Understanding Data Structures and Algorithms

Using Python

March 18, 2022

Contents

1

Algorithmic Complexity

1.1 Complexity analysis

The complexity of an algorithm is the amount of time and space required by an algorithm for an input of given size n.

Metric of measurement:

- **Speed**: Number of operations required (best, average or worst case)
- **Space** Amount of memory usage

To assess the complexity, the order of the count of operation is always considered instead of the exact number of operations. We use asymptotic notation to analyze the running time of an algorithm.

1.2 Asymptotic notation

Suppose,
n = The size of input, and
$f(n)$ = Number of steps to be executed by the algorithm for input size n

Θ-notation

A function $f(n)$ belongs to set $\Theta\left(g(n)\right)$, if there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for sufficiently large n ($n \geq n_0$).

O-notation (or, Big-O-notation)

A function $f(n)$ belongs to set $O\left(g(n)\right)$, if there exist positive constants c and n such that $0 \leq f(n) \leq cg(n)$ for $n \geq n_0$. O-notation provides an asymptotic upper bound.

Ω-notation

A function $f(n)$ belongs to set $\Omega(g(n))$, if there exist positive constants c and n such that $0 \leq cg(n) \leq f(n)$ for $n \geq n_0$. Ω-notation provides an asymptotic lower bound.

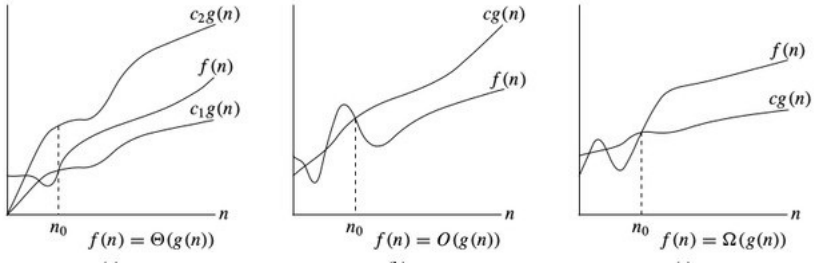


Figure 1.1: Asymptotic notation

Usually we care most about upper bound on the running time of an algorithm, which is why we most often use the Big-O notation for algorithm analysis.

Time complexity	**Big-O notation**
Constant complexity	$O(1)$
Logarithmic complexity	$O(\log n)$
Linear complexity	$O(n)$
Quasilinear complexity	$O(n\log n)$
Quadratic complexity	$O(n^2)$
Qubic complexity	$O(n^3)$
Exponential complexity	$O(2^n)$
Factorial complexity	$O(n!)$

1.3 Recursive algorithms

Iterative algorithms are easy to analyze which involves counting the number of elementary operations. On the other hand analyzing a recursive algorithm is a difficult task. To analyze a recursive algorithm we need to formulate a *recurrence equation* (a.k.a. *difference equation*). For example for the following script to generate the Fibonacci sequence, the recurrence equation for the number of operation is

$$T(n) = \begin{cases} T(n-1) + T(n-2) + 4, & \text{if } n \geq 1 \\ 1, & \text{otherwise} \end{cases}$$

Here the constant 4 is coming due to one comparison, one addition and two subtraction operations.

```
def fib(n):
    if (n <= 1):
        return n
    else:
        return fib(n-1) + fib(n-2)
```

We can use the substitution method to solve a recurrence equation. Substitution method involves two steps, which can be informally written as follows:

- **Plug**: Substitute repeatedly
- **Chug**: Simplify the expression

Example 1: Use the substitution method for the following recurrence equation:

$$T(n) = \begin{cases} T(n-1) + 3, & \text{if } n > 1 \\ 4, & \text{if } n = 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + 3 \text{ (Plug)} \\ &= [T(n-2) + 3] + 3 = T(n-2) + 2 \times 3 \text{ (Chug)} \\ &= T(n-3) + 3 \times 3 \\ &= \cdots \\ &= T(n-k) + k \times 3 \end{aligned}$$

Now let's find the value of k for which $n - k = 1$ i.e. $k = n - 1$. Therefore,

$$\begin{aligned} T(n) &= T(n-(n-1)) + (n-1) \times 3 \\ &= T(1) + 3(n-1) \\ &= 4 + 3(n-1) \\ &= 3n + 1 \in O(n) \end{aligned}$$

Example 1: Use the substitution method for the following recurrence equation:

$$T(n) = \begin{cases} T(n-3) + 5, & \text{if } n \geq 3 \\ 4, & \text{if } n = 0, 1, 2 \end{cases}$$

$$\begin{aligned} T(n) &= T(n-3) + 5 \text{ (Plug)} \\ &= [T(n-3-3) + 5] + 5 = T(n-2 \times 3) + 2 \times 5 \text{ (Chug)} \\ &= T(n-3 \times 3) + 3 \times 5 \\ &= \cdots \\ &= T(n-k \times 3) + k \times 5 \end{aligned}$$

Now let's find the value of k for which $n-3k=0$ i.e. $k=\frac{n}{3}$. If k is integer, we can write

$$\begin{aligned} T(n) &= T(0) + \frac{n}{3} \times 5 \\ &= 4 + \frac{5n}{3} \in O(n) \end{aligned}$$

If $\frac{n}{3}$ is not an integer, then find the value of integer $k = \lfloor \frac{n}{3} \rfloor$ so that,

$$n = 3 \times \lfloor \frac{5n}{3} \rfloor + n\%3$$

Putting the value of $k = \lfloor \frac{n}{3} \rfloor$ in the expression of $T(n)$

$$\begin{aligned} T(n) &= T\left(n - 3 \times \lfloor \frac{5n}{3} \rfloor\right) + \lfloor \frac{5n}{3} \rfloor \times 5 \\ &= T(n\%3) + \lfloor \frac{5n}{3} \rfloor \times 5 \\ &= 4 + \lfloor \frac{5n}{3} \rfloor \times 5 \in O(n) \end{aligned}$$

Example 3: Try the substitution method for the recurrence equation of the Fibonacci sequence:

$$T(n) = \begin{cases} T(n-1) + T(n-2) + c, & \text{if } n > 1 \\ 1, & \text{if } n = 0, 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\leq 2T(n-1) + c \\ &= 2\left[2T(n-2) + c\right] + c \\ &= 2^2 T(n-2) + 2c + c \\ &= 2^2\left[2T(n-3) + c\right] + 2c + c \\ &= 2^3 T(n-3) + 2^2 c + 2c + c \\ &= \cdots \\ &= 2^k T(n-k) + \underbrace{2^{k-1}c + 2^{k-2}c + \cdots + 2^2 c + 2c + c}_{(2^k-1)c} \end{aligned}$$

Now let's find the value of k for which $n-k=0$ i.e. $k=n$. Therefore,

$$\begin{aligned} T(n) &= 2^n T(0) + (2^n - 1) \times c \\ &= 2^n(1+c) - c \in O(2^n) \end{aligned}$$

Example 4: Solve the recurrence equation of the form:

$$T(n) = \begin{cases} T(n-1) + T(n-2) + T(n-3) + c, & \text{if } n > 2 \\ 1, & \text{if } n = 0, 1, 2 \end{cases}$$

In the similar way as before it can be shown that the *time complexity* for this problem is

$$T(n) \in O(3^n)$$

Example 5: Solve the recurrence equation of the form (Geometric sequence):

$$T(n) = \begin{cases} rT(n-1), & \text{if } n > 0 \\ \alpha, & \text{if } n = 0 \end{cases}$$

$$\begin{aligned} T(n) &= rT(n-1) \\ & r \times [rT(n-2)] = r^2T[n-2] \\ & = r^2 \times [rT(n-3)] = r^3T[n-3] \\ & = \cdots \\ & = r^kT[n-k] \end{aligned}$$

Now let's find the value of k for which $n - k = 0$ i.e. $k = n$. Therefore,

$$\begin{aligned} T(n) &= r^nT(0) \\ &= \alpha r^n \in O(r^n) \end{aligned}$$

Example 6: Solve the recurrence equation for calculating factorial:

$$T(n) = \begin{cases} nT(n-1), & \text{if } n > 0 \\ 0, & \text{if } n = 0 \end{cases}$$

$$\begin{aligned} T(n) &= nT(n-1) \\ & n \times [(n-1)T(n-2)] = n(n-1)T[n-2] \\ & = n(n-1) \times [(n-2)T(n-3)] = n(n-1)(n-2)T[n-3] \\ & = \cdots \\ & = n(n-1)(n-2)\cdots(n-(k-1))T[n-k] \end{aligned}$$

Now let's find the value of k for which $n - k = 0$ i.e. $k = n$. Therefore,

$$\begin{aligned} T(n) &= n(n-1)(n-2)\cdots(n-(n-1))T[0] \\ &= n(n-1)(n-2)\cdots 1 \\ &= n! \in O(n!) \end{aligned}$$

1.4 Master Theorem

The Master theorem is a general method for solving (getting *time complexity* for) recurrence relations that arise frequently in *divide and conquer algorithms*, which have the following recurrence relation of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is an asymptotically positive function. The *time complexity* of such a recursive relation is given by the following rules:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.
4. If $f(n) = \Theta(n^{\log_b a} \log^k(n))$ for $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Simply put, if $f(n)$ is polynomially smaller than $n^{\log_b a}$, then the runtime is $\Theta(n^{\log_b a})$. If $f(n)$ is instead polynomially larger than $n^{\log_b a}$, then the runtime is $\Theta(f(n)$. Finally, if $f(n)$ and $n^{\log_b a}$ are asymptotically the same, then the runtime is $\Theta(n^{\log_b a} \log n)$.

Example 1. $T(n) = 9T(n/3) + n$

- Here $a = 9$, $b = 3$, $f(n) = n$, $n^{\log_b a} = n^2$. n^2 is polynomially bigger.
- So, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

Example 2. $T(n) = \Theta(f(n)) = 3T(n/7) + n^{\frac{3}{4}}$

- Here $a = 3$, $b = 7$, $f(n) = n^{\frac{3}{4}} = n^{0.75}$, $n^{\log_b a} \approx n^{0.56}$. $n^{0.75}$ is polynomially bigger.
- So, $T(n) = \Theta(f(n)) = \Theta(n^{\frac{3}{4}})$.

Note: Since $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, Master Theorem definitely applies!

Example 3. $T(n) = T(4n/5) + \log^2 n$

- Here $a = 1$, $b = 5/4$, $f(n) = \log^2 n$, $n^{\log_b a} = 1$. $f(n)$ is not polynomially bigger. But Rule 4 applies (with $k = 2$).
- So, $T(n) = \Theta(\log^3(n))$.

Example 4. $T(n) = 2T(n/4) + \sqrt{n}\log\log(n)$

- Here $a = 2$, $b = 4$, $f(n) = \sqrt{n}\log\log n$, $n^{\log_b a} = n^{1/2}$, so $f(n)$ is not polynomially larger than $n^{\log_b a}$, it is larger only by a logarithmic factor.
- So, Master Theorem does not apply.

2

Data Structures

2.1 Linked list

A linked list is a linear collection of data elements in which the linear ordering is maintained by each element pointing to the next (not by physical placements in memory). It is a data structure consisting of a group of nodes which together form a sequence. Each node is composed of two parts: **data** and a **reference** (link) to the next node in the sequence.

Advantages:

1. Unlike list (array), they are dynamic in nature which allocates memory whenever required.
2. This structure allows efficient insertion and removal of elements.
3. Stacks and queues can easily be implemented.
4. Linked list reduces the access time.

Disadvantages:

1. The memory is wasted as the reference requires extra memory for storage.
2. No element can be accessed randomly.
3. Reverse traversing is difficult.

2.1.1 Inserting item into linked list

We can add elements to either at the end or at the beginning or in the middle of the linked list.

Insert at the end

- Traverse to last node.
- Change next of last node to recently created node.

Time complexity: $O(n)$

Insert at the begining

- Change next of new node to point to head.
- Set the recently created node as head of linked list .

Time complexity: $O(1)$

Insert in the middle

- Traverse to node x just before the required position.
- Set the next pointer of the new node to the next pointer of node x.
- Change next pointer of node x to the new node.

Time complexity: $O(n)$

```
class Node:
    # Constructor
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    # Constructor
    def __init__(self):
        self.head = None

    # Display the linked list
    def display(self):
        cur_node = self.head
        while cur_node:
            print('['+str(cur_node.data)+'] ---> ', end='')
            cur_node = cur_node.next
        print('None')

    # Returns the length of the linked list
    def length(self):
        count = 0
        cur_node = self.head
```

```
        while cur_node:
            count += 1
            cur_node = cur_node.next
        return count

    # Insert item at the end of linked list
    # \textit{Time complexity}: O(n)
    def insertLast(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return None

        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    # Insert item at the beginning of linked list
    # \textit{Time complexity}: O(1)
    def insertFirst(self, data):
        new_node = Node(data)

        new_node.next = self.head
        self.head = new_node

    # Insert item at a specific position of linked list
    def insert_at_pos(self, pos, data):
        if pos < 0 or pos > self.length():
            print('No node exists at this position')
            return None

        new_node = Node(data)
        cur_node = self.head
        if pos == 0:
            self.insertFirst(data)
        else:
            for _ in range(pos-1):
                cur_node = cur_node.next
            new_node.next = cur_node.next
            cur_node.next = new_node

if __name__ == "__main__":
    linklist_obj = LinkedList()
    linklist_obj.insertLast("B")
    linklist_obj.insertLast("C")
    linklist_obj.insertFirst("A")
    linklist_obj.display()
```

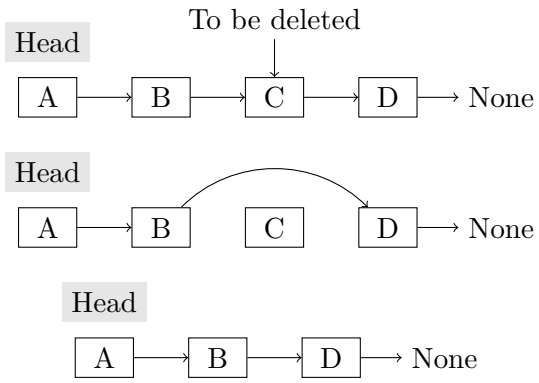


Figure 2.1: Deleting item from linked list

```
linklist_obj.insert_at_pos(2, 'D')
linklist_obj.insert_at_pos(1, 5)
linklist_obj.display()
```

Output

```
[A] ---> [B] ---> [C] ---> None
[A] ---> [5] ---> [B] ---> [D] ---> [C] ---> None
```

2.1.2 Deleting item from linked list

To delete a node from the linked list, we need to do the following steps.

- Find the previous node of the node to be deleted.
- Make the previous node point to the node which is after the node to be deleted.
- Free memory for the node to be deleted.

Time complexity: $O(n)$

```
# Delete item by value (key)
def delete_by_key(self, key):
    cur_node = self.head

    if cur_node and cur_node.data == key:
        self.head = cur_node.next
        cur_node = None
        return None
```

```
    prev_node = None
    while cur_node and cur_node.data != key:
        prev_node = cur_node
        cur_node = cur_node.next

    if cur_node is None:
        return None

    prev_node.next = cur_node.next
    cur_node = None

# Delete item by position(index)
def delete_at_pos(self, pos):

    cur_node = self.head

    if pos == 0:
        self.head = cur_node.next
        cur_node = None
        return None

    prev_node = None
    count = 1
    while cur_node and count != pos:
        prev_node = cur_node
        cur_node = cur_node.next
        count += 1

    if cur_node is None:
        return

    prev_node.next = cur_node.next
    cur_node = None
```

2.1.3 Search element in a linked list using key

Time complexity: $O(n)$

```
# Search element in a linked list using key
def searchkey(self, key):
    cur_node = self.head
    pos = 0

    if self.head == None:
        print('No such value found in the Linked List')
    else:
```

```
        while cur_node:
            if cur_node.data == key:
                print("Search value index = "+str(pos))
                return None
            cur_node = cur_node.next
            pos += 1
        if cur_node == None:
            print('No such value found in the Linked List')
```

2.1.4 Reversing a linked list

Iterative method:

1. Initialize pointers `prev_node` as NULL, `cur_node` as `head`.
2. Iterate through the linked list. In loop, do following.
   (a) Create a `next_node` pointer for saving `current_node.next`
   (b) Point the `cur_node` point to the previous node.
   (c) Move `prev_node` and `cur_node` one step forward

Time complexity: $O(n)$
Space complexity: $O(1)$

```
def reverse(self):
    cur_node = self.head
    prev_node = None

    while cur_node:
        next_node = cur_node.next
        # next_node for saving cur_node.next
        cur_node.next = prev_node
        prev_node = cur_node
        cur_node = next_node
    self.head = prev_node
```

Using recursive method:

1. Divide the list in two parts - first node and rest of the linked list.
2. Call reverse for the rest of the linked list.
3. Link rest to first.
4. Fix head pointer

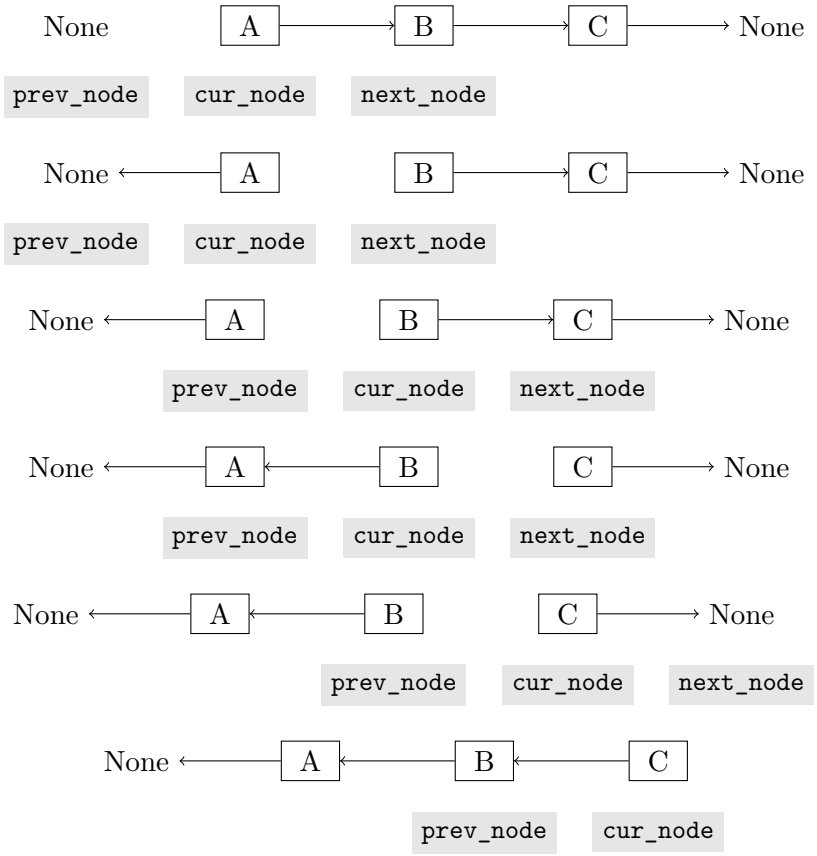


Figure 2.2: Reversing a linked list: iterative approach

Time complexity: $O(n)$
Space complexity: $O(1)$

```
def reverse(self, head):
    # If head is empty or has reached the list end
    if head is None or head.next is None:
        return head

    # Reverse the rest list
    rest = self.reverse(head.next)

    # Put first element at the end
    head.next.next = head

    # Fix the header pointer
    head.next = None

    return rest
```

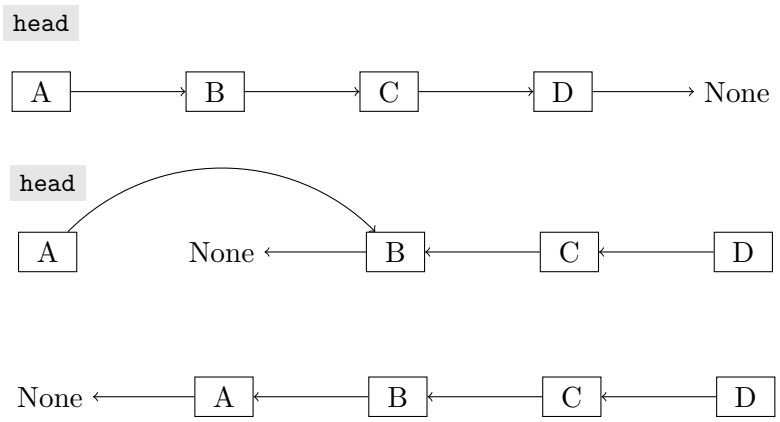


Figure 2.3: Reversing a linked list: recursive approach

2.1.5 Check if a linked list is palindrome or not (in constant *space complexity*)

1. Divide the list into two equal parts. If the linked list contains an odd number of nodes, then ignore the middle element.
   (a) Create two pointers `slow` and `fast` which moves by one node and two nodes at a time respectively. Both start at the head.
   (b) After traversing the list, the `fast` pointer would point to the last node if the number of nodes is odd. If the number of nodes is even, the `fast` pointer would point to the None.

