 2^1 + 2^3 + \cdots + 2^{k-1}} + 2^k = 2^{k+1} - 1$

$\Rightarrow 2^0 + 2^1 + 2^3 + \cdots + 2^{k-1} = 2^{k+1} - 2^k - 1$

$= 2^k \cdot 2 - 2^k - 1$

$= 2^k - 1$

$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

$= n\,T(1) + 2^k - 1$

$= n \cdot 1 + n - 1 \Rightarrow T(n) = 2n - 1 = O(n)$

Write down the advantages of dynamic programming over the greedy strategy. Find the optimal bracketing to multiply 4 matrices of order 2, 3, 4, 2, 5.

Advantages of Dynamic Programming over Greedy Strategy:

4. **Optimality**: Dynamic programming **guarantees finding the globally optimal solution** by **considering all possible subproblems,** whereas greedy algorithms make **locally optimal choices** *without considering future consequences.*
5. **Complexity Handling**: Dynamic programming **handles more complex problems** where decisions at one stage affect future decisions, which greedy algorithms may overlook due to their myopic nature.
6. **Flexibility in Problem Types**: Dynamic programming can be applied to a broader range of problems that exhibit overlapping subproblems and optimal substructure, whereas greedy algorithms are suitable only for problems with the greedy choice property.

7. **Memory Utilization**: Dynamic programming can trade memory for computation time by storing solutions to subproblems in a table (memoization), which can be reused, whereas greedy algorithms typically do not require extra memory beyond what is needed for the input.
8. **Guaranteed Solution Quality**: Dynamic programming ensures that the solution found is optimal by systematically considering all possibilities and using previously computed subproblem solutions, whereas greedy algorithms may not always find the best solution.

What is a Greedy Approach?

A Greedy approach is one of the most famous techniques utilized to solve problems. It is an algorithm used for solving the problem by choosing the best option open at the moment. This technique needs to focus on the optimal result or whether the current result will impact the optimal output or not.

Also, this technique follows the top-down approach and never changes the judgment even if the option is wrong.

What is Dynamic Programming?

Dynamic Programming (DP) is a method used for decrypting an optimization problem by splitting it down into easier subproblems so that we can reuse the results. These techniques are mainly used for optimization. Some important pointers related to dynamic programming are mentioned below:

- The problem should be able to be split into easier subproblems or overlapping sub-problems.
- We can get the optimum solution through the optimum output of smaller subproblems.
- This algorithm technique prefers memoization.

Difference between the Greedy Approach and Dynamic Programming

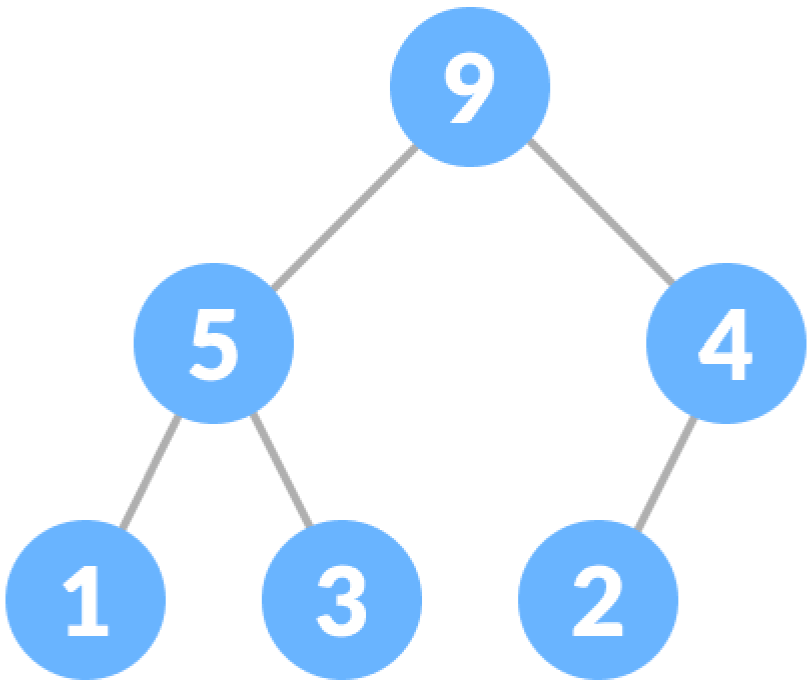
S. No.	Greedy Method	Dynamic Programming
1	In a Greedy Algorithm, we make our decision based on the best current situation.	In Dynamic Programming, we select individually in every step, however, the selection may rely on the solution to sub-problems.
2	In this technique, there is no assurance of obtaining the optimal solution.	With this technique, you can get the assurance of an optimal solution.
3	A greedy approach is used to get the optimal solution.	Dynamic programming is also used to get the optimal solution.
4	The greedy method never alters the earlier choices, thus making it more efficient in terms of memory.	This technique prefers memoization due to which the memory complexity increases, making it less efficient.
5	Greedy techniques are faster than dynamic programming.	Dynamic programming is comparatively slower.

Discuss the heapify operation with an example. Write down its algorithm and analyze its time and space complexity. (10)

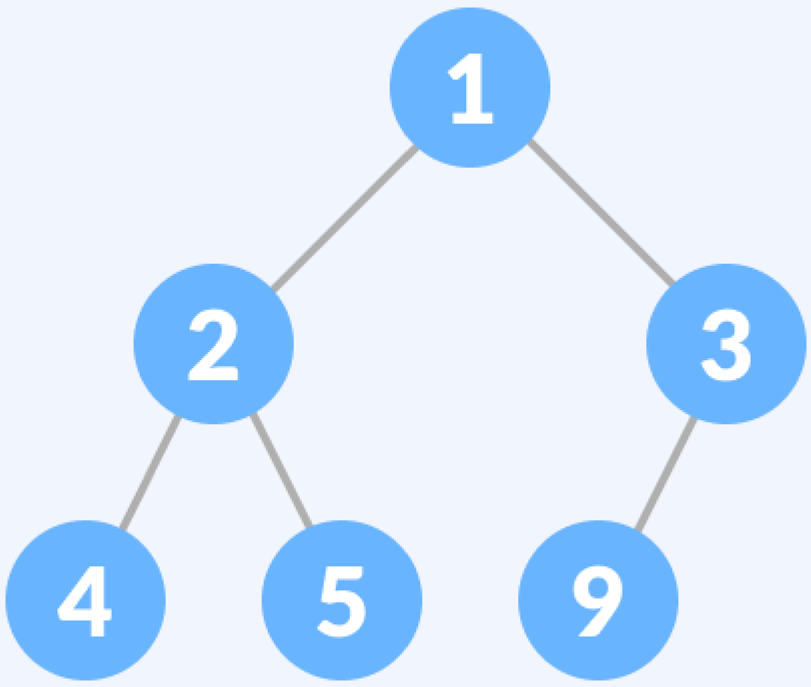
Heap Data Structure

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.



Max-heap



Min-heap

This type of data structure is also called a binary heap.

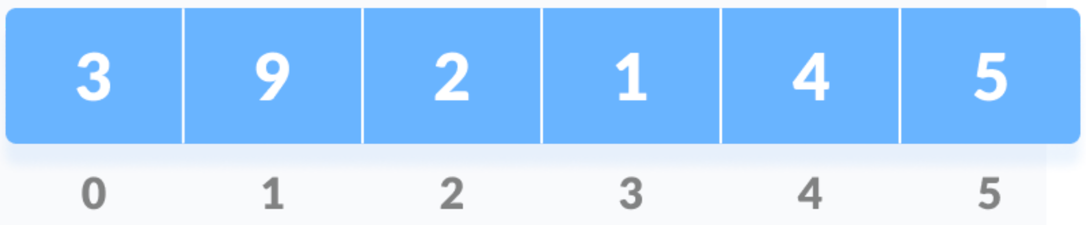
Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