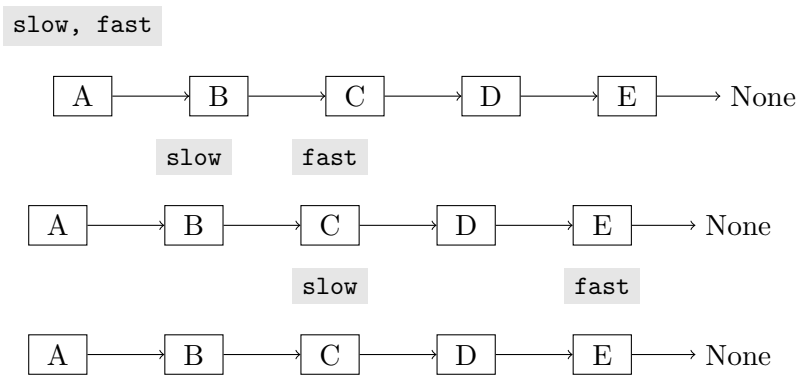


Figure 2.4: Check palindrome: Movement of `slow` and `fast` pointers for odd number of nodes

   (c) If the number of nodes is even, the slow pointer would be pointing to the first element of the second list. On the other hand, if the

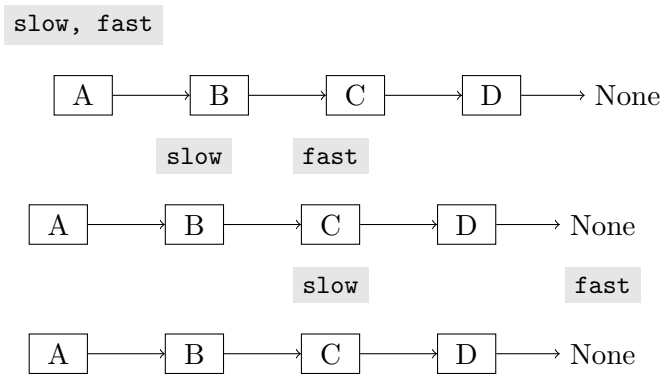


Figure 2.5: Check palindrome: Movement of `slow` and `fast` pointers for even number of nodes

number of nodes is odd the slow pointer would be pointing to the middle node. the next node is first element of the second list.

2. Reverse the second half.
3. Check if the first and second half is similar.

Time complexity: $O(n)$
Space complexity: $O(1)$

```
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

# Recursive function to reverse nodes of a linked list
def reverse(head):
    # If head is empty or has reached the list end
    if head is None or head.next is None:
        return head
    # Reverse the rest list
    rest = reverse(head.next)
    # Put first element at the end
    head.next.next = head
    # Fix the header pointer
    head.next = None
    return rest

# Recursive function to check if two linked lists are equal or not
def compare(a, b):
    # see if both lists are empty
    if a is None and b is None:
        return True
```

```
    return a and b and (a.data == b.data) and compare(a.next,
        b.next)

# Function to split the linked list into two equal parts and return
    the pointer to the second half
def findMiddle(head):
    prev = None
    slow = head
    fast = head

    # Find the middle pointer
    while fast and fast.next:
        prev = slow
        slow = slow.next
        fast = fast.next.next

    # For odd nodes, 'fast' points to the last node
    # For even number of nodes, 'fast' points to None
    if fast:
        odd = True
    else:
        odd = False

    # Make next of previous node (of slow pointer) to None
    prev.next = None

    # If the total number of nodes is odd, advance mid by one node
        to get the head of the second linked list
    if odd:
        return slow.next
    else:
        return slow

# Function to check if the linked list is a palindrome or not
def checkPalindrome(head):
    # Base case
    if head is None or head.next is None:
        return True

    # Find the second half of the linked list
    # as well as the odd or even indicator flag
    head_second = findMiddle(head)

    # reverse the second half
    head_second = reverse(head_second)

    # compare the first and second half
    return compare(head, head_second)
```

```
if __name__ == '__main__':
    head = Node(1)
    head.next = Node(2)
    head.next.next = Node(3)
    head.next.next.next = Node(2)
    head.next.next.next.next = Node(1)

    if checkPalindrome(head):
        print('The linked list is a palindrome')
    else:
        print('The linked list is not a palindrome')
```

Output

```
The linked list is a palindrome
```

2.2 Stack

Stacks are the data structures in which the items to be inserted or removed follow the **Last-in-First-out** (LIFO) principle. The insertion or deletion of items takes place one end of the stack known as the **top** of the stack.
When an item is added into a stack, the operation is called **push** and when an item is removed from the stack, the operation is called **pop**.
Most microprocessors use a stack-based architecture. When a method (subroutine) is called, its return address is pushed onto a stack and when it returns, the address is popped off.

Applications of stack:

- Reversing a word: You push a word onto a stack letter-by-letter and then pop letters from the stack.
- Doing 'undo' mechanism in text editors. This is accomplished by keeping all text changes in a stack.
- Compiler's syntax check for matching parentheses is acomplished by using stack.

```
class Stack():
    # Constructor
    def __init__(self):
        self.items = []

    # Push item onto stack
    def push(self, item):
        self.items.append(item)

    # Pop item from stack
    def pop(self):
        return self.items.pop()

    # Check whether the stack is empty or not
    def is_empty(self):
        return self.items == []

    # Peek at the topmost element of the stack
    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    # Display the stack
    def display(self):
        #print(self.items[::-1])
```

```
        for i in self.items[::-1]:
            st = '| '+str(i)+' |'
            print(st)
            print('-'*len(st))

if __name__ == "__main__":
    s = Stack()

    s.push(1)
    s.push('A')
    s.push(31)
    s.push(4)
    s.display()

    print('Popped item =', s.pop())
    s.display()
```

Output

```
| 4 |
-----
| 31 |
------
| A |
-----
| 1 |
-----

Popped item = 4

| 31 |
------
| A |
-----
| 1 |
-----
```

2.2.1 Reverse a string using stack

Time complexity: $O(N)$
Space complexity: $O(N)$

```
def reverse_string(input_str):
    stack = Stack()
    for x in input_str:
        stack.push(x)

    rev_str = ""
```

```
    while not stack.is_empty():
        rev_str += stack.pop()

    return rev_str
```

2.2.2 Convert decimal to binary using stack

1. Divide the number by 2 through % (modulus operator) and store the remainder in array.

2. Divide the number by 2 through // (integer division operator).

3. Repeat the above two steps until the number is greater than zero.

4. Print the array in reverse order.

Time complexity: $O(N)$
Space complexity: $O(N)$

```
def convert_binary(dec_num):
    '''
    Example : 242
    --------------
    242 / 2 = 121 -> 0
    121 / 2 = 60 -> 1
    60 / 2 = 30  -> 0
    30 / 2 = 15  -> 0
    15 / 2 = 7   -> 1
    7 / 2 = 3    -> 1
    3 / 2 = 1    -> 1
    1 / 2 = 0    -> 1
    '''
    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())

    return int(bin_num)
```

2.2.3 Check the balancing and ordering of parentheses in an expression

1. Declare a character stack.
2. Now traverse the expression string exp.
   (a) If the current character is a starting bracket then push it to stack.
   (b) If the current character is a closing bracket then pop from stack. If the popped closing bracket matches with the starting bracket then fine else brackets are not balanced.
3. After complete traversal, if there is some starting bracket left in stack then brackets are not balanced.

```
def is_paren_balanced(paren_string):
    s = Stack()
    is_balanced = True
    index = 0

    while index < len(paren_string) and is_balanced:
        paren = paren_string[index]
        if paren in "([{":
            s.push(paren)
        elif paren in ")}]":
            if s.is_empty():
                is_balanced = False
            else:
                top = s.pop()
                if not is_match(top, paren):
                    is_balanced = False
        index += 1

    if s.is_empty() and is_balanced:
        return True
    else:
        return False

def is_match(p1, p2):
    '''
    Compares the starting bracket p1 with the closing bracket p2.
        In case of match it returns True else False.
    '''
    if p1 == "(" and p2 == ")":
        return True
    elif p1 == "{" and p2 == "}":
        return True
    elif p1 == "[" and p2 == "]":
```

```
        return True
    else:
        return False

if __name__ == "__main__":
    print('[a+b][c+d]:', is_paren_balanced("[a+b][c+d]"))
    print('[a+b][c+}d]:', is_paren_balanced("[a+b][c+}d]"))
```

Output

```
[a+b][c+d]: True
[a+b][c+}d]: False
```

2.3 Queue

Queue is an abstract data type or a linear data structure in which the addition of items takes place from one end called the **REAR**(also called **tail**), and the removal of existing items takes place from the other end called as **FRONT**(also called **head**).
This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first. The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.

Applications of Queue:

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.
- Phone answering system: The person who calls first gets a response first from the phone answering system.

```
class Queue:
    # Constructor
    def __init__(self):
        self.items = []

    # Check whether the queue is empty or not
    def is_empty(self):
        return self.items == []

    # Insert new item to the rear of the queue
    def enqueue(self, item):
        self.items.insert(0, item)

    # Removes the front item from the queue and returns it
    def dequeue(self):
        return self.items.pop()

    # Returns the number of items in the queue
    def size(self):
        return len(self.items)

    # Display the queue
    def display(self):
        for i in self.items:
            print('| {} '.format(i), end='')
        print('|')
```

```
if __name__ == "__main__":
    q = Queue()
    q.enqueue('hello')
    q.enqueue('dog')
    q.enqueue(3)
    q.display()

    print('Removed item =', q.dequeue())
    q.display()

    q.enqueue(61)
    q.display()
    print('Queue size =', q.size())
```

Output

```
| 3 | dog | hello |
Removed item = hello
| 3 | dog |
| 61 | 3 | dog |
Queue size = 3
```

Time complexity

Access: $O(n)$
Search: $O(n)$
Insertion: $O(1)$
Deletion: $O(1)$

2.3.1 Priority queue

A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue.

The difference between priority queue and normal queue is that in a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are accessed on the basis of priority. The element with the highest priority is removed first. Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues which we will discuss in section 2.7.

2.3.2 Deque

Deque or *double ended queue* is a generalized version of queue data structure that allows insert and delete at both ends.
What makes a deque different is the nonrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

```
class Deque:
    # Constructor
    def __init__(self):
        self.items = []

    # Check whether the deque is empty or not
    def is_empty(self):
        return self.items == []

    # Insert new item at the front of the deque
    def add_front(self, item):
        self.items.append(item)

    # Insert new item to the rear of the deque
    def add_rear(self, item):
        self.items.insert(0,item)

    # Removes the front item from the deque and returns it
    def remove_front(self):
        return self.items.pop()

    # Removes the rear item from the deque and returns it
    def remove_rear(self):
        return self.items.pop(0)

    # Returns the number of items in the deque
    def size(self):
        return len(self.items)

    # Display the deque
    def display(self):
        for i in self.items:
            print('| {} '.format(i), end='')
        print('|')

if __name__ == "__main__":
    d = Deque()
    d.add_rear(4)
```

```
    d.display()
    d.add_rear('dog')
    d.display()
    d.add_front('cat')
    d.display()
    d.add_front(45)
    d.display()
    print('Deque size =', d.size())

    d.display()
    print('Removed item from rear =', d.remove_rear())
    d.display()
    print('Removed item from front =', d.remove_front())
    d.display()
```

Output:

```
| 4 |
| dog | 4 |
| dog | 4 | cat |
| dog | 4 | cat | 45 |
Deque size = 4
| dog | 4 | cat | 45 |
Removed item from rear = dog
| 4 | cat | 45 |
Removed item from front = 45
| 4 | cat |
```

2.3.3 Palindrome checker using deque

An interesting problem that can be easily solved using the deque data structure is the classic palindrome problem. A palindrome is a string that reads the same forward and backward, for example, radar, madam etc.

```
def palindrome_checker(input_str):
    d = Deque()
    is_palindrome = True

    for ch in input_str:
        d.add_rear(ch)

    while d.size() > 1 and is_palindrome:
        first = d.remove_front()
        last = d.remove_rear()
        if first != last:
            is_palindrome = False
```

```
    return is_palindrome

if __name__ == "__main__":
   print(palindrome_checker("technique"))
   print(palindrome_checker("radar"))
```

Output

```
False
True
```

2.3.4 Python collection.deque module

Deque can be implemented in python using the "collections" module.

```
import collections

# Initializing deque
d = collections.deque([1,2,3])
print (d)

# append() inserts element at right end
d.append(4)
print ('Insert 4 at right:', d)

# appendleft() inserts element at left end
d.appendleft(6)
print ('Insert 6 at left:', d)

# pop() deletes element from right end
d.pop()
print ('Delete from right:', d)

# popleft() to deletes element from left end
d.popleft()
print ('Delete from right:', d)
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
print('~'*40)
d = collections.deque([1, 2, 3, 3, 4, 2, 4])
print (d)

# index(element, beg, end) : This function returns the first index
# of the value element, starting searching from beg till end index.
print ("Number 4 first occurs at: {}".format(d.index(4, 2, 5)))

# insert(i, a) insert the value i at index a
d.insert(4, 3)
```

```
print ("After inserting 3 at 5th position: \n{}".format(d))

# count() counts the occurrences of 3
print ("Count of 3 : {}".format(d.count(3)))

# remove() removes the first occurrence of 3
d.remove(3)
print ("After deleting first occurrence of 3: \n{}".format(d))
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
print('~'*40)
d = collections.deque([1, 2, 3,])
print(d)
# extend() to add multiple numbers to right end
d.extend([4,5,6])
print ("The deque after extending deque at end: ")
print (d)

# extendleft() add multiple numbers to left end
d.extendleft([7,8,9])
print ("The deque after extending deque at beginning: ")
print (d)

# rotate(n) to rotates the deque by n to the right
d.rotate(-3) # rotates to the left by 3

# printing modified deque
print ("The deque after rotating deque: ")
print (d)
```

Output

```
deque([1, 2, 3])
Insert 4 at right: deque([1, 2, 3, 4])
Insert 6 at left: deque([6, 1, 2, 3, 4])
Delete from right: deque([6, 1, 2, 3])
Delete from right: deque([1, 2, 3])
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
deque([1, 2, 3, 3, 4, 2, 4])
Number 4 first occurs at: 4
After inserting 3 at 5th position:
deque([1, 2, 3, 3, 3, 4, 2, 4])
Count of 3 : 3
After deleting first occurrence of 3:
deque([1, 2, 3, 3, 4, 2, 4])
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
deque([1, 2, 3])
The deque after extending deque at end:
deque([1, 2, 3, 4, 5, 6])
The deque after extending deque at beginning:
```

```
deque([9, 8, 7, 1, 2, 3, 4, 5, 6])
The deque after rotating deque:
deque([1, 2, 3, 4, 5, 6, 9, 8, 7])
```

2.4 Tree

A Tree is a Data structure in which data items are connected using references in a hierarchical manner. Each Tree consists of a root node from which we can access each element of the tree. Starting from the root node, each node contains zero or more nodes connected to it as children. A tree must have some properties so that we can differentiate from other data structures.

- The numbers of nodes in a tree must be a finite and nonempty set.
- There must exist a path to every node of a tree i.e., every node must be connected to some other node.
- There must not be any cycles in the tree. It means that the number of edges is one less than the number of nodes.

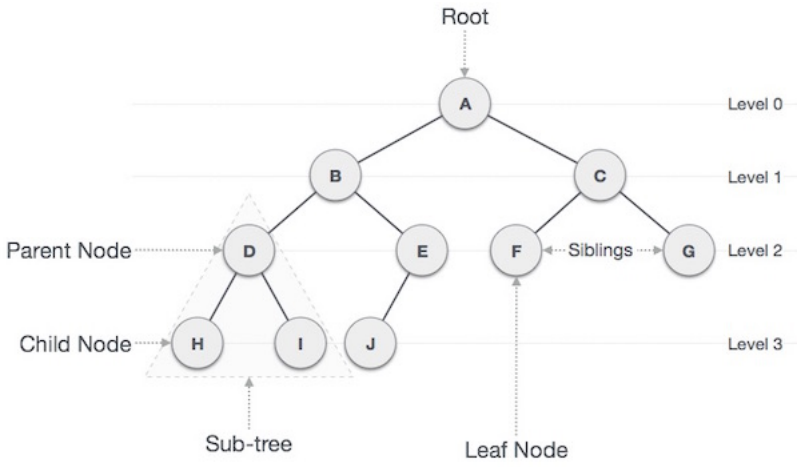


Figure 2.6: Tree data structure

- **Root Node**: Root node is the topmost node of a tree. It is always the first node created while creating the tree and we can access each element of the tree starting from the root node.
- **Parent node**: The parent of any node is the node which references the current node.
- **Child node**: Child nodes of a parent node are the nodes at which the parent node is pointing using the references.
- **Edge**: The reference through which a parent node is connected to a child node is called an edge.
- **Leaf node**: These are those nodes in the tree which have no children.

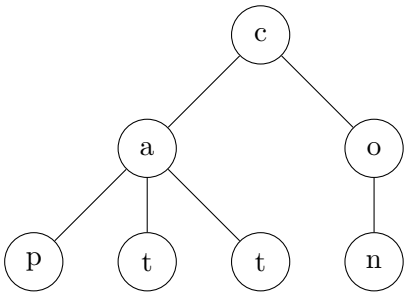


Figure 2.7: A tree used for representing as an array

- **Siblings**: Nodes with the same parent are called siblings.
- **Level**: Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Height**: The height of a node is the number of edges on the longest path from the node to a leaf.
- **Height of tree**: Height of a tree is the height of its root.
- **Depth**: The depth of a node is the number of edges on the path from the root to the node.
- **Node degree**: It is the maximum number of children a node has.
- **Tree degree**: Tree degree is the maximum of the node degrees.

We can use arrays to represent a hierarchical structure by making array of arrays. For example, for the tree 2.7, the corresponding array will be:

```
T = ['c', ['a', ['p'], ['n'], ['t']], ['o', ['n']]]
```

2.4.1 Binary tree

A binary tree is a tree data structure in which each node can have a maximum of 2 children. It means that each node in a binary tree can have either one, or two or no children. We can also classify a binary tree into different categories.

- **Full binary tree**: A binary tree in which every node has 2 children except the leaves is known as a full binary tree.
- **Complete binary tree**: A binary tree in which the last level may not be completely filled and the bottom level is filled from left to right.

- **Perfect binary tree**: In a perfect binary tree, each leaf is at the same level and all the interior nodes have two children.

2.4.2 Depth First Search of a Binary tree

DFS (Depth-first search) is technique used for traversing a tree or graph. Here backtracking is used for traversal. In this traversal first the deepest node is visited and then backtracks to it's parent node if no sibling of that node exist.
There are three ways we can implement the DFS for a Binary tree.

Pre-order traversal

1. Visit the root.
2. Traverse the left subtree, i.e., call Pre-order on left-subtree resursively.
3. Traverse the right subtree, i.e., call Pre-order on right-subtree recursively.

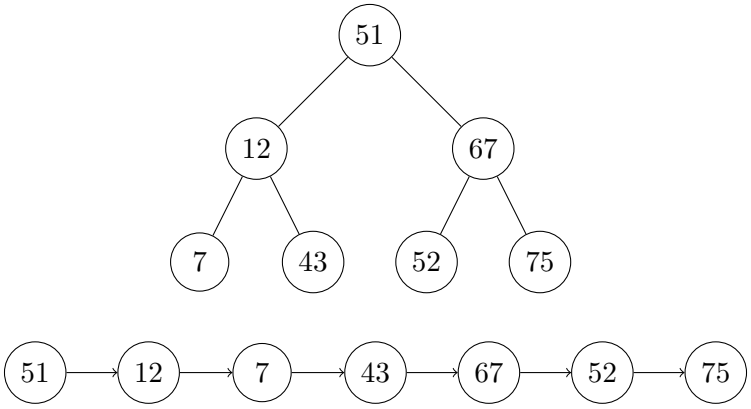


Figure 2.8: Pre-order traversal

In-order traversal

1. Traverse the left subtree, i.e., call In-order on left-subtree resursively.
2. Visit the root.
3. Traverse the right subtree, i.e., call In-order on right-subtree recursively.

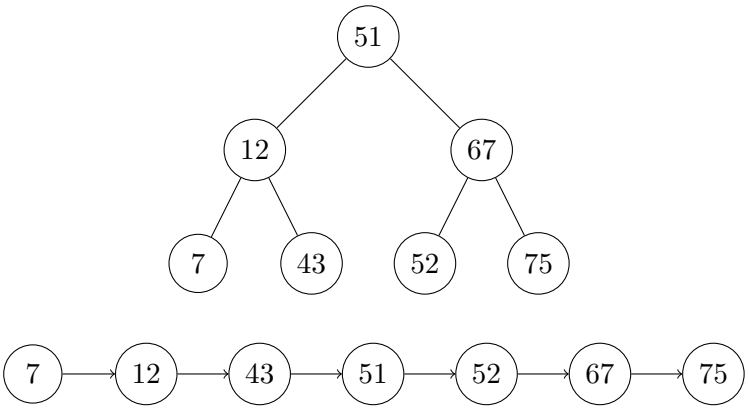


Figure 2.9: in-order traversal

Post-order traversal

1. Traverse the left subtree, i.e., call Post-order on left-subtree resursively.
2. Traverse the right subtree, i.e., call Post-order on right-subtree recursively.
3. Visit the root.

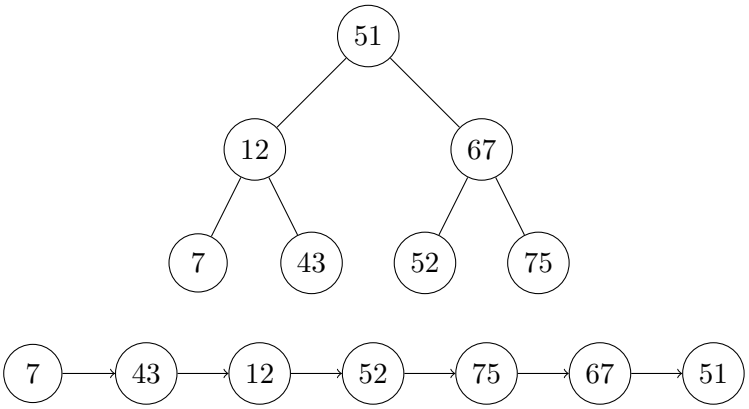


Figure 2.10: Post-order traversal

Time complexity: $O(n)$
Space complexity: $O(h)$ where h is the height (maximum depth from root) of the tree. Even if we aren't using any additional data structures, we are using a recursive function, which means that we have to have a look on the maximum number of calls that can be on the call stack, and in this case, it represents the height h of the tree. This is our *space complexity.*

```
class Tree_node:
   def __init__(self, data, left = None, right = None):
      self.data = data
      self.left = left
      self.right = right

def dfsPreorder(root):
   if root is None:
      return
   print(root.data, end=" ")
   dfsPreorder(root.left)
   dfsPreorder(root.right)

def dfsInorder(root):
   if root is None:
      return
   dfsInorder(root.left)
   print(root.data, end=" ")
   dfsInorder(root.right)

def dfsPostorder(root):
   if root is None:
      return
   dfsPostorder(root.left)
   dfsPostorder(root.right)
   print(root.data, end=" ")

def main():
   # Creating the BST
   node_51 = Tree_node(51) # root node of the tree
   node_12 = Tree_node(12)
   node_67 = Tree_node(67)
   node_7 = Tree_node(7)
   node_43 = Tree_node(43)
   node_52 = Tree_node(52)
   node_75 = Tree_node(75)
   node_51.left = node_12
   node_51.right = node_67
   node_12.left = node_7
   node_12.right = node_43
   node_67.left = node_52
   node_67.right = node_75

   # Pre-order traversal
   dfsPreorder(node_51)
   print("\n")
   # In-order traversal
   dfsInorder(node_51)
   print("\n")
```

```
    # Post-order traversal
    dfsPostorder(node_51)

if __name__ == "__main__":
    main()
```

Output

```
51 12 7 43 67 52 75
7 12 43 51 52 67 75
7 43 12 52 75 67 51
```

2.4.3 Breadth First Search of a Tree

Breadth-first search (BFS) also known as *Level-order traversal* is a tree or graph traversal algorithm that starts traversing from the root node and explores all the child nodes.

Using Queue (Iterative approach): For each node, first the node is visited and then it's child nodes are put in a FIFO queue.

1. Create an empty queue.
2. Push root node into queue.
3. Loop while queue is not empty:
   (a) Pop a node from queue.
   (b) Print the associated data.
   (c) Push the node's children (first left then right children for binary tree) into queue if any.

Time complexity: $O(n)$
Space complexity: $O(n)$

Using Queue (Recursive approach):

1. If the queue is not empty do the following:
   (a) Pop a node from queue.
   (b) Print the associated data.
   (c) Push the node's children (first left then right children for binary tree) into queue if any.
2. Recursively call the function with the new state of queue.

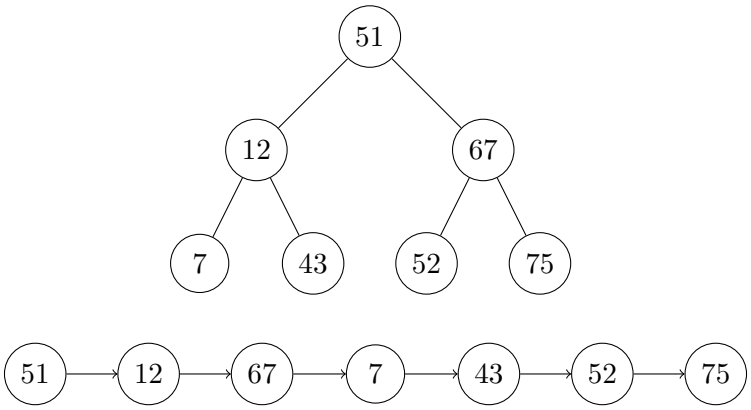


Figure 2.11: Breadth First Search (BFS)

Time complexity: $O(n)$
Space complexity: $O(n)$

```
class Tree_node:
    def __init__(self, data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right

def LevelOrder(root):
    # Base Case
    if root is None:
        return

    # Create an empty queue
    queue = []

    # Enqueue Root
    queue.append(root)

    while(len(queue) > 0):
        # Remove the first node from queue and print the associated
            data
        node = queue.pop(0)
        print(node.data, end=" ")

        # Enqueue left child
        if node.left is not None:
            queue.append(node.left)

        # Enqueue right child
        if node.right is not None:
```

```
            queue.append(node.right)

def LevelOrderRecursive(root, queue):
   # Base Case
    if root is None:
        return

    if len(queue) == 0:
        return
    else:
        # Remove the first node from queue and print the associated
            data
        node = queue.pop(0)
        print(node.data, end=" ")

        # Enqueue left child
        if node.left is not None:
            queue.append(node.left)

        # Enqueue right child
        if node.right is not None:
            queue.append(node.right)

    LevelOrderRecursive(root, queue)

def main():
    # Creating the BST
    node_51 = Tree_node(51) # root node of the tree
    node_12 = Tree_node(12)
    node_67 = Tree_node(67)
    node_7 = Tree_node(7)
    node_43 = Tree_node(43)
    node_52 = Tree_node(52)
    node_75 = Tree_node(75)
    node_51.left = node_12
    node_51.right = node_67
    node_12.left = node_7
    node_12.right = node_43
    node_67.left = node_52
    node_67.right = node_75

    # Level-order traversal (Iterative approach)
    LevelOrder(node_51)
    print()
    # Level-order traversal (Recursive approach)
    LevelOrderRecursive(node_51, [node_51])

if __name__ == "__main__":
    main()
```

Output

```
51 12 67 7 43 52 75
51 12 67 7 43 52 75
```

2.5 Binary Search Tree (BST)

A Binary search Tree (BST) is one data structure that supports faster searching, rapid sorting and easy insertion and deletion. A BST is a sorted version of Binary tree. A tree whose elements have at most 2 children is called a binary tree. The ordering of BST is also referred to as BST property. A BST must have the following properties:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Duplicate keys are not allowed.

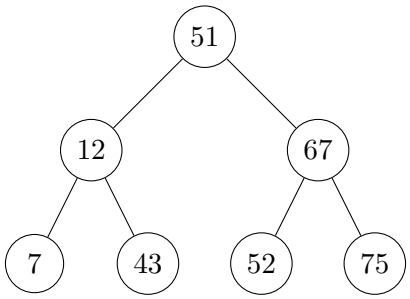


Figure 2.12: Binary swearch Tree

Advantages of Binary search tree

1. Searching an element in the Binary search tree is easy and fast ($\sim O(\log n)$) as we always have a hint that which subtree has the desired element.
2. As compared to array and linked lists, insertion and deletion operations are faster ($\sim O(\log n)$) in BST.

Disdvantages of Binary search tree

1. The shape of the binary search tree totally depends on the order of insertions, and it can degenerate.

2. The main disadvantage is that we should always implement a balanced binary search tree - AVL tree, Red-Black tree, Splay tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.

3. Duplicate values cannot be inserted.

2.5.1 Insertion operation

Iterative method:

1. Set the `currentNode` pointer to the root node.

2. Compare the data to be inserted with the `currentNode`'s data.

3. If the data is smaller, move the `currentNode` pointer to the left child if it exist. If it does not exist, insert it as the left child of the `currentNode`.

4. If the data is larger, move the `currentNode` pointer to the right child if it exist. If it does not exist, insert it as the right child of the `currentNode`.

5. Repeat the steps (2), (3) and (4) till the node is inserted.

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Insert a New Node in a Binary Search Tree
    def insert(self, data):
        # If the self.data is None, create a new node
        if not self.data:
            self.data = data
            return

        # Duplicate value not added
        if self.data == data:
            return

        currentNode = self
        while True:
            if data < currentNode.data:
                # If there is no left child, insert the node on the
                    left
                if not currentNode.left:
                    currentNode.left = BSTNode(data)
                    return
```

```
                # Move the currentNode pointer to the left child
                currentNode = currentNode.left
            else:
                # If there is no right child, insert the node on the
                    right
                if not currentNode.right:
                    currentNode.right = BSTNode(data)
                    return
                # Move the currentNode pointer to the right child
                currentNode = currentNode.right

if __name__ == '__main__':
    root = BSTNode(8)
    root.insert(11)
    root.insert(14)
    root.insert(5)
    root.insert(1)
    root.insert(6)

    root.display()
```

Output [1]

```
  _8_
 /   \
 5  11_
/ \    \
1 6   14
```

Recursive method:

- Starting from the root node, compare the data to be inserted with the root node's data.
- If the data is smaller, recursively call insert for the left subtree if it exists. If it does not exist, then attach it as as the left child.
- If the data is larger, recursively call insert for the right subtree if it exists. If it does not exist, then attach it as as the right child.

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

[1] Here display() is a method of the BSTNode class which is not shown in the code. It will just display the all the nodes in tree structure in the console.

```
    # Insert a New Node in a Binary Search Tree
    def insertRec(self, data):
        # If the self.data is None, create a new node
        if not self.data:
            self.data = data
            return

        # Duplicate value not added
        if self.data == data:
            return

        # If the given data is less than the self node,
        # Then if self.left exist, recur for the left subtree
        # Else add the node as the left child to the self node.
        if data < self.data:
            if self.left:
                self.left.insertRec(data)
                return
            self.left = BSTNode(data)
        # If the given data is less than the self node,
        # Then if self.right exist, recur for the right subtree
        # Else add the node as the right child to the self node.
        else:
            if self.right:
                self.right.insertRec(data)
                return
            self.right = BSTNode(data)

if __name__ == '__main__':
    root = BSTNode(8)
    root.insertRec(11)
    root.insertRec(14)
    root.insertRec(5)
    root.insertRec(1)
    root.insertRec(6)

    root.display()
```

Output [2]

```
  _8_
 /   \
 5  11_
/ \    \
1 6   14
```

[2]Here display() is a method of the BSTNode class which is not shown in the code. It will just display the all the nodes in tree structure in the console.

2.5.2 Search operation

Iterative method:

- Set the `currentNode` pointer to the root node.
- Compare the data to be searched with the `currentNode`'s data.
- If the data is smaller, move the `currentNode` pointer to the left child.
- If the data is larger, move the `currentNode` pointer to the right child.
- Repeat the steps (2), (3) and (4) till the searched node is found or `currentNode` is Null.
- Return the `currentNode`.

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Find a Node in a BST
    def findNode(self, data):
        # If the self.data is None, return None
        if not self.data:
            return None

        currentNode = self
        while data != currentNode.data:
            # If the data is less than the currentNode's data, move
                the currentNode pointer to the left child
            if data < currentNode.data:
                currentNode = currentNode.left
            # If the data is greater than the currentNode's data,
                move the currentNode pointer to the right child
            else:
                currentNode = currentNode.right
            # If no match
            if currentNode is None:
                return None
        # In case the self node contains the searched data, the
            while loop will be skipped altogether and the self node
            will be returned.
        return currentNode

if __name__ == '__main__':
    root = BSTNode(8)
```

```
    root.insertRec(11)
    root.insertRec(14)
    root.insertRec(5)
    root.insertRec(6)
    root.insertRec(1)

    foundNode = root.findNode(7)
    if foundNode is None:
        print("Could not find the node.")
    else:
        print("Found the node.")
```

Output

```
Could not find the node.
```

Recursive method:

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Find a Node in a BST
    def findNodeRec(self, data):
        # If the self.data is None, return None
        if not self.data:
            return None

        # If the data is less than the root's data, recursively call
            the left subtree
        if data < self.data:
            if self.left:
                return self.left.findNodeRec(data)
            return None
        # If the data is greater than the root's data, recursively
            call the right subtree
        elif data > self.data:
            if self.right:
                return self.right.findNodeRec(data)
            return None
        # Return self if self.data matches with the data
        else:
            return self

if __name__ == '__main__':
    root = BSTNode(8)
```

```
    root.insertRec(11)
    root.insertRec(14)
    root.insertRec(5)
    root.insertRec(6)
    root.insertRec(1)

    foundNode = root.findNodeRec(7)
    if foundNode is None:
        print("Could not find the node.")
    else:
        print("Found the node.")
```

Output

```
Found the node.
```

2.5.3 Finding the min and max value

The left-most leaf node of the tree contains the minimum value while the right-most leaf node of the tree contains the maximum value.

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Get the Minimum Value in a BST
    # The left most value is the Smallest Value
    def getMinValue(self):
        currentNode = self
        while currentNode.left is not None:
            currentNode = currentNode.left
        return currentNode.data

    # Get the Maximum Value in a BST
    # The right most value is the Maximum Value
    def getMaxValue(self):
        currentNode = self
        while True:
            if not currentNode.right:
                return currentNode.data
            currentNode = currentNode.right

if __name__ == '__main__':
    root = BSTNode(8)
    root.insertRec(11)
```

```
    root.insertRec(14)
    root.insertRec(5)
    root.insertRec(6)
    root.insertRec(1)

    print('Min value:', root.getMinValue())
    print('Max value:', root.getMaxValue())
```

Output

```
Min value: 1
Max value: 14
```

2.5.4 Height and size of BST

The *height/depth* of a binary tree is defined as the length of the longest path from its root node to a leaf.
The *size* of a binary tree is defined as the number of nodes in the tree.