Heapify

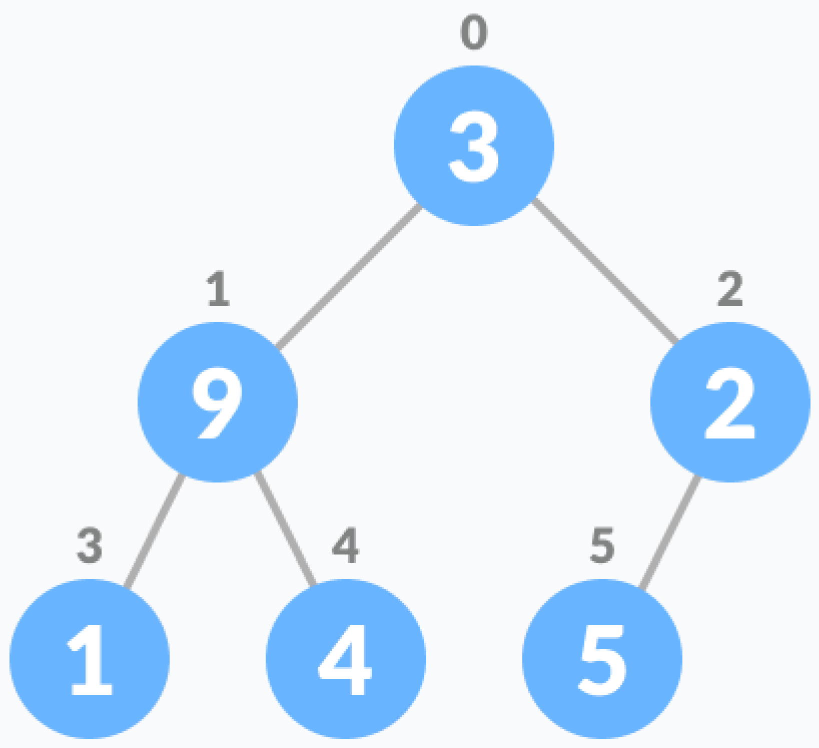
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

1. Let the input array be



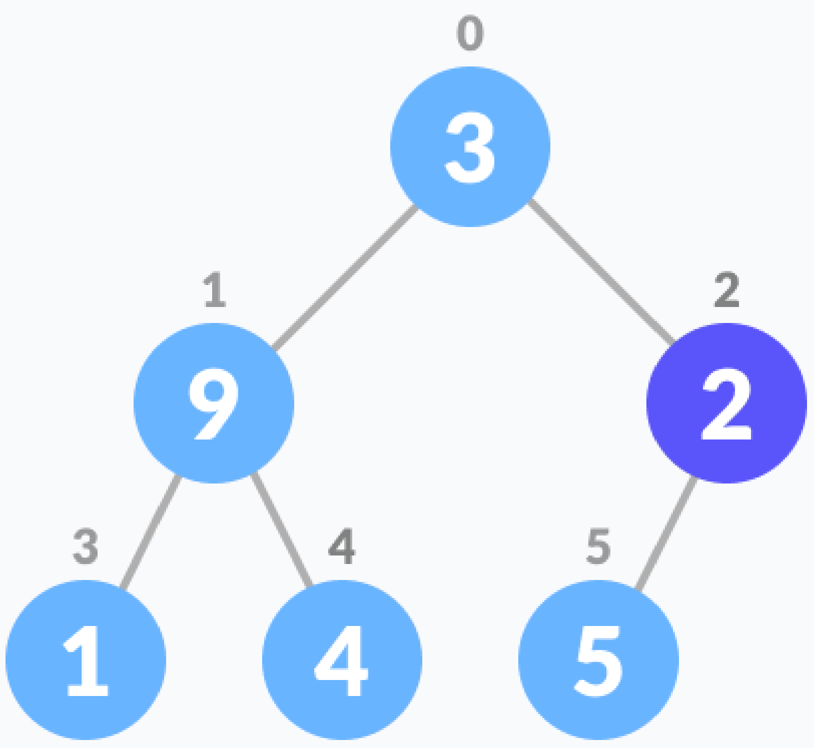
Initial Array

2. Create a complete binary tree from the array



Complete binary tree

3. Start from the first index of non-leaf node whose index is given by `n/2`



`- 1`.