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Computation of height/depth of a binary tree
    def tree_height(self):
        if self.data is None:
            return 0
        if self.left:
            height_of_left_subtree = self.left.tree_height()
        else:
            height_of_left_subtree = 0
        if self.right:
            height_of_right_subtree = self.right.tree_height()
        else:
            height_of_right_subtree = 0
        return 1 + max(height_of_left_subtree,
            height_of_right_subtree)

    # Computation of number of nodes in a binary tree
    def tree_size(self):
        if self.data is None:
            return 0
        if self.left:
            size_of_left_subtree = self.left.tree_size()
        else:
```

```
            size_of_left_subtree = 0
        if self.right:
            size_of_right_subtree = self.right.tree_size()
        else:
            size_of_right_subtree = 0
        return 1 + size_of_left_subtree + size_of_right_subtree

if __name__ == '__main__':
    root = BSTNode(8)
    root.insert(11)
    root.insert(14)
    root.insert(5)
    root.insert(1)
    root.insert(6)
    root.insert(9)
    root.insert(16)

    root.display()

    print('Height of the tree:', root.tree_height())
    print('Size of the tree:', root.tree_size())
```

Output

```
  _8__
 /    \
 5   11_
/ \ /  \
1 6 9 14_
         \
         16
Height of the tree: 4
Size of the tree: 8
```

2.5.5 Finding the parent node

1. If the data is found in the root node then there is no parent node for that. So return Null.

2. Create a pointer `currentNode` to point to the root.

3. If the data is smaller than the `currentNode`'s data, move the pointer to the left child after saving the parent node in a separate pointer called `parentNode`.

4. If the data is larger than the `currentNode`'s data, move the pointer to the right child after saving the parent node in a separate pointer called `parentNode`.

5. Repeat the steps (3) and (4) until the `currentNode` contains the given data or `currentNode` points to Null. In the first case return the Node pointed by `parentNode`. In the second case return Null.

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    # Find the parent node
    def findParentNode(self, data):
        if data == self.data:
            return None # Data found at root node, no parent
        currentNode = self
        while data != currentNode.data:
            if data < currentNode.data:
                parentNode = currentNode # Saving the parent node
                currentNode = currentNode.left
            else:
                parentNode = currentNode
                currentNode = currentNode.right
            if currentNode is None: # data is not found
                return None
        return parentNode
if __name__ == '__main__':
    root = BSTNode(8)
    root.insert(11)
    root.insert(14)
    root.insert(5)
    root.insert(1)
    root.insert(6)
    root.insert(9)
    root.insert(16)

    root.display()

    foundNode = root.findParentNode(6)
    if foundNode is None:
        print("Could not find the parent node.")
    else:
        print(f"The data in the parent node is {foundNode.data}.")
```

Output

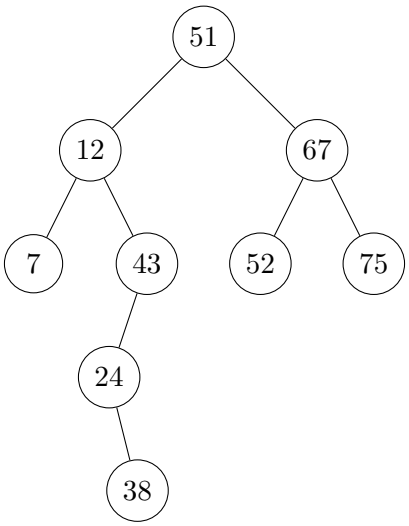
```
  _8__
 /    \
 5   11_
```

```
/ \ /  \
1 6 9 14_
         \
          16
The data in the parent node is 5.
```

2.5.6 Successor and Predecessor

If we sort the nodes of a BST based on the data in increasing order (in-order walk), each node is the *successor* of the preceding node. Similarly, each node is the *predecessor* of the following node.
To find the successor of a node `x`, we do the following:

1. If the right subtree of node `x` is non-empty, then the successor of `x` is just the leftmost node in `x`'s right subtree.
2. If the right subtree of node `x` is empty, then move towards parent nodes, untill the node `x` becomes a part of the left subtree of some parent node `y` for the first time. If such node `y` exist, then it is the successor of node `x`.



In-order walk: $7 \to 12 \to 24 \to 38 \to 43 \to 51 \to 52 \to 67 \to 75$

Here in the picture, the successor of node (12) is node (24). The successor of node (43) is node (51).

In a similar way, to find the predecessor of a node `x`, we do the following:

1. If the left subtree of node `x` is non-empty, then the successor of `x` is just the rightmost node in `x`'s left subtree.

2. If the left subtree of node x is empty, then move towards parent nodes, untill the node x becomes a part of the right subtree of some parent node y for the first time. If such node y exist, then it is the predecessor of node x.

Here in the previous picture, the predecessor of node (43) is node (38). The predecessor of node (24) is node (12).

```
def findSuccessor(x):
    # If the right subtree of node x is non-empty, then the
        successor of x is just the left most node in x's right
        subtree
    if x.right is not None:
        temp = x.right
        while temp.left:
            temp = temp.left
        return temp
    # If the right subtree of node x is empty, then move towards
        parentnodes, untill the node x becomes a part of the left
        subtree of some parent node y for the first time.
    else:
        # We need to have a parent pointer in the BSTNode, else we
            have to traverse the entire tree.
        y = x.parent
        while (y != None) and (x == y.right):
            x = y
            y = y.parent
    return y


def findPredecessor(x):
    # If the right subtree of node x is non-empty, then the
        predecessor of x is just the right most node in x's left
        subtree
    if x.left is not None:
        temp = x.left
        while temp.right:
            temp = temp.right
        return temp
    # If the left subtree of node x is empty, then move towards
        parent nodes, untill the node x becomes a part of the right
        subtree of some parent node y for the first time.
    else:
        # We need to have a parent pointer in the BSTNode, else we
            have to traverse the entire tree.
        y = x.parent
        while (y != None) and (x == y.left):
            x = y
```

```
            y = y.parent
    return y
```

2.5.7 Delete operation

Deletion of an element involves a slightly complicated operation than insertion of an element. Three cases exist with respect to the deletion of a node:

- **Case 1: If the node to be deleted is a leaf node.**

  This is the simplest case. The leaf element can be deleted directly.

- **Case 2: If the node to be deleted has only one child.**

  In this case the element has to be replaced by the child node.

- **Case 3: If the node to be deleted has two children.**

  In this case the largest key in the left subtree (inorder successor) has to be found, which will replace the element to be deleted. Alternatively, the smallest key in the right subtree (inorder predecessor) has to be found, which will replace the element to be deleted.

```
class BSTNode():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def findsuccessor(self):
        # If the right subtree of node current_node is non-empty,
            then the successor of current_node is just the left most
            node in current_node's right subtree
        if self.right is not None:
            temp = self.right
            while temp.left:
                temp = temp.left
        return temp

    def deleteNode(self, data):
        # Base Case
        if self.data is None:
            return self

        # If the data to be deleted is smaller than the self's data
            then it lies in left subtree
        if data < self.data:
```

```
            if self.left:
                self.left = self.left.deleteNode(data)

        # If the data to be delete is greater than the self's data
            then it lies in right subtree
        elif(data > self.data):
            if self.right:
                self.right = self.right.deleteNode(data)

        # If data is same as self's data, then this is the node
        # to be deleted
        else:
            # Case 1: Leaf node
            if (self.left is None) and (self.right is None):
                self = None
                return None

            # Case 2: Node with only one child
            if self.left is None:
                # Left child
                temp = self.right
                self = None
                return temp

            elif self.right is None:
                # Right child
                temp = self.left
                self = None
                return temp

            # Case 3: Node with two children:
            else:
                # Get the inorder successor
                # (smallest node in the right subtree)
                temp = self.findsuccessor()
                # Copy the inorder successor's content to this node
                self.data = temp.data

                # Delete the inorder successor
                self.right = self.right.deleteNode(temp.data)

        return self

if __name__ == '__main__':
    root = BSTNode(8)
    root.insert(11)
    root.insert(14)
    root.insert(5)
    root.insert(1)
```

```
    root.insert(6)
    root.insert(9)
    root.insert(16)

    root.display()

    root.deleteNode(16)
    print('After deleting node 16 (leaf node)')
    root.display()
    root.deleteNode(1)
    print('After deleting node 1 (one child)')
    root.display()
    root.deleteNode(8)
    print('After deleting node 8 (two children)')
    root.display()
```

Output

```
  _8__
 /    \
 5   11_
/ \ /  \
1 6 9 14_
         \
        16
After deleting node 1 (leaf node)
 _8__
/    \
5   11_
 \ /  \
 6 9 14_
        \
       16
After deleting node 14 (one child)
 _8__
/    \
5   11_
 \ /  \
 6 9 16
After deleting node 8 (two children)
 _9_
/   \
5  11_
 \    \
 6   16
```

2.6 Hash table

A hash table (a.k.a Hash map) is an important data structure which stores data based on key-value pairs in such a way as to access the elements faster in $O(1)$ time. Each position of the hash table, often called a **slot**, can hold an item and is indexed by an integer value starting at 0. For example, we will have a slot at index 0, a slot at index 1, a slot at index 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special Python value `None`. The following figure shows a hash table of size $m = 11$. In other words, there are m slots in the table, at index 0 through 10.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

Figure 2.13: Hash initialization

The mapping between an item (a key-value pair) and the slot is called the **hash function**. The hash function will take the key of any item in the collection and return an integer in the range of slot indices, between 0 and $m - 1$. Assume that we have the set of integers 54, 26, 93, 17, 77, and 31 as keys and their respective values as V54, V26, V93, V17, V77, and V31. Our first hash function, sometimes referred to as the **remainder method**, simply takes a key and divides it by the table size, returning the remainder as its hash value

$$h(\text{key}) = \text{key}\%11$$

```
Key     Hash Value
54       54 % 11 = 10
26       26 % 11 = 4
93       93 % 11 = 5
17       17 % 11 = 6
77       77 % 11 = 0
31       31 % 11 = 9
```

Once the hash values have been computed, we can insert each item into the hash table at the index position given bu the computed hash value as shown in Figure 5. Note that 6 out of 11 slots are now occupied. This is referred to as the **load factor** λ, and is commonly defined as

$$\lambda = \frac{\text{number of items}}{\text{table size}}$$

0	1	2	3	4	5	6	7	8	9	10
V77	None	None	None	V26	V93	V17	None	None	V31	V54

Figure 2.14: After placing values into the hash

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is $O(1)$.
this technique is going to work only if each item maps to a unique location in the hash table. For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 ($44\%11 = 0$). Since 77 also had a hash value of 0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a **collision**.

```
# Python program to demonstrate working of Hash-table
CAPACITY = 10
hashTable = [None,] * CAPACITY

def hashFunction(key):
    return key % CAPACITY

def insertData(key, data):
    index = hashFunction(key)
    hashTable[index] = [key, data]

def removeData(key):
    index = hashFunction(key)
    hashTable[index] = 0

if __name__ == "__main__":
    print(hashTable)
    insertData(123, "apple")
    insertData(432, "mango")
    insertData(213, "banana")
    insertData(654, "guava")

    print(hashTable)

    removeData(123)

    print(hashTable)
```

Output

```
[None, None, None, None, None, None, None, None, None, None]
[None, None, [432, 'mango'], [213, 'banana'], [654, 'guava'], None,
    None, None, None, None]
[None, None, [432, 'mango'], 0, [654, 'guava'], None, None, None,
    None, None]
```

2.6.1 Hashing function

The most important part of a hash table is the **hash function**. The hash function converts a specified key into an index for an array that stores all the data. The three primary requirements for a good hash function are as follows:

1. Deterministic: Same keys produce same hash values.
2. Efficiency: Should be computed in $O(1)$ time.
3. Collision resistant: A good hash function may not prevent the collisions completely however it can reduce the number of collisions.
4. Uniform distribution: A non-uniform distribution increases the number of collisions and the cost of resolving them.

There are many hash functions that use numeric or alphanumeric keys.

1. **Division method**: This is the most simple and easiest method to generate a hash value. If k is the key and M is the size of the hash table, then the hash formula is

   $$h(k) = k \mod M$$

   It is best suited that M is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

   ```
   k = 12345
   M = 95
   h(12345) = 12345 mod 95 = 90
   ```

   Pros:

   (a) The division method is very fast since it requires only a single division operation.

   Cons:

   (a) This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.

(b) Sometimes extra care should be taken to chose value of M.

2. **Mid square method**: It involves two steps to compute the hash value: (1) Square the value of the key k i.e. k^2, (2) Extract the middle r digits as the hash value.

$$h(k) = \text{middle } r \text{ digits of } k^2$$

The value r is decided based on the size of the table.

```
k = 60
k^2 = 60 x 60 = 3600
h(60) = 60
```

Pros:

(a) The performance of this method is good as most or all digits of the key value contribute to the result.

(b) The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

(a) The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.

(b) Another disadvantage is that there will be collisions.

3. **Digit folding method**: This method involves two steps:

(a) Divide the key k into a number of parts i.e. $k_1, k_2, k_3, \cdots k_n$, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.

(b) Add the individual parts. The hash value is obtained by ignoring the last carry if any.

```
k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3 = 12 + 34 + 5 = 51
h(k) = 51
```

4. **Multiplication method**: This method involves the following steps:

(a) Choose a constant value A such that $0 < A < 1$.

(b) Multiply the key with A.

(c) Extract the fractional part of kA.

(d) Multiply the result of the above step by the size of the hash table i.e. M.

(e) The resulting hash value is obtained by taking the floor of the result obtained in step 4.

$$h(k) = \lfloor (M(kA \mod 1)) \rfloor$$

```
k = 12345
A = 0.357840
M = 100

h(12345) = floor[ 100 (12345*0.357840 mod 1)]
         = floor[ 100 (4417.5348 mod 1) ]
         = floor[ 100 (0.5348) ]
         = floor[ 53.48 ]
         = 53
```

2.6.2 Hash collision

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a hash collision. We can resolve the hash collision using one of the following techniques.

1. **Collision resolution by chaining**: If a hash function produces the same index for multiple elements, these elements are stored in the same index by using a linked list. If j is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, j contains `NULL`.

2. **Open Addressing**: Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left with `NULL`. Different techniques used in open addressing are:

   (a) *Linear probing*: Linear probing works by finding the next available slot by incrementing one index at a time. A disadvantage to linear probing is the tendency for clustering; items become clustered in the table. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

   (b) *Quadratic probing*: It is a variation to linear probing. Instead of using a constant 'skip' value, we skip the slots using a quadratic formula. This means that if the first hash value is h, the successive values are $h+1^2$, $h+2^2$, $h+3^2$, $h+4^2$, and so on. This technique

addresses the clustering issue and also helps to evenly distribute across available indices.

(c) *Double hashing*: Another great way to uniformly distribute the keys is by having a second hashing function that hashes the result from the original.

2.7 Heap

Heap ('binary heap' to be specific) data structure is a *complete binary tree*[3] that satisfies the heap property, where any given node is

- always greater than its child node(s). This property is also called **max heap property**. The root node is the largest among all other nodes.
- always smaller than the child node(s). This property is also called **min heap property**. The root node is the smallest among all other nodes.

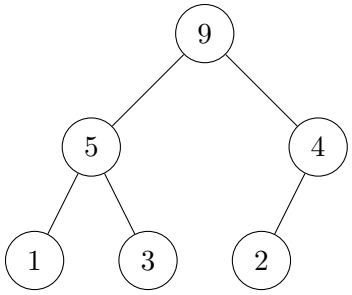


Figure 2.15: Max heap

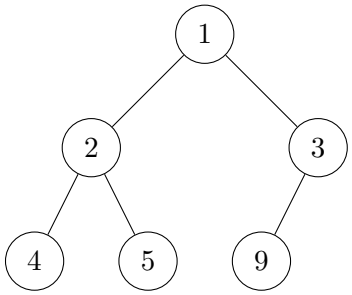


Figure 2.16: Min heap

You can implement a tree structure by a pointer or an array. We choose to use the array implementation like below. In terms of *space complexity*, the array implementation has more benefits than the pointer implementation. In the array representation of a heap, for an element in array index i,

- The parent node would be at index $\lfloor \frac{i-1}{2} \rfloor$.
- The left child would be at index $2i + 1$.

[3] A *complete binary tree* is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

- The right child would be at index $2i + 2$.
- The root node will be at index 0

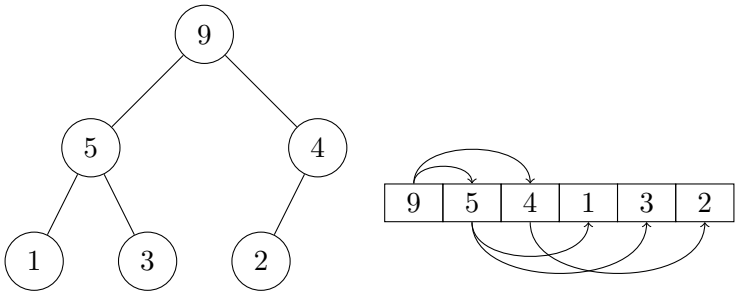


Figure 2.17: Array representation of heap

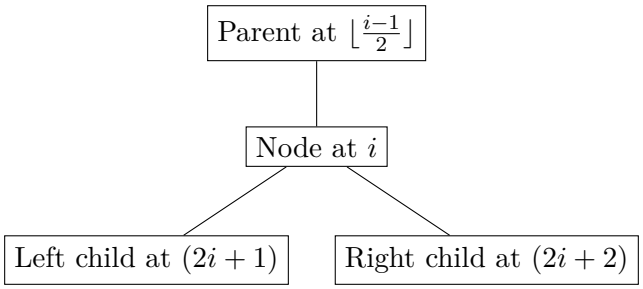


Figure 2.18: Heap relationship

Applications of Heaps

1. *Priority Queues* can be implemented using heaps.
2. Heap is used in heap sort.
3. Heaps are used in implementing various graph algorithms like *Dijkstra's algorithm* and *Prim's algorithm*.
4. If we want to get ordered statistics, heaps serve as a great choice. If we want the kth smallest or largest element, we can pop the heap k times to retrieve them.

2.7.1 Heapify

The process of creating a heap data structure using the binary tree is called heapify. It is used to create a Max-Heap or a Min-Heap. You need two operations to build a Max-heap (Min-heap) from an arbitrary array.

1. `heapify`: to make some node and its descendant nodes meet the heap property.

2. `buildHeap`: to produce a heap from an arbitrary array.

- `buildHeap`:

    1. We are given an array. Firstly we will create a complete binary tree from the array.
    2. Start from the first index of non-leaf node whose index is given by `n/2 - 1` where `n` is the length of the array. We will apply `heapify` on this node.
    3. Perform reverse level order traversal from last non-leaf node and `heapify` each node.

- `heapify`:

    1. Set current element `i` as `largest` (`smallest`).
    2. The index of left child is given by `2i + 1` and the right child is given by `2i + 2`.
        (a) If left child is greater (smaller) than current element (i.e. element at `i`th index), set `leftChildIndex` as `largest` (`smallest`).
        (b) If right child is greater (smaller) than the element in `largest` (`smallest`), set `rightChildIndex` as `largest` (`smallest`).
    3. Swap `largest` (`smallest`) with current element.
    4. Repeat `heapify` to the `largest` (`smallest`) element recursively.

```
'''Max heap implementation'''

def heapify(arr, i):
    # To heapify a subtree rooted with node i which is an index in
        arr.
    largest = i # Initialize largest as root
    leftChildIndex = 2*i + 1 # leftChildIndex = 2*i + 1
    rightChildIndex = 2*i + 2 # rightChildIndex = 2*i + 2

    N = len(arr)

    # If leftChildIndex child is larger than root
    if leftChildIndex < N and arr[leftChildIndex] > arr[largest]:
        largest = leftChildIndex

    # If rightChildIndex child is larger than largest so far
    if rightChildIndex < N and arr[rightChildIndex] > arr[largest]:
        largest = rightChildIndex

    # If largest is not root
    if largest != i:
```

```
        arr[i], arr[largest] = arr[largest], arr[i]

        # Recursively heapify the affected sub-tree
        heapify(arr, largest)


def buildHeap(arr):
    # Function to build a Max-Heap from the given array index of
        last non-leaf node
    N = len(arr)
    startIdx = N // 2 - 1

    # Perform reverse level order traversal from last non-leaf node
        and heapify each node
    for i in range(startIdx, -1, -1):
        heapify(arr, i)

if __name__ == '__main__':
    # Binary Tree Representation of input array
    #            1
    #          /   \
    #        3       5
    #      /  \    /  \
    #    4     6  13  10
    #   / \   / \
    #  9   8  15 17
    arr = [1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17]

    buildHeap(arr)
    print(arr)

    # Final Heap:
    #            17
    #          /   \
    #        15     13
    #       /  \    / \
    #      9    6  5   10
    #     / \  / \
    #    4  8 3  1
```

Output

```
[17, 15, 13, 9, 6, 5, 10, 4, 8, 3, 1]
```

Time complexity: The function `heapify` exchanges two items in an array, which runs in constant time and recursively calls itself. So the *time complexity* of `heapify` will be in proportional to the number of repeats. In the worst case, `heapify` should repeat the operation the height of the tree times. This is because in the worst case, `heapify` will exchange the root nodes with the

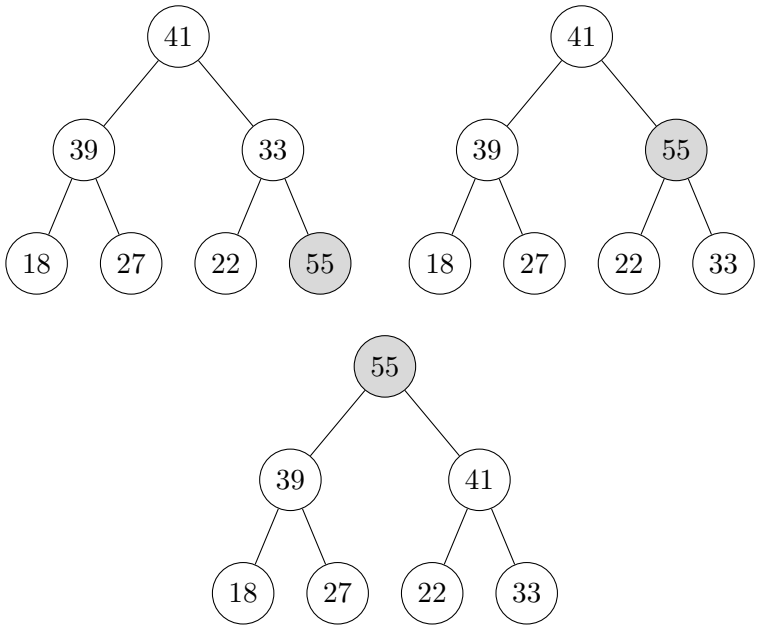


Figure 2.19: Percolation: Bubble up for max-heap

most depth leaf node. Therefore the *time complexity* of `heapify` is $O(\log_2 n)$ time. The function `buildHeap` basically calls `heapify` and makes $O(n)$ such calls. So the overall *time complexity* is $O(n \log_2 n)$.

2.7.2 Percolation: Bubbling up and down

When elements are added or removed, the structure must also fulfil the heap property. This may require some items to swap and “bubble up" to the top of the heap while some other items to “bubble down" to their rightful position in order to keep the structure of the heap. Percolation take $O(\log_2 n)$ time as it needs to swap elements along the height of the tree.

```
def bubble_up(arr, i):
# This function will bubble up the element at index i (for max-heap)
    parentIndex = (i-1)//2
    if parentIndex < 0:
        return
    # If the element is grater than the parent element, swap values
        with the parent
    if arr[i] > arr[parentIndex]:
        arr[i], arr[parentIndex] = arr[parentIndex], arr[i]
        bubble_up(arr, parentIndex)
```

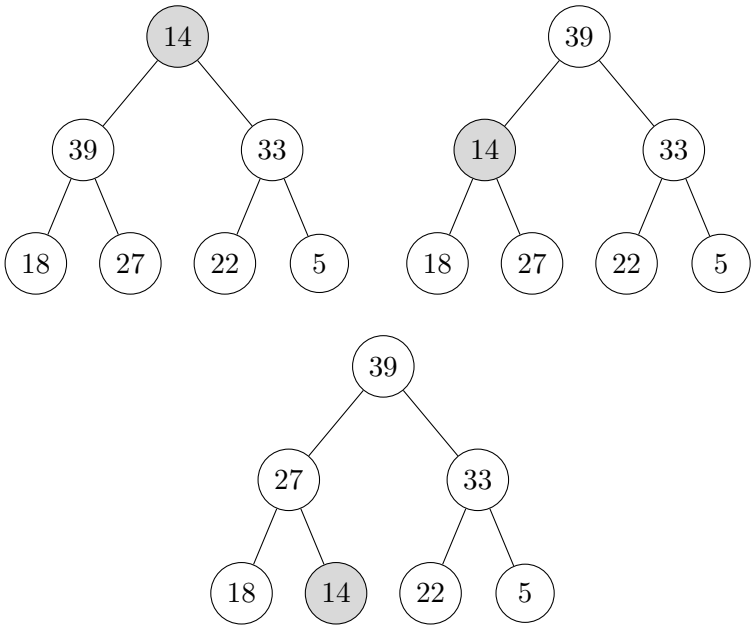


Figure 2.20: Percolation: Bubble down for max-heap

```
def bubble_down(arr, i):
    # This function will bubble down the element at index i (for
        max-heap)
    largest = i # Initialize largest as root
    leftChildIndex = 2*i + 1 # leftChildIndex = 2*i + 1
    rightChildIndex = 2*i + 2 # rightChildIndex = 2*i + 2

    N = len(arr)
    if leftChildIndex > N:
        return

    # If leftChildIndex child is larger than root
    if leftChildIndex < N and arr[leftChildIndex] > arr[largest]:
        largest = leftChildIndex

    # If rightChildIndex child is larger than largest so far
    if rightChildIndex < N and arr[rightChildIndex] > arr[largest]:
        largest = rightChildIndex

    # If largest is not root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]

        # Recursively heapify the affected sub-tree
        bubble_down(arr, largest)
```

```
if __name__ == "__main__":
    arr = [41, 39, 33, 18, 27, 22, 55]
    print('Initial array:', arr)
    bubble_up(arr, 6)
    print('After bubbling up:', arr)
    arr = [14, 39, 33, 18, 27, 22, 5]
    print('Initial array:', arr)
    bubble_down(arr, 0)
    print('After bubbling down:', arr)
    print()
    arr = [17, 15, 2, 13, 9, 6, 5, 10, 4, 8, 16, 3, 1]
    print('Initial array:', arr)
    bubble_up(arr, 10)
    print('After bubbling up:', arr)
    arr = [17, 15, 2, 13, 9, 6, 5, 10, 4, 8, 3, 1]
    print('Initial array:', arr)
    bubble_down(arr, 2)
    print('After bubbling down:', arr)
```

Output

```
Initial array: [41, 39, 33, 18, 27, 22, 55]
After bubbling up: [55, 39, 41, 18, 27, 22, 33]
Initial array: [14, 39, 33, 18, 27, 22, 5]
After bubbling down: [39, 27, 33, 18, 14, 22, 5]

Initial array: [17, 15, 2, 13, 9, 6, 5, 10, 4, 8, 16, 3, 1]
After bubbling up: [17, 16, 2, 13, 15, 6, 5, 10, 4, 8, 9, 3, 1]
Initial array: [17, 15, 2, 13, 9, 6, 5, 10, 4, 8, 3, 1]
After bubbling down: [17, 15, 6, 13, 9, 2, 5, 10, 4, 8, 3, 1]
```

2.7.3 Insert element into heap

1. Insert the new element at the end of the array.
2. Bubble-up the element through the array.

Time complexity: $O(\log_2 n)$

```
def insertNode(arr, element):
    arr.append(element)
    N = len(arr)
    bubble_up(arr, N-1)
```

2.7.4 Delete element from heap

1. Find the index element to be deleted.

2. Swap it with the last element.

3. Delete the last element

4. Bubble-down the element through the array.

Time complexity: $O(\log_2 n)$

```
def deleteNode(arr, element):
    N = len(arr)
    i = 0 # To find the index of the element to be removed
    for i in range(0, N):
        if element == arr[i]:
            break
    # Swap with the last item
    arr[i], arr[N-1] = arr[N-1], arr[i]
    arr.remove(element)
    bubble_down(arr, i)
```

2.7.5 Heap sort

Heap sort is one kind of sorting algorithm using a heap data structure. Note that the order of sort is **ascending for max-heap** while the order is **descending for min-heap**. Let's consider an array `arr` which is to be sorted in ascending order using heap sort.

1. Build a max-heap from the elements of the array `arr`.

2. The root element, that is `arr[0]`, will contain maximum element of `arr`. Swap this element with the last element of `arr`. The last element is already in its correct position.

3. Decrease the length of heap by one which will exclude the last element.

4. Heapify the root element (bubble-down operation of root) again so that we have the highest element moves at the root again.

5. The process is repeated until all the items of the list are sorted.

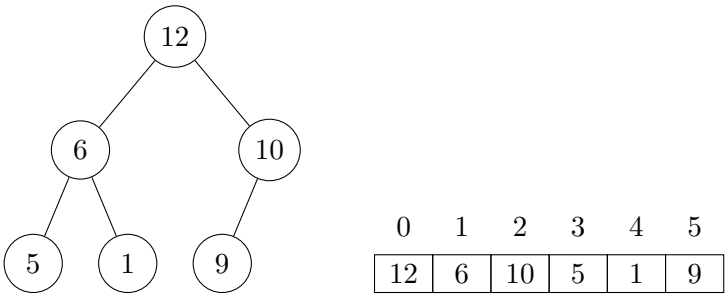


Figure 2.21: Heap sort: Step 1. Initialisation

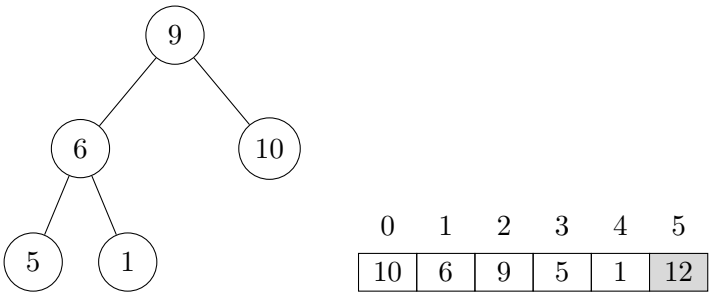


Figure 2.22: Heap sort: Step 2. Swapping 12 with 9 and remove 12 from heap

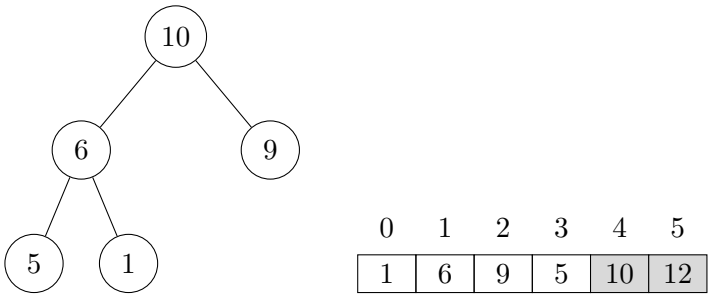


Figure 2.23: Heap sort: Step 3. Bubble-down 9

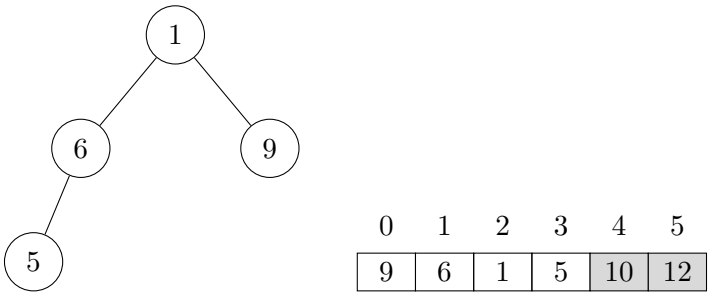


Figure 2.24: Heap sort: Step 4. Swapping 10 with 1 and remove 10 from heap

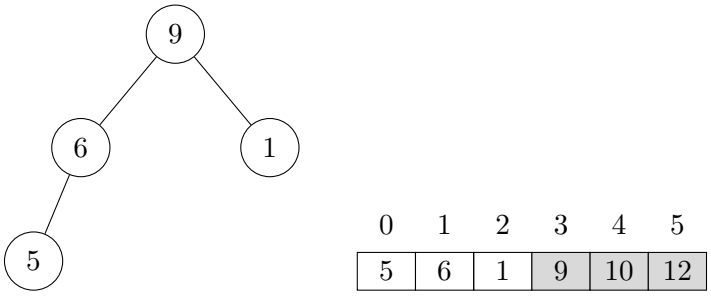


Figure 2.25: Heap sort: Step 5. Bubble-down 1

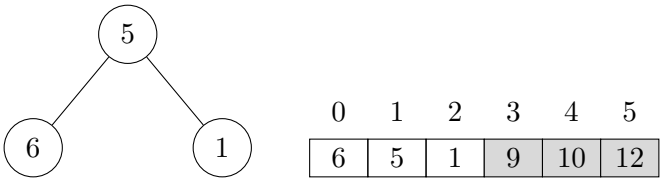


Figure 2.26: Heap sort: Step 6. Swap 9 with 5 and remove 9 from heap

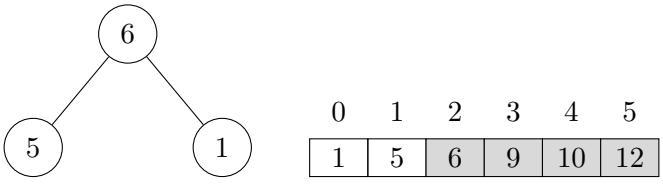


Figure 2.27: Heap sort: Step 7. Bubble-down 5

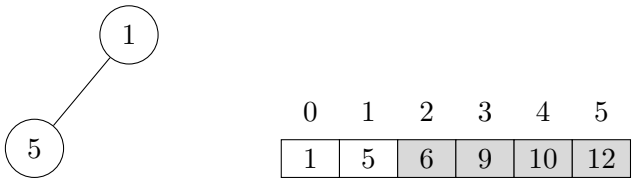


Figure 2.28: Heap sort: Step 8. Swap 6 with 1 and remove 6 from heap

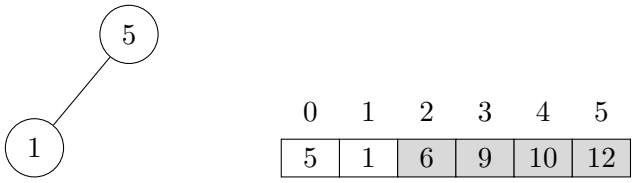


Figure 2.29: Heap sort: Step 9. Bubble down 1

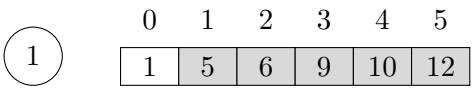


Figure 2.30: Heap sort: Step 10. Swap 5 with 1 and remove 5 from heap

Time complexity: The initial building of heap takes $O(n \log n)$ time. The the bubbline-down operation of root would take $O(\log n)$ time, which is to be performed n times at maximum. So the *time complexity* for heap sort is $O(n \log n)$.
Space complexity: $O(1)$

```
def heapify(arr, i, size):
    # To heapify a subtree rooted with node i which is an index in
        arr.
    # 'size' is size of heap
    largest = i # Initialize largest as root
    leftChildIndex = 2*i + 1 # leftChildIndex = 2*i + 1
    rightChildIndex = 2*i + 2 # rightChildIndex = 2*i + 2

    # If leftChildIndex child is larger than root
    if leftChildIndex < size and arr[leftChildIndex] > arr[largest]:
        largest = leftChildIndex

    # If rightChildIndex child is larger than largest so far
    if rightChildIndex < size and arr[rightChildIndex] >
        arr[largest]:
        largest = rightChildIndex

    # If largest is not root
    if largest != i:
```

```
        arr[i], arr[largest] = arr[largest], arr[i]

        # Recursively heapify the affected sub-tree
        heapify(arr, largest, size)


def buildHeap(arr):
    # Function to build a Max-Heap from the given array index of
        last non-leaf node
    N = len(arr)
    startIdx = N // 2 - 1

    # Perform reverse level order traversal from last non-leaf node
        and heapify each node
    for i in range(startIdx, -1, -1):
        heapify(arr, i, N)


def heapSort(arr):
    # First build a max-heap
    buildHeap(arr)

    N = len(arr)
    for i in range(N-1, 0, -1):
        # Swap the root and last element of heap
        arr[0], arr[i] = arr[i], arr[0]
        # Bubble-down the root element
        heapify(arr, 0, i)


if __name__ == "__main__":
    arr = [24, 56, 23, 9, 45, 18, 60, 107, 45, 27]
    heapSort(arr)
    print(arr)
```

Output

```
[9, 18, 23, 24, 27, 45, 45, 56, 60, 107]
```

2.7.6 Priority queues

A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first (not first-in-first-out rule). However, if elements with the same priority occur, they are served according to their order in the queue. Generally, the value of the element itself is considered for assigning the priority. We can perform two operations on a

priority queue:

1. **add**(element): This function will insert 'element' into the priority queue.
2. **poll**(): This function will remove the highest priority element from the priority queue.