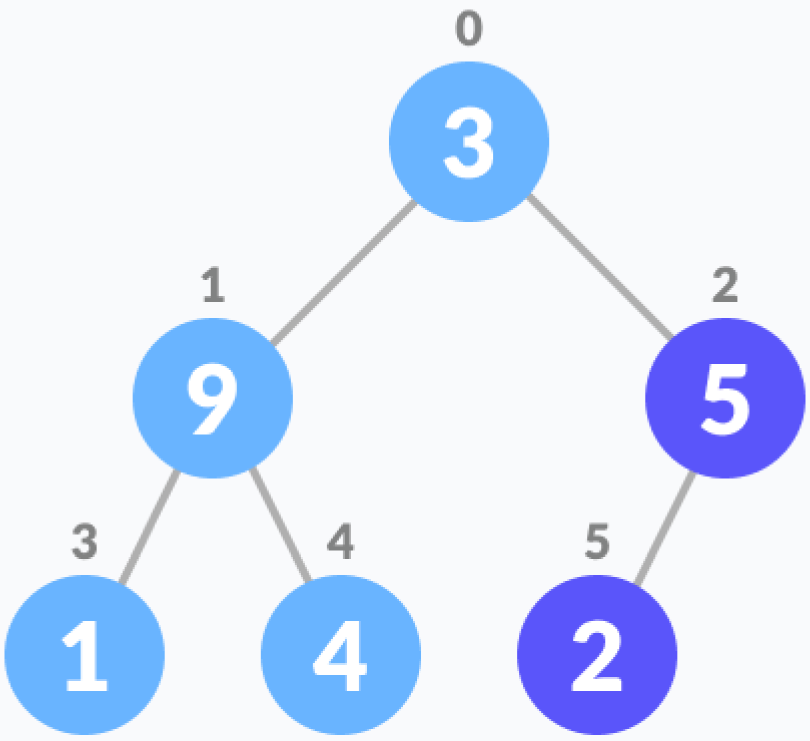
Start from the first on leaf node

4. Set current element `i` as `largest`.
5. The index of left child is given by `2i + 1` and the right child is given by `2i + 2`.

   If `leftChild` is greater than `currentElement` (i.e. element at `ith` index), set `leftChildIndex` as largest.

   If `rightChild` is greater than element in `largest`, set `rightChildIndex` as `largest`.

6. Swap `largest` with `currentElement`



Swap if necessary

7. Repeat steps 3-7 until the subtrees are also heapified.
8.

Complexity Analysis of Heap Sort

Time Complexity: O(N log N)

Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.

Time and Space Complexity

1. **Time Complexity:**
   - The heapify operation involves a comparison of the root with its children and possibly a swap and recursive call.
   - In a binary heap with n nodes, the height of the tree is O(logn)O(\log n)O(logn). The worst-case time complexity of

    heapifying a subtree is O(logn)O(\log n)O(logn) because the height of the subtree is proportional to the logarithm of the number of nodes.
  - In the worst case, heapify is called O(n)O(n)O(n) times during heap construction, leading to an overall time complexity of O(nlogn)O(n \log n)O(nlogn) for building the heap from an unsorted array.
2. **Space Complexity:**
   - The space complexity is O(1)O(1)O(1) for the heapify operation itself because it uses a constant amount of additional space beyond the input array.
   - Recursion depth can be up to O(logn)O(\log n)O(logn) in the worst case, so the auxiliary space for recursive function calls is O(logn)O(\log n)O(logn).

Summary:

- **Heapify Algorithm:** Ensures a subtree rooted at a given node satisfies the heap property.
- **Time Complexity:** O(logn)O(\log n)O(logn) for each heapify operation.
- **Space Complexity:** O(1)O(1)O(1) auxiliary space plus O(logn)O(\log n)O(logn) for recursion.

```
heapify(array, size, i)

 largest <- i                # Initialize largest as root

 left <- 2*i + 1             # Left child index

 right <- 2*i + 2            # Right child index



 # If left child exists and is greater than root

 if left < size and array[left] > array[largest]

   largest <- left
 # If right child exists and is greater than the current
largest
 if right < size and array[right] > array[largest]
   largest <- right


 # If largest is not the root, swap and heapify the affected
subtree
 if largest != i
   swap array[i] and array[largest]
   heapify(array, size, largest)  # Recursively heapify the
affected subtree
```

Section B
Attempt any eight questions.

1. Define the RAM model. Write down the iterative algorithm for finding factorial and provide its detailed analysis. (5)

   ### RAM Model

   RAM Model (Random Access Machine Model) is a theoretical model of computation used to describe the performance of algorithms. It provides a simple abstraction of a computer's operations, allowing for the analysis of algorithms in a more straightforward manner.

   Key Features of the RAM Model:

   Registers: The model includes a set of registers, each of which can hold an integer value.

   Memory Access: The model allows for direct access to any memory location in constant time, which is why it's called "random access."

Arithmetic Operations: Basic arithmetic operations (addition, subtraction, multiplication, division) are performed in constant time.

Instructions: The RAM model includes instructions for arithmetic operations, memory access, and control flow.