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

- If higher priority corresponds to higher number in the element, then a *max-heap* can be used to implement the priority queue. this is known as *descending order priority queue*.

  Highest priority

18	12	9	7

  Figure 2.31: Descending order priority queue

- If higher priority corresponds to lower number in the element, then a *min-heap* can be used to implement the priority queue. this is known as *ascending order priority queue*.

  Highest priority

7	9	12	18

  Figure 2.32: Ascending order priority queue

Adding an element to a priority queue is equivalent to adding the element into the corresponding heap. Polling of element from the priority queue is basically deleting the root node form the heap.

2.8 Graph

A graph is a common data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them. Generally, a graph G is represented as $G = (V, E)$, where V is set of vertices and E is set of edges. The figure 2.33 is a graph with 5 vertices and 6 edges. This graph can be defined as $G = (V, E)$ where

$$V = \{A, B, C, D, E\}$$

and

$$E = \{(A, B), (A, C)(A, D), (B, D), (C, D), (C, E), (D, E)\}$$

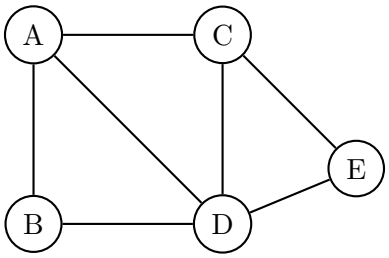


Figure 2.33: Graph data structure

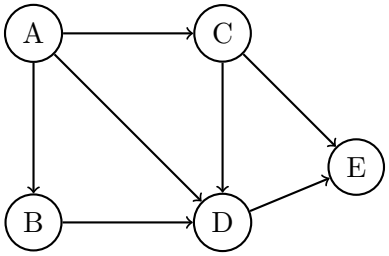


Figure 2.34: Directed graph

2.8.1 Directed and Undirected graph

A graph can be directed or undirected. In an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the figure 2.33 since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from $B \rightarrow A$ as well as $A \rightarrow B$. In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node. A directed graph is shown in the figure 2.34.

2.8.2 Graph terminologies

- **Path**: A path can be defined as the sequence of nodes that are followed in order to reach some terminal node v from the initial node u.

- **Closed path**: A path with sequence of nodes $\{v_0, v_1, \cdots v_N\}$ will be called as closed path if the initial node is same as terminal node i.e. a path will be closed path if $v_0 = v_N$.

- **Simple path**: If all the nodes of the graph are distinct with an exception $v_0 = v_N$, then such path P is called as closed simple path.

- **Cycle**: A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

- **Connected graph**: A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- **Complete graph**: A complete graph is the one in which every node is connected with all other nodes. A complete graph contain $\frac{n(n-1)}{2}$ edges where n is the number of nodes in the graph.
- **Weighted graph**: In a weighted graph, each edge is assigned with some number such as length or weight. The weight of an edge e can be given as $w(e)$ which must be a positive $(+)$ value indicating the cost of traversing the edge.
- **Digraph**: A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- **Loop**: An edge from a vertex to itself.
- **Adjacent nodes**: If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- **Degree of the node**: A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called an *isolated node*.
- **Order**: The number of vertices in the graph.
- **Size**: The number of edges in the graph.

2.8.3 Graph implementations

There are three main ways to implement graphs:

1. **Adjacency matrix**

   In adjacency matrix, we represent the graph $G = (V, E)$ in a two-dimensional array of $n \times n$ elements where $n = |V|$ is the number of nodes. If ijth element a_{ij} of the matrix is 1, this means that there is an edge connecting vertex i with vertex j.

   $$a_{ij} = \begin{cases} 1, & \text{if there is an edge from } i \to j \\ 0, & \text{otherwise} \end{cases}$$

   For weighted graph,

   $$a_{ij} = \begin{cases} w_{ij}, & \text{if there is an edge from } i \to j \\ 0, & \text{otherwise} \end{cases}$$

This approach if very fast for look-ups. Because we can instantly get the value in the cell ij. But it's also very heavy on memory because we have to reserve a space in memory for every possible edge $n \times n$.

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	0
C	1	0	0	1	1
D	1	1	1	0	0
E	0	0	1	1	0

Table 2.1: Adjacency matrix for figure 2.33

2. **Adjacency lists**

   In adjacency lists, an array of lists is used to represent the graph. The size of the array is equal to the number of vertices. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. The adjacency matrix for figure 2.33 is given in the following:

```
A ---> B ---> C ---> D ---> Null
B ---> A ---> D ---> Null
C ---> A ---> D ---> E ---> Null
D ---> A ---> B ---> C ---> E ---> Null
E ---> C ---> D ---> Null
```

3. **Adjacency sets**

   This is a new alternative to adjacency lists. Instead of having an array of linked lists, have it be an array of sets. An entry array[i] represents the set of vertices adjacent to the ith vertex. The adjacency matrix for figure 2.33 is given in the following:

```
[
   {A, B, C, D},
   {B, A, D},
   {C, A, D, E},
   {D, A, B, C, E},
   {E, C, D}
]
```

If m is the number of neighbors of a given vertex, and v is the number of vertices in the graph, then the *time complexity* comparison for different representations are given in the following table.

Operation	Adjacency Matrix	Adjacency List	Adjacency Sets
Edge existence check	$O(1)$	$O(m)$	$O(1)$
Iterating over neighbours	$O(\|V\|)$	$O(m)$	$O(m)$
Adding an edge	$O(1)$	$O(1)$	$O(1)$
Removing an edge	$O(1)$	$O(m)$	$O(1)$

Table 2.2: *Time complexity* comparison

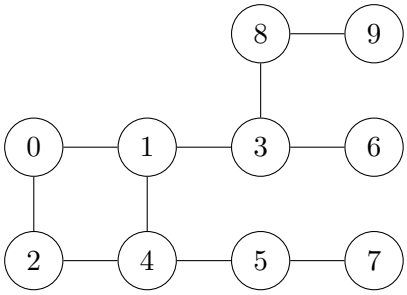


Figure 2.35: Graph for the given DFS python program

2.8.4 Depth First Search (DFS)

DFS (Depth-first search) is technique used for traversing a tree or graph. Here backtracking is used for traversal. In this traversal first the farthest node along the path is visited and then backtracks to the previous node. A standard DFS implementation puts each vertex of the graph into 'visited' category. The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

```
# Iterative approach using a stack
def dfs_iterative(graph, node):
    visited = set()
    # Create a stack for DFS
    stack = []
    # Push the current node.
    stack.append(node)

    while len(stack) != 0:
```

```
        # Pop a vertex from stack
        node = stack.pop()

        # Stack may contain same vertex twice. So we need to print
            the popped item only if it is not visited and add the
            item in the visited list
        if node not in visited:
            print(node, end=' ')
            visited.add(node)

        # Get all adjacent vertices of the popped vertex. If a
            adjacent has not been visited, then push it to the stack.
        for neighbour in graph[node] - visited:
            if neighbour not in visited:
                stack.append(neighbour)

# Recursive approach
def dfs_recursive(graph, node, visited = set()):
    if node in visited:
        return
    else:
        print(node, end=" ")
        visited.add(node)
        for neighbour in graph[node] - visited:
            dfs_recursive(graph, neighbour, visited)


if __name__ == "__main__":
    graph = {
        '0': set(['1', '2']),
        '1': set(['0', '3', '4']),
        '2': set(['0', '4']),
        '3': set(['1', '6', '8']),
        '4': set(['1', '2', '5']),
        '5': set(['4', '7']),
        '6': set(['3']),
        '7': set(['5']),
        '8': set(['3', '9']),
        '9': set(['8'])
    }
    print("Iterative approach")
    dfs_iterative(graph, '0')
    print()
    print("Recursive approach")
    dfs_recursive(graph, '0')
```

Output

```
Iterative approach
```

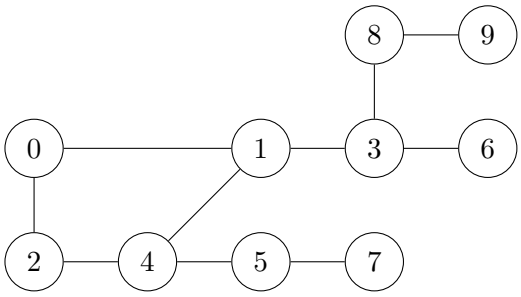


Figure 2.36: Graph for the given BFS python program

```
0 1 4 2 5 7 3 6 8 9
Recursive approach
0 2 4 5 7 1 3 8 9 6
```

Time complexity: $O(|V| + |E|)$
Space Complexity: $O(|V|)$ for the `visited` array

Applications

1. Used to find a path between two vertices.
2. Used to detect cycles in a graph.
3. Used in topological sorting.

2.8.5 Breadth First Search (BFS)

Breadth-first search (BFS) is a graph traversal algorithm that starts traversing from any node and explores all the neighbouring nodes before moving to the neighbour of neighbours. Similar to DFS, the standard BFS implementation puts each vertex of the graph into 'visited' category. The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. The BFS algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

```
# Iterative approach using a queue
def bfs_iterative(graph, node):
```

```
    visited = set()
    # Create a queue for BFS
    queue = []
    # Push the current node.
    queue.insert(0, node)

    while len(queue) != 0:
        # Pop a vertex from queue
        node = queue.pop()

        # queue may contain same vertex twice. So we need to print
            the popped item only if it is not visited and add the
            item in the visited list
        if node not in visited:
            print(node, end=' ')
            visited.add(node)

        # Get all adjacent vertices of the popped vertex. If a
            adjacent has not been visited, then push it to the stack.
        for neighbour in graph[node] - visited:
            if neighbour not in visited:
                queue.insert(0, neighbour)


if __name__ == "__main__":
    graph = {
        '0': set(['1', '2']),
        '1': set(['0', '3', '4']),
        '2': set(['0', '4']),
        '3': set(['1', '6', '8']),
        '4': set(['1', '2', '5']),
        '5': set(['4', '7']),
        '6': set(['3']),
        '7': set(['5']),
        '8': set(['3', '9']),
        '9': set(['8'])
    }
    print("Iterative approach")
    bfs_iterative(graph, '0')
```

Output

```
Iterative approach
0 1 2 3 4 6 8 5 9 7
```

Time complexity: $O(|V| + |E|)$
Space Complexity: $O(|V|)$ for the `visited` array

Applications

1. Used to determine the shortest path and **minimum spanning tree**.
2. Used by search engine crawlers to build indexes of web pages.
3. Used to search on social networks.
4. Used to find available neighbour nodes in peer-to-peer networks such as BitTorrent.

2.8.6 Shortest path

The shortest path from one vertex to another vertex is a path in the graph such that the sum of the weights of the edges that should be travelled is minimum.

Dijkstra's Algorithm

1. Create a list of "distances" d equal to the number of nodes n and initialize each value to infinity.

$$d[v] = \infty, \quad \forall v = 0, 1, 2, \cdots (n-1)$$

2. Set the "distances" to the starting node equal to 0 since the distance from the source node to itself is 0. If v_0 is the index of the starting node, then

$$d[v_0] = 0$$

3. Create a list of "visited" nodes u set to False for each node (since we haven't visited any yet).

$$u[v] = \text{False}, \quad \forall v = 0, 1, 2, \cdots (n-1)$$

4. Loop through n iterations:
   (a) Select the node with index v for which the distance $d[v]$ is minimum and $u[v] = \text{False}$.
   (b) Set that node to visited i.e. $u[v] = \text{True}$.
   (c) Perform **relaxation** to all its adjacent nodes. If σ is the index of an adjacent node, then update the distance $d[\sigma]$ as follows:

$$d[\sigma] = \min\left(d[\sigma], d[v] + l(v, \sigma)\right)$$

   where $l(v, \sigma)$ is the length corresponding to the edge between the nodes at index v and σ.

5. The original "distances" list should now contain the shortest distance to each node or infinity if a node is unreachable from the desired starting node.

N.B. This algorithm requires that there are no negative weight edges in the graph.

Example of Dijkstra's algorithm:

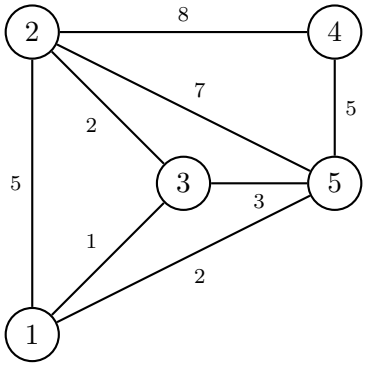


Figure 2.37: Example graph for Dijkstra's algorithm

Vertex	**distances** d	**visited** u
1	0	False
2	∞	False
3	∞	False
4	∞	False
5	∞	False

Table 2.3: Dijkstra's algorithm: Initial step

Vertex	**distances** d	**visited** u
1*	0	True
2	$\min(\infty, 0+5) = 5$	False
3	$\min(\infty, 0+1) = 1$	False
4	∞	False
5	$\min(\infty, 0+2) = 2$	False

Table 2.4: Dijkstra's algorithm: After 1st iteration

Vertex	**distances** d	**visited** u
1	0	True
2	$\min(5, 1+2) = 3$	False
3*	1	True
4	∞	False
5	$\min(2, 1+3) = 2$	False

Table 2.5: Dijkstra's algorithm: After 2nd iteration

Vertex	**distances** d	**visited** u
1	0	True
2	$\min(3, 2+7) = 3$	False
3	$\min(1, 2+2) = 1$	True
4	$\min(\infty, 2+5) = 7$	False
5*	2	True

Table 2.6: Dijkstra's algorithm: After 3rd iteration

Vertex	**distances** d	**visited** u
1	0	True
2*	3	True
3	1	True
4	$\min(7, 3+8) = 7$	False
5	$\min(2, 3+3) = 2$	True

Table 2.7: Dijkstra's algorithm: After 4th iteration

Vertex	**distances** d	**visited** u
1	0	True
2	3	True
3	1	True
4*	7	True
5	2	True

Table 2.8: Dijkstra's algorithm: After 5th iteration

```
def dijkstras(graph, starting_node):
    n = len(graph)

    # Step 1: Initialize distance list (dict) as all infinities
    distances = {x: float("Inf") for x in graph.keys()}
```

```
    # Step 2: Set the distance for the starting_node to be 0
    distances[starting_node] = 0

    # Step 3: Initialize list (dict) of visited nodes
    visited = {x: False for x in graph.keys()}

    # Step 4: Loop through n iterations
    for _ in range(n):

        # Step 4a: Select the node for which the distance is minimum
            and is not visited before.
        u = -1
        for i in graph.keys():
            if not visited[i] and (u == -1 or distances[i] <
                distances[u]):
                u = i
        # All the nodes have been visited or we can't reach this node
        if distances[u] == float("Inf"):
            break

        # Step 4b: Set the node as visited
        visited[u] = True

        # Step 4c: Performing relaxation to all adjacent nodes
        for v, l in graph[u]:
            if distances[u] + l < distances[v]:
                distances[v] = distances[u] + l
    return distances


if __name__ == "__main__":
    # Create our graph using an adjacency list representation
    # each "node" in our list should be a node index and a distance
    graph = {
        '0': [('1', 1)],
        '1': [('0', 1), ('2', 2), ('3', 3)],
        '2': [('1', 2), ('3', 1), ('4', 5)],
        '3': [('1', 3), ('2', 1), ('4', 1)],
        '4': [('2', 5), ('3', 1)]
    }
    print(dijkstras(graph, '1'))
```

Output

```
{'0': 1, '1': 0, '2': 2, '3': 3, '4': 4}
```

Time complexity: We have a loop with $n = |V|$ iterations. On each iteration, we need to find the minimum distance node. Then we need to update the distance of all the adjacent node. In the worst case this operation will take

$O(|V|)$ time. Thus overall we have the *time complexity* $O(|V|^2)$. The *space complexity* is $O(|V|)$, since we have to maintain two lists d and u each of length $|V|$.

Bellman-Ford algorithm

Bellman-Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

The Bellman-Ford algorithm can also be used to check if the graph contains a *cycle with negative length*. In this case, any path that contains the cycle can be shortened infinitely many times, so the concept of a shortest path is not meaningful.

1. Create a list of "distances" d equal to the number of nodes n and initialize each value to infinity.

$$d[v] = \infty, \quad \forall v = 0, 1, 2, \cdots (n-1)$$

2. Set the "distances" to the starting node equal to 0 since the distance from the source node to itself is 0. If v_0 is the index of the starting node, then

$$d[v_0] = 0$$

3. Loop through $(n-1)$ iterations:

   (a) Loop through each edge $(v_{\text{source}}, v_{\text{destination}})$ with weight w

   - **Relax** the edge:

$$d[v_{\text{destination}}] = \min\left(d[v_{\text{destination}}], w + d[v_{\text{source}}]\right)$$

4. Detect negative cycle. We loop through each edge again. If we found for some edge,

$$d[v_{\text{destination}} < w + d[v_{\text{source}}]$$

   then there is a negative edge and we halt the algorithm.

5. The original "distances" list should now contain the shortest distance to each node or infinity if a node is unreachable from the desired starting node.

```
def bellman_ford(graph, n, starting_node):
    # Step 1: fill the distance array and predecessor array
    distances = [float("Inf")] * n
```

```
    # Mark the distance of the source vertex as zero
    distances[starting_node] = 0

    # Step 2: Relax edges |V|-1 times
    for _ in range(n-1):
        for s, d, w in graph:
            if distances[s] != float("Inf") and distances[s] + w <
                distances[d]:
                distances[d] = distances[s] + w

    # Step 3: Detect negative cycle
    # If value changes then we have a negative cycle in the graph
        and we cannot find the shortest distances
    for s, d, w in graph:
        if distances[s] != float("Inf") and distances[s] + w <
            distances[d]:
            print("Graph contains negative weight cycle")
            return

    # No negative weight cycle found!
    # Return the distances array
    return distances


if __name__ == "__main__":
    graph = [
        (0, 1, 5),
        (0, 2, 4),
        (1, 3, 3),
        (2, 1, 6),
        (3, 2, 2)
    ]
    n = 4 # Number of vertices

    distances = bellman_ford(graph, 4, 0)
    for i in range(n):
        print(f"{i}\t\t{distances[i]}")
```

Output

```
0               0
1               5
2               4
3               8
```

Time complexity: The *time complexity* of the algorithm is $O(|V||E|)$, because the algorithm consists of $|V| - 1$ rounds and iterates through all $|E|$ edges during a round.

2.8.7 Minimum spanning tree (MST)

A *spanning tree* is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. The edges may or may not have weights assigned to them.
A *minimum spanning tree* (MST) is a spanning tree in which the sum of the weight of the edges is the minimum.

Prim's algorithm

Prim's algorithm is a greedy algorithm. It starts with an empty spanning tree.

1. At first the spanning tree consists only of a single vertex (chosen arbitrarily).

2. Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices.

3. Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex.

4. The process is repeated until the spanning tree contains all vertices.

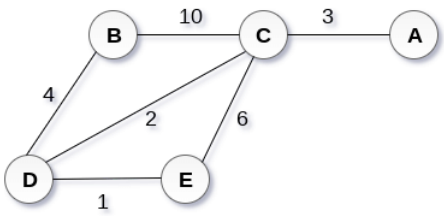


Figure 2.38: Prim's algorithm: starting graph

As an example, lets construct a minimum spanning tree of the graph given in the figure 2.38 by using Prim's algorithm.

- Choose a starting vertex B.

- Find the vertices that are adjacent to B. The edges that connecting the vertices are shown by dotted lines.

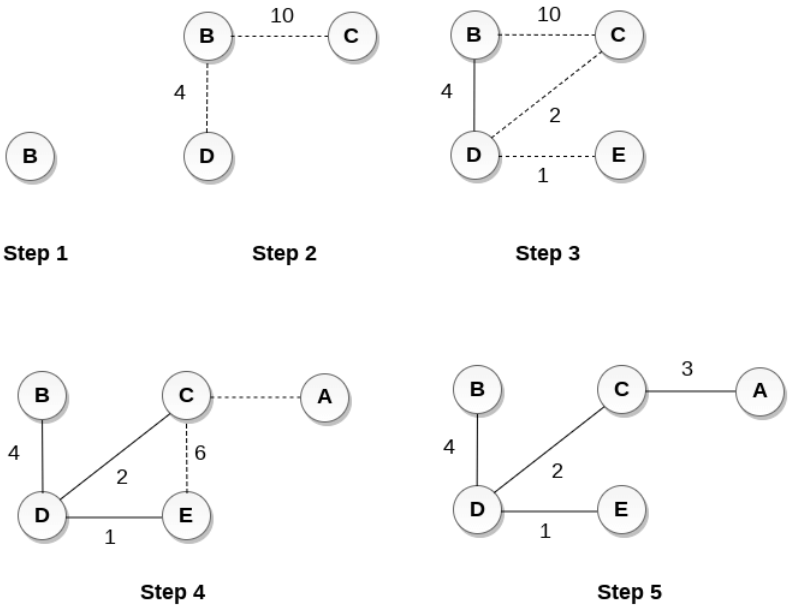


Figure 2.39: Prim's algorithm steps

- Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Now find all the edges that are adjacent to B or D and which are not yet selected i.e. BC, CD and DE.
- Choose the edge with the minimum weight among all. In this case, the edge CD is chosen. Add them to MST and then explore the edges adjacent to B, C and D i.e. CE and AC.
- Choose the edge with the minimum weight i.e. AC. (We can't choose CE as it would cause cycle in the graph.)

The graph produces in the final step is the minimum spanning tree of the graph shown in the above figure. The cost of MST is calculated as $4 + 2 + 1 + 3 = 10$ units.

```
def prim_algo(G):
    # Number of vertices in graph
    V = len(G)

    # Create a array to track selected vertex
    # Selected will become True otherwise False
    selected = [False]*V
    # Initially set number of edge to 0
    num_edge = 0
    # Choose 0th vertex and make it true
    selected[0] = True
```

```
    # Print for edge and weight
    print("Edge : Weight")
    while (num_edge < V-1):
        # For every vertex in 'selected', find the all adjacent
            vertices
        # Calculate the distance from the vertex selected at step 1.
        # If the vertex is already in 'selected', discard it
        # Choose another vertex nearest to selected vertex at step 1.
        minimum = float("inf") # To store the minimum edge
        x = 0 # To store the vertex from 'selected' when the edge
            toward 'not selected' is minimum
        y = 0 # To store the vertex not from 'selected' when the
            edge toward 'selected' is minimum
        for i in range(V):
            if selected[i]:
                for j in range(V):
                    if ((not selected[j]) and G[i][j] > 0):
                        # Not in selected and there is an edge
                        if minimum > G[i][j]:
                            minimum = G[i][j]
                            x = i
                            y = j
        print(str(x) + "-" + str(y) + " : " + str(G[x][y]))
        selected[y] = True
        num_edge += 1


if __name__ == "__main__":
    # Adjacency matrix representation of a graph
    G = [[0, 9, 75, 0, 0],
        [9, 0, 95, 19, 42],
        [75, 95, 0, 51, 66],
        [0, 19, 51, 0, 31],
        [0, 42, 66, 31, 0]]

    prim_algo(G)
```

Output

```
Edge : Weight
0-1 : 9
1-3 : 19
3-4 : 31
3-2 : 51
```

Time complexity: At every iteration of the loop, we add one node to the tree. Since there are $|V|$ nodes, it follows that there are $O(|V|)$ iterations of this loop. Within each iteration of the loop, we need to find and test edges in the tree. If there are $|E|$ edges, the naive searching implementation uses

$O(|E|)$ to find the edge with minimum weight. So in combination, we should expect the complexity to be $O(|V||E|)$, which may be $O(|V|^3)$ in the worst case.
Space complexity: $O(|V|)$

3

Algorithm Techniques

3.1 Brute force approach

Brute force approach is a direct, and straightforward technique of problem-solving in which all the possibilities are tried out till a satisfactory solution is not found. The advantage is that it guarantees that it will find the correct solution to a problem. But this approach is inefficient and slow.

3.1.1 Linear search

Given aa array `arr` of n elements, write a function to search a given element x in `arr`.

Linear search is a brute force searching algorithm where we start from one end and check every element of the list until the desired element is found.

Time complexity: $O(n)$
Space complexity: $O(1)$

```
def linearSearch(arr, x):
    n = len(arr)
    # Going through arr sequencially
    for i in range(n):
        if (arr[i] == x):
            return i
    return -1


if __name__ == "__main__":
    arr = [2, 4, 0, 1, 9]
    x = 1
    result = linearSearch(arr, x)
    if(result == -1):
        print("Element not found")
```

```
    else:
        print("Element found at index:", result)
```

Output

```
Element found at index: 3
```

The same linear search algorithm can also be implemented in recursive fashion as well.

Time complexity: $O(n)$ because of n number of recursive calls.
Space complexity: $O(1)$

```
def linearSearchRecursive(arr, idx, x):
    if idx == len(arr):
        return -1
    if (arr[idx] == x):
        return idx
    else:
        return linearSearchRecursive(arr, idx+1, x)


if __name__ == "__main__":
    arr = [2, 4, 0, 1, 9]
    x = 9
    result = linearSearchRecursive(arr, 0, x)
    if(result == -1):
        print("Element not found")
    else:
        print("Element found at index: ", result)
```

Output

```
Element found at index: 4
```

3.2 Divide and Conquer method

Divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. Here are the steps involved:

1. **Divide**: Divide the given problem into sub-problems using recursion.

2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.

3. **Combine**: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

3.2.1 Binary search

Given a sorted array `arr` of n elements, write a function to search a given element x in `arr`.

Binary search algorithm:

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we call the algorithm on the right half recursively.
4. Else (x is smaller) call the algorithm on the left half recursively.

Time complexity: $O(\log n)$
Space complexity: $O(1)$

```
def binarySearch(arr, left, right, element):
    # Check base case
    if right >= left:
        mid = left + (right - left) // 2
        # If element is present at the middle itself
        if arr[mid] == element:
            return mid
        # If element is smaller than mid, then it can only be
            present in left subarray
        elif arr[mid] > element:
            return binarySearch(arr, left, mid-1, element)
        # Else the element can only be present in right subarray
        else:
            return binarySearch(arr, mid + 1, right, element)
    else:
        # Element is not present in the array
        return -1


if __name__ == "__main__":
    arr = [2, 3, 4, 10, 40]
    element = 10
    result = binarySearch(arr, 0, len(arr)-1, element)

    if result != -1:
```

```
        print("Element is present at index % d" % result)
    else:
        print("Element is not present in array")
```

Output

```
Element is present at index 3
```

3.3 Greedy approach

A **greedy algorithm**, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.
Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision. Greedy algorithms have some advantages and disadvantages:

- It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.
- Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer).
- The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct.

3.3.1 Coin change problem

Given a value n and an infinite supply of coins with denominations 1, 5, 10 and 20 each, find the list coins to make the change so that the number of coins is minimum.

Let's start by having the values of the coins in an array in reverse sorted order i.e., `denominations = [20, 10, 5, 1]`.
Now if we have to make a value of n using these coins, then we will check for the first element in the array (greedy choice) and if it is greater than n, we will move to the next element, otherwise take it. Now after taking one coin with value `denominations[i]`, the total value which we have to make will now become `n-denominations[i]`.

```
def coin_change_greedy(n):
    denominations = [20, 10, 5, 1]
    i = 0
    min_coins = []

    while(n > 0):
        if denominations[i] > n:
            i += 1
        else:
            min_coins.append(denominations[i])
            n = n - denominations[i]
    return min_coins

if __name__ == '__main__':
    print(coin_change_greedy(23))
```

Output

```
[20, 1, 1, 1]
```

We can easily see that the algorithm is not going to take more than linear time. As n is decreased by `denominations[i]` at the end of the while loop, we can say that for most of the cases it will take much less than $O(n)$ time. So, we can say that our algorithm has $O(n)$ *time complexity*.

3.3.2 Job sequencing problem with deadlines

Given a list of tasks with deadlines and total profit earned on completing a task, find the maximum profit earned by executing a list of tasks within a maximum specified deadline. Assume that each task takes one unit of time to complete, and a task can't execute beyond its deadline. Also, only a single task will be executed at a time.

Example:

```
Input: Five Jobs with following deadlines and profits
JobID  Deadline Profit
  a       2       100
  b       1       19
  c       2       27
  d       1       25
  e       3       15
Maximum deadline: 10
Output: Following is maximum profit sequence of jobs
       c, a, e
```

A simple solution is to generate all subsets of given set of jobs and check individual subset for feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The *time complexity* of this solution is exponential. We can also solve this problem using greedy method which is more optimal than simple solution. So let's solve this problem via greedy method. To solve the job sequencing problem via greedy method follow this steps:

1. Sort all jobs in decreasing order of profit.
2. Iterate through the jobs and perform the following:
   (a) Choose a slot i if $i <$ deadline for the job and i is maximum.
   (b) If no such slot exists, ignore the job and continue.
3. Do following for remaining $n - 1$ jobs.
4. If the current job can fit in the current result sequence without missing the deadline, add current job to the result. Else ignore the current job.

Jobs	J1	J2	J3	J4	J5	J6	J7
Profits	35	30	25	20	15	12	5
Deadlines	3	4	4	2	3	1	2

Figure 3.1: An example of the job sequencing problem with deadlines

Time complexity $O(n^2)$.

```
# A class to store job details. Each job has an identifier,
# a deadline, and profit associated with it.
class Job:
    def __init__(self, taskId, deadline, profit):
        self.taskId = taskId
        self.deadline = deadline
        self.profit = profit


# Function to schedule jobs to maximize profit
def scheduleJobs(jobs, T):

    # stores the maximum profit that can be earned by scheduling
        jobs
    profit = 0

    # list to store used and unused slots info
    slot = [-1] * T
```

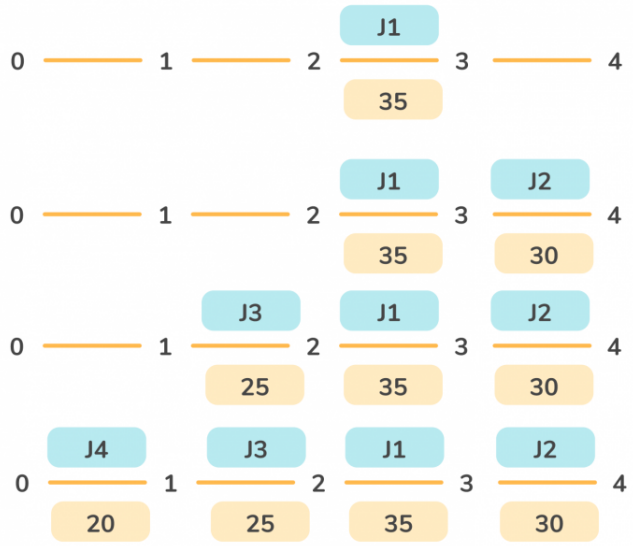


Figure 3.2: The solution of the job sequencing problem with deadlines

```
    # arrange the jobs in decreasing order of their profits
    jobs.sort(key=lambda x: x.profit, reverse=True)

    # consider each job in decreasing order of their profits
    for job in jobs:
        # search for the next free slot and map the task to that slot
        for j in reversed(range(job.deadline)):
        # for j in range(job.deadline):
            if j < T and slot[j] == -1:
                slot[j] = job.taskId
                profit += job.profit
                break

    # print the scheduled jobs
    print('The scheduled jobs are', list(filter(lambda x: x != -1,
        slot)))

    # print total profit that can be earned
    print('The total profit earned is', profit)


if __name__ == '__main__':

    # List of given jobs. Each job has an identifier, a deadline,
        and
    # profit associated with it
    jobs = [
```

```
        Job('T1', 9, 15),
        Job('T2', 2, 2),
        Job('T3', 5, 18),
        Job('T4', 7, 1),
        Job('T5', 4, 25),
        Job('T6', 2, 20),
        Job('T7', 5, 8),
        Job('T8', 7, 10),
        Job('T9', 4, 12),
        Job('T10', 3, 5)
    ]

    # Maximum deadline
    T = 15

    # schedule jobs and calculate the maximum profit
    scheduleJobs(jobs, T)
```

Output

```
The scheduled jobs are ['T7', 'T6', 'T9', 'T5', 'T3', 'T4', 'T8',
    'T1']
The total profit earned is 109
```

3.3.3 Fractional knapsack problem

Given weights w_i and values (or profits) V_i of n items ($i = 1, 2, \cdots n$), put these items in a knapsack of capacity W to get the maximum total profit in the knapsack.

The problem is called fractional knapsack because we can either include any fraction of the items into the knapsack. Let's take an example. Take the following input values.

$$\begin{aligned} V &= [50, 100, 150, 200] \\ w &= [8, 15, 32, 40] \\ W &= 60 \end{aligned}$$

Here we get the maximum profit when we include 8 units of item 1, 15 units of item 2 and 37 units of item 4 giving us a total profit of

$$50 + 100 + \frac{200}{40} \times 37 = 335 \text{ units}$$

We can apply greedy method to solve this fractional knapsack problem. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio.

$$\frac{V_1}{w_1} \geq \frac{V_2}{w_2} \geq \cdots \frac{V_n}{w_n}$$

Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

Time complexity: First we need to sort the items in decreasing order of value/weight ratio, which takes $O(n \log n)$ time. Then iteration over each element takes $O(n)$ time. So the *time complexity* is $O(n \log n)$.
Space complexity: $O(n)$ to store the value/weight ratios in an array.

```
def fractional_knapsack(value, weight, capacity):
    # index = [0, 1, 2, ..., (n-1)] for n items
    index = list(range(len(value)))
    # Contains ratios of values to weight
    ratio = [v/w for v, w in zip(value, weight)]
    # index is sorted according to value-to-weight ratio in
        decreasing order
    index.sort(key=lambda i: ratio[i], reverse=True)

    max_value = 0
    fractions = [0]*len(value)
    for i in index:
        if weight[i] <= capacity:
            fractions[i] = 1 # Full fraction is included
            max_value += value[i]
            capacity -= weight[i]
        else:
            fractions[i] = capacity/weight[i]
            max_value += value[i]*capacity/weight[i]
            break

    return max_value, fractions


if __name__ == "__main__":
    value = [50, 100, 150, 200]
    weight = [8, 15, 32, 40]
    capacity = 60

    max_value, fractions = fractional_knapsack(value, weight,
        capacity)
    print('The maximum value of items that can be carried:',
        max_value)
    print('The fractions in which the items should be taken:',
        fractions)
```

Output

```
The maximum value of items that can be carried: 335.0
The fractions in which the items should be taken: [1, 1, 0, 0.925]
```

3.4 Backtracking method

Backtracking is an improvement of the *brute-force approach*. It tries to search for a solution to a problem among all the available options. It finds a solution set by developing a solution step by step, increasing levels with time, using recursive calling. Since a problem would have constraints, the solutions that fail to satisfy them will be removed. In order to find these solutions, a search tree named **state-space tree** is used. In a state-space tree, each branch is a variable, and each level represents a solution.