No Input/Output Costs: Input and output operations are not considered in the model, focusing only on computation.

RAM Model

The Random Access Machine (RAM) model is a theoretical model of computation that approximates the capabilities and performance characteristics of a real computer. It assumes:

- A fixed-size memory is divided into words.
- Each memory location can be accessed in constant time.
- A CPU with a fixed set of instructions that can perform arithmetic, logical, and control operations.
- Instructions are executed sequentially.

F**actorial Algorithm:** The iterative algorithm for computing factorial has a time complexity of O(n) and a space complexity of O(1), making it efficient for calculating factorial values.

```
def factorial(n):
 """ Calculates the factorial of a non-negative integer n iteratively."""
 if n == 0:
   return 1
 result = 1
 for i in range(2, n+1):
   result *= i
 return result
```

Analysis of Time and Space Complexity

Time Complexity:

- The algorithm iterates from 2 to n, performing a multiplication in each iteration.
- The loop runs n-1 times.
- Each iteration involves a multiplication and an assignment, which are constant time operations.
- Therefore, the overall time complexity is O(n).

Space Complexity:

- The algorithm uses a constant amount of extra space for variables like `result` and `i`.
- The space used does not grow with the input size.
- Therefore, the space complexity is O(1).

2. Write down the algorithm for insertion sort and analyze its time and space complexity. (5)

Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

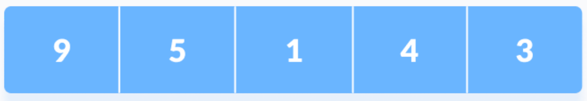
Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

Working of Insertion Sort

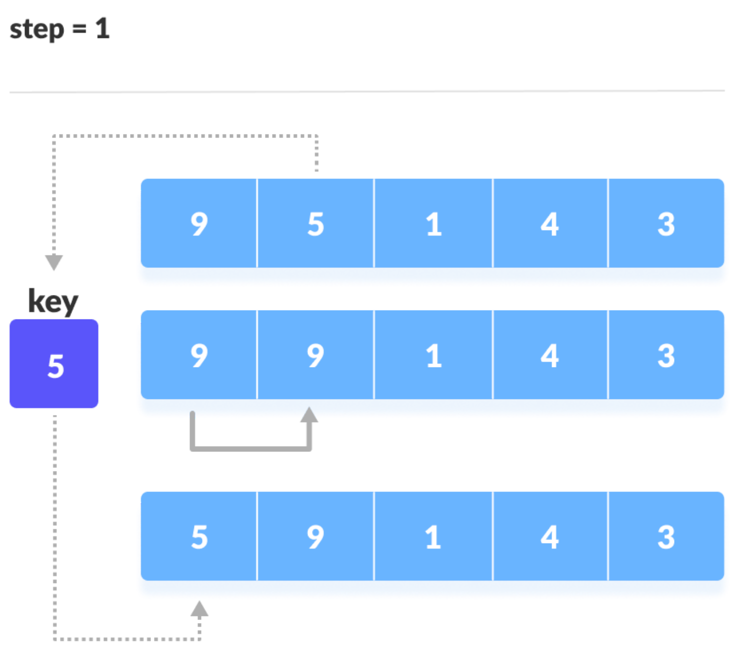
Suppose we need to sort the following array.