Before start solving the problem we must be able to recognize if it can be solved using a backtracking algorithm. There are the following types of problems that can be solved using backtracking:

1. *Decision Problem*: In this type of problem we always search for a feasible solution.
2. *Optimization Problem*: In this type of problem we always search for the best possible solution.
3. *Enumeration Problem*: In this type of problem we try to find all feasible solutions.

3.4.1 Suitcase password problem

Consider a suitcase lock, where a three-digit password is forgotten. Assume the digits are restricted to binary number and the unknown password is said to be having at most one 0. Construct a backtracking solution to generate all possible passwords.

The potential password is of three digits which can be visualized as three digits (x_1, x_2, x_3) where $x_i \in \{0, 1\}, i = 1, 2, 3$. To construct a state-space tree, start with a Root. There are two choices 1 or 0, at every stage. This leads to the generation of the binary tree as shown in the figure 3.3. It can be observed all potential passwords are marked with green while all non-relevant passwords are marked with red. Check all the green leaves of the tree to find the valid solution.

```
password_list = []
```

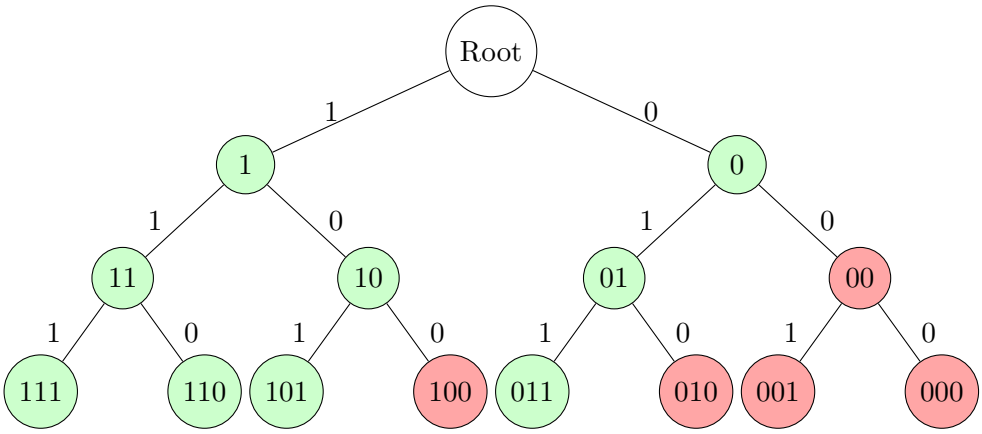


Figure 3.3: Suitcase password problem

```
# Recursive helper function, adds character until length is reached
def add_character(arr, i, s, N):
    # Base case
    if (i == 0): # when len has been reached
        # print out the elements of the state-space tree
        print(s)
        if len(s) == N:
            password_list.append(s)
        return

    # Iterate through the array
    for j in arr:
        # 1. Create new string after appending next character
        # 2. In case there is already one zero we will not append
            any zero
        # 3. Call add_character() again until string has reached its
            length
        if j == '0' and s.count('0') == 1:
            # Solution is not feasible, so skip
            continue

        appended = s + j
        add_character(arr, i-1, appended, N)

    return

# Function to generate all possible passwords
def generate(arr, N):
    # arr: Array of characters used in the password
    # N: Length of the password
    for i in range(1, N+1):
        add_character(arr, i, "", N)
```

```
if __name__ == "__main__":
    arr = ['1', '0'] # Array of characters used in the password
    N = 3 # Length of the password
    print('The elements of the state-space tree:')
    generate(arr, N)
    print('password_list:', password_list)
```

Output

```
The elements of the state-space tree:
1
0
11
10
01
111
110
101
011
password_list: ['111', '110', '101', '011']
```

3.4.2 Permutation of a string

Given a string S, print all the possible permutations of S.

A permutation is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has $n!$ permutation. We will apply backtracking approach to find all the possible solutions by exploring all possible ways. If one of the found solutions turns out to not satisfy the given criteria, it discards the solution and makes some changes and backtracks again.

```
Input: S = "abc"
Output: ["abc", "acb", "bac", "bca", "cba", "cab"]
```

1. Define a pointer `idx` which will initially point to the first index of the given string S.

2. Now with another pointer i loop through the characters of the string from `idx` $\rightarrow$ last charater of the string at $(n-1)$, where n is the length of the string and do the following:

   (a) Swap $S[i]$ and $S[idx]$.

(b) Recall the backtracking function again after moving `idx` pointer to the right by one character i.e. `idx = idx+1`.

(c) When `idx` moves to the final character, print the string. This will serve as the base case.

For example let's take the string `"ABC"`. We set the pointer `idx` at 0, pointing at the first character `"A"`. Swapping `"A"` with all the three characters yield `"ABC"`, `"BAC"`, `"CBA"`. Now we apply the permutation function again on each of the three strings with `idx` pointer set at 1 thereby making the first character fixed. From `"ABC"`, we thus get `"ABC"` and `"ACB"` where `"A"` is fixed. Similarly from `"BAC"`, we get `"BAC"` and `"BCA"` and from `"CAB"`, we thus get `"CAB"` and `"CBA"`. After that the `idx` pointer set at 2 which is the stopping condition. We then take print out of the six permutations we got so far.

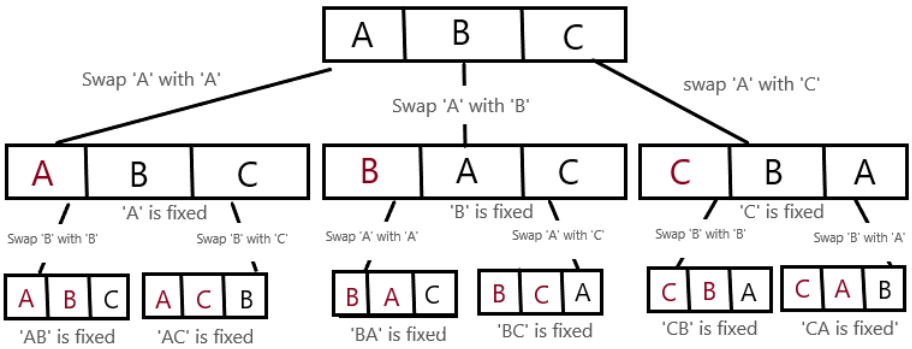


Figure 3.4: Permutation of string "ABC"

Time Complexity: $O(n * n!)$, where n is the length of the string since there are $n!$ permutations and it requires $O(n)$ time to print a permutation.
Space Complexity: $O(n!)$, since one has to keep $n!$ solutions.

```
def permutations(char_list, idx=0):
    # Final case
    if idx == len(char_list)-1:
        print(''.join(char_list))

    for i in range(idx, len(char_list)):
        # Swap the current index with i
        char_list[idx], char_list[i] = char_list[i], char_list[idx]
        permutations(char_list, idx + 1)

if __name__ == '__main__':
    a_string = 'ABC'
    # 'str' object does not support item assignment or swapping, so
        we convert it to character list
```

```
    char_list = [x for x in a_string]
    permutations(char_list)
```

Output

```
ABC
ACB
CAB
CBA
ABC
ACB
```

4

Sorting Algorithms

4.1 Bubble sort

1. Compare the first two elements of the list to check them which one is greater.
2. For ascending ordering, place the larger value on the right.
3. Do the same for second, third elements and so on till the whole list is completed. This is the first round.
4. Execute steps (1), (2) and (3) for n number of times where n is the length of the list. (In second round you don't need to consider the last element. In third round no need to consider the last two elements and so on.)

Time complexity: $O(n^2)$ for number of rounds (n) $\times$ number of comparisons (n) in the worst case
Space complexity: $O(1)$

Table 4.1: **Bubble sort: Total round = length(list) - 1 = 3**

First round	Second round	Third round
$\underline{7}\ \underline{2}\ 5\ 4$	$\underline{2}\ \underline{5}\ 4\ 7$	$\underline{2}\ \underline{4}\ 5\ 7$
$2\ \underline{7}\ \underline{5}\ 4$	$2\ \underline{5}\ \underline{4}\ 7$	$2\ 4\ 5\ 7$
$2\ 5\ \underline{7}\ \underline{4}$	$2\ 4\ 5\ 7$	
$2\ 5\ 4\ 7$		
Iteration num. = 3	Iteration num. = 2	Iteration num. = 1

```
def bubble_sort(input_list):
   L = len(input_list)
   Round_num = L-1
```

```
   for rnd in range(Round_num):
      Iterations_per_round = L - rnd
      for iteration in range(Iterations_per_round):
         for i in range(0, L-1-iteration):
            if input_list[i] > input_list[i+1]:
               swap_value_by_index(input_list, i, i+1)

def swap_value_by_index(a_list, a, b):
    a_list[b], a_list[a] = a_list[a], a_list[b]
```

4.2 Selection sort

1. The first element is compared with all other elements in the list one-by-one.
2. If any element is found to be greater than any other element, then they are swapped.
3. After the first iteration, the smallest number is placed at the first position. The same procedure is repeated for the second element and so on.

Time complexity: $O(n^2)$
Space complexity: $O(1)$

Table 4.2: **Selection sort: Total round = length(list) - 1 = 3**

First round	Second round	Third round
$\underline{7}\ \underline{2}\ 5\ 4$	$2\ \underline{7}\ \underline{5}\ 4$	$2\ 4\ \underline{7}\ \underline{5}$
$\underline{2}\ 7\ \underline{5}\ 4$	$2\ \underline{5}\ 7\ \underline{4}$	$2\ 4\ 5\ 7$
$\underline{2}\ 7\ 5\ \underline{4}$	$2\ 4\ 7\ 5$	
$2\ 7\ 5\ 4$		
Iteration num. = 3	Iteration num. = 2	Iteration num. = 1

```
def selection_sort(input_list):

    L = len(input_list)

    # Iteration for 1st element
    for i in range(L-1):
        # Iteration for 2nd element
        for j in range(i+1, L):
            min_idx = i # Indicating the min value index
            if input_list[min_idx] > input_list[j]:
                min_idx = j
                swap_value_by_index(input_list, min_idx, i)

def swap_value_by_index(a_list, a, b):
    a_list[b], a_list[a] = a_list[a], a_list[b]
```

4.3 Insertion sort

1. The first iteration starts with comparing the first and second elements and ordering them accordingly.
2. In second iteration, the third element is compared with the first and second elements and placed in a suitable position by shifting other elements.
3. The procedure is repeated for all elements in the array.

Time complexity: $O(n^2)$
Space complexity: $O(1)$

Figure 4.1: Insertion sort

```
def insertion_sort(input_list):
    L = len(input_list)

    for i in range(1, L):
        current_value = input_list[i] # The value currently being
            compared
        pos = i # Saving the index i in a temporary variable

        # Move elements of array [0, 1, 2, ..., (i-1)] that are
        # greater than current_value to one position ahead of
        # their current position
        while pos > 0 and input_list[pos - 1] > current_value:
            input_list[pos] = input_list[pos - 1]
            pos -= 1
        # Place the current_value to the gap that is created
        # after moving the elements that are greater than
        # current_value to the right.
        input_list[pos] = current_value

if __name__ == "__main__":
    input_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    print("Before sorting:")
    print(input_list)
    insertion_sort(input_list)
    print("After insertion sort:")
    print(input_list)
```

Output

```
Before sorting:
[54, 26, 93, 17, 77, 31, 44, 55, 20]
After insertion sort:
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

4.4 Quick sort

Quick sort is a divide-and-conquer algorithm which works by obtaining a *pivot* item from the array and partitioning the array so that all elements smaller than pivot will be on the left side of the array and all elements greater than pivot will be on the right side of the array. The pivot element will be at its final sorted position. Perform the same operation for the sub-arrays on the left and right of pivot.

1. Find the pivot item in the array. For simplicity take the first value.
2. Start a *low* pointer at the second item of the array and a *high* pointer at the last item of the array.
3. Move the low pointer to the right as long as the element $\leq$ pivot.
4. Move the high pointer to the left as long as the element $\geq$ pivot.
5. At this point we find a low pointer value which is higher than the high pointer value. Swap the values only if the low pointer $\leq$ high pointer.
6. Continue moving the low and high pointer in the same fashion until low pointer crosses the high pointer.
7. When the low pointer crosses the high pointer, swap the pivot and high pointer value. All the elements to the left of new pivot is smaller than the pivot and all the elements to the right of new pivot is larger than the pivot i.e. the new pivot is in the correct position.
8. Now we recursively apply the same logic to the left and right sub-array.

```
# This function takes first element as pivot, places the pivot
    element at its correct position in sorted arr, and places all
    smaller (smaller than pivot) to left of pivot and all greater
    elements to right of pivot
def partition(arr, start, end):
    pivot = arr[start]
    low = start + 1
    high = end
    while True:
        # If the current value we're looking at is less than the
            pivot, then it's in the right place (right side of
            pivot) and we can move right, to the next element.
        # We also need to make sure we haven't surpassed the low
            pointer, since that indicates we have already moved all
            the elements to their correct side of the pivot
        while low <= high and arr[low] <= pivot:
            low = low + 1
```

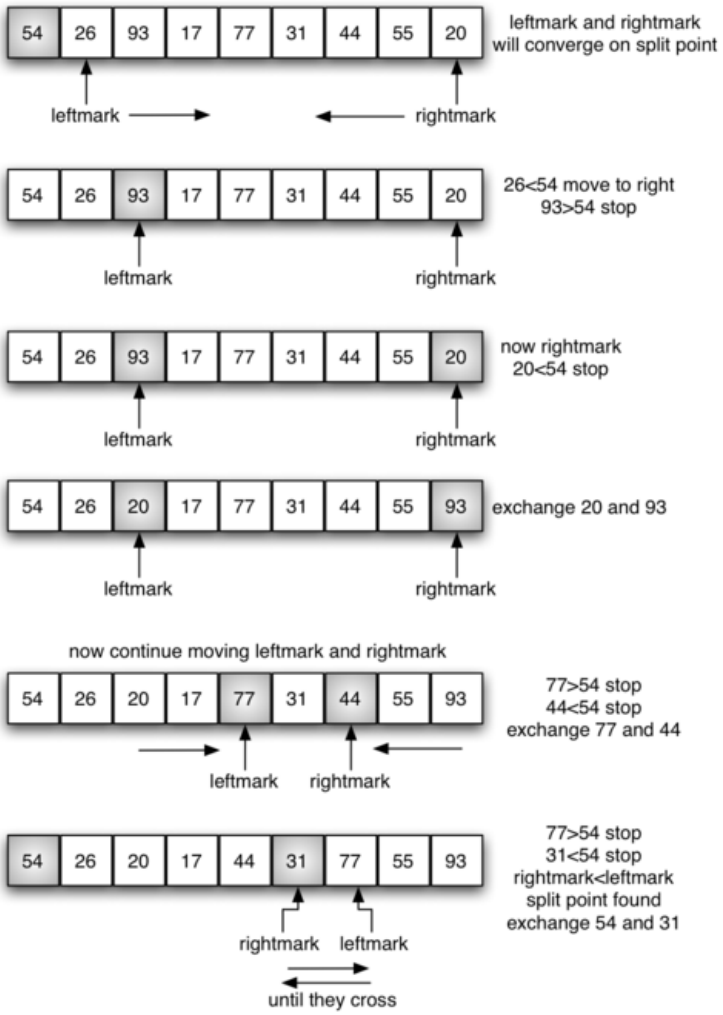


Figure 4.2: Quick sort: Partition process

```
    while low <= high and arr[high] >= pivot:
        high = high - 1

    # We either found a value for both high and low that is out
        of order or low is higher than high, in which case we
        exit the loop
    if low <= high:
        arr[low], arr[high] = arr[high], arr[low]
        # The loop continues
    else:
        # We exit out of the loop
        break
# Swap the pivot and high indexed element
```

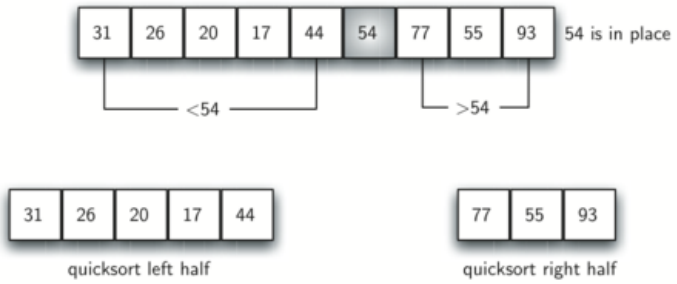


Figure 4.3: Quick sort: Recursion after partition

```
    arr[start], arr[high] = arr[high], arr[start]
    return high


def quick_sort(arr, start, end):
    if start >= end:
        return
    p = partition(arr, start, end)
    quick_sort(arr, start, p-1)
    quick_sort(arr, p+1, end)


if __name__ == "__main__":
    arr = [29, 99, 27, 41, 66, 28, 44]
    print('Before sorting:', arr)
    quick_sort(arr, 0, len(arr) - 1)
    print('After sorting:', arr)
```

Output

```
Before sorting: [29, 99, 27, 41, 66, 28, 44]
After sorting: [27, 28, 29, 41, 44, 66, 99]
```

To analyze the `quick_sort` function, note that for a list of length n, if the partition always occurs in the middle of the list, there will again be $\log n$ divisions. In order to find the split point, each of the n items needs to be checked against the pivot value. The result is $n \log n$. This results in an average case complexity $\Theta(n \log n)$.
Unfortunately, in the worst case, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division. In this case, sorting a list of n items divides into sorting a list of 0 items and a list of $n - 1$ items. Then sorting a list of $n - 1$ divides into a list of size 0 and a list of size $n - 2$, and so on. The result is the worst case complexity $O(n^2)$ sort with all of the overhead that recursion requires.

4.5 Merge sort

Merge sort is based on divide-and-conquer approach. It divides the input array into two halves. Each half is solved individually, and then the two sorted halves are merged to get final result.

MergeSort(array)

If length of array > 1, do the following:

1. Find the middle point m

$$m = (\text{length of array})/2$$

2. Divides the input array into two halves

$$\text{leftArray} = \text{array[1:m]}$$

$$\text{rightArray