Initial array

3. The first element in the array is assumed to be sorted. Take the second element and store it separately in `key`.

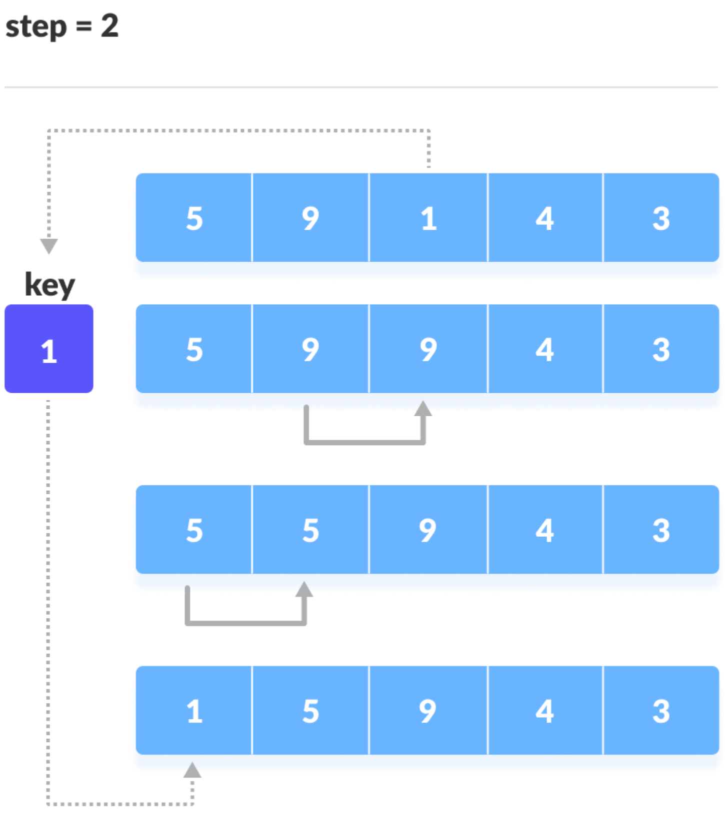
   Compare `key` with the first element. If the first element is greater than `key`, then `key` is placed in front of the first element.



If the first element is greater than key, then key is placed in front of the first element.

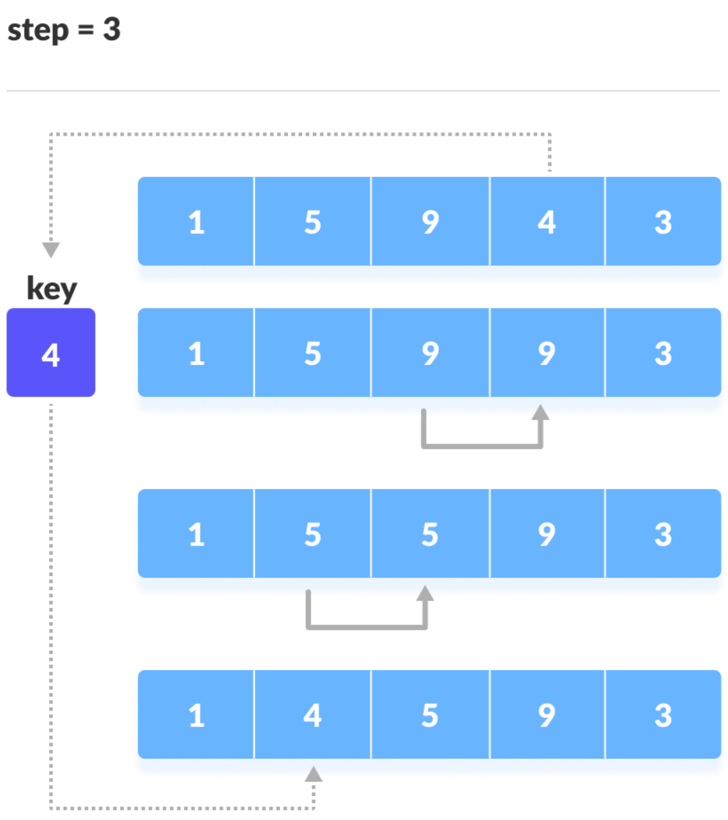
4. Now, the first two elements are sorted.

   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

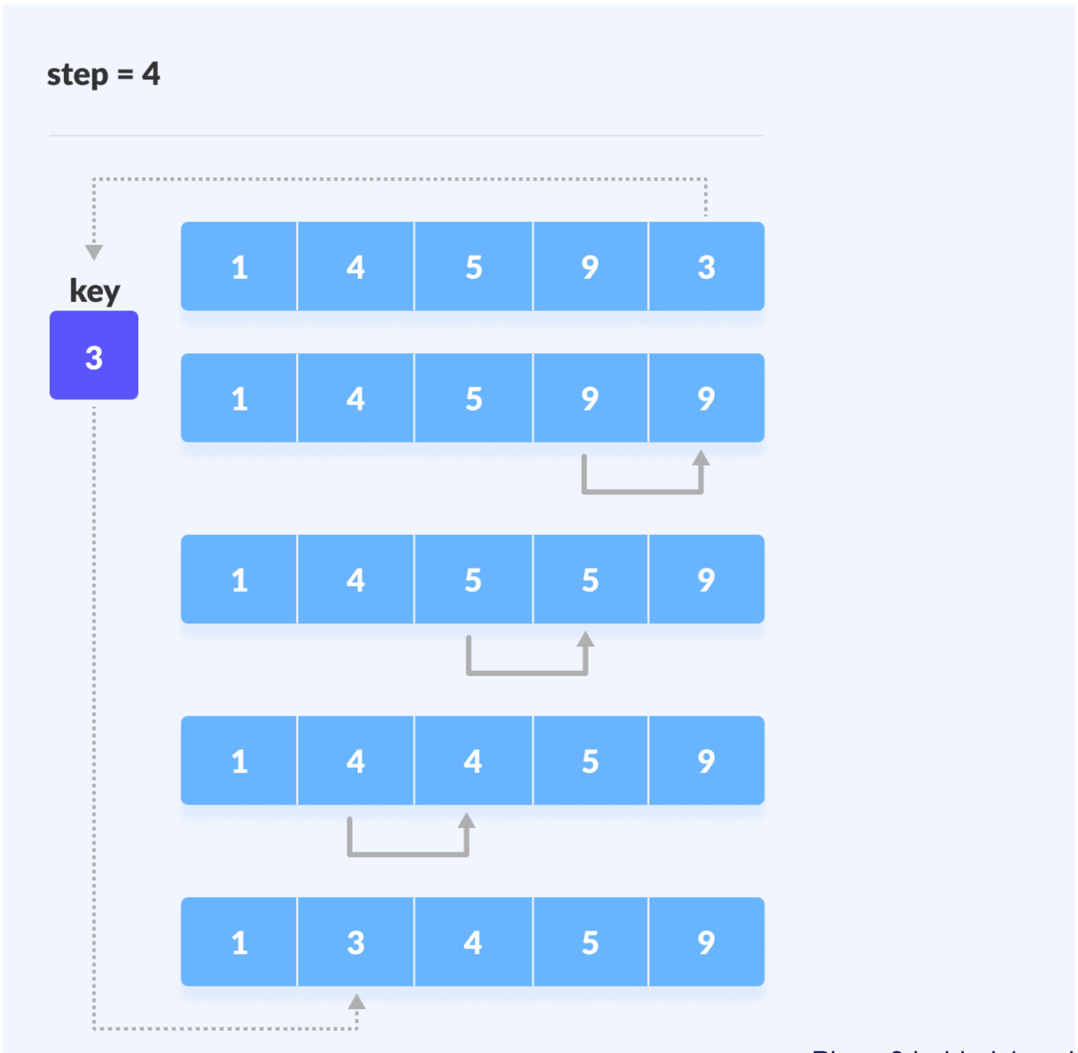


Place 1 at the beginning

5. Similarly, place every unsorted element at its correct position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

Insertion Sort Complexity

Time Complexity	
Best	O(n)
Worst	O(n2)

Average	$O(n^2)$
Space Complexity	O(1)
Stability	Yes

Time Complexities

- Worst Case Complexity: `O(n2)`

  Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

  Each element has to be compared with each of the other elements so, for every nth element, `(n-1)` number of comparisons are made.

  Thus, the total number of comparisons = `n*(n-1) ~ n2`
- Best Case Complexity: `O(n)`

  When the array is already sorted, the outer loop runs for `n` number of times whereas the inner loop does not run at all. So, there are only `n` number of comparisons. Thus, complexity is linear.
- Average Case Complexity: `O(n2)`

  It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Space Complexity

Space complexity is `O(1)` because an extra variable `key` is used.

Insertion Sort Applications

The insertion sort is used when:

6. the array is has a small number of elements
7. there are only a few elements left to be sorted

```
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 $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

8. Write down the min-max algorithm and analyze its complexity. (5)
9. When does the greedy strategy provide an optimal solution? Write down the job sequencing with deadlines algorithm and analyze its complexity. (5)
10. Suppose that a message contains alphabet frequencies as given below. Find the Huffman codes for each alphabet:

Symbol	Frequency
a	30
b	20
c	25
d	15
e	35 (5)

11. Does backtracking give multiple solutions? Trace the subset sum algorithm for the set {3, 5, 2, 4, 1} and sum = 8. (5)
12. Why is the extended Euclidean algorithm used? Write down its algorithm and analyze its complexity. (5)
13. Define NP-complete problems with examples. Give a brief proof of the statement “SAT is NP-complete”. (5)

14. Write short notes on:
    a) Aggregate Analysis
    b) Selection problems (5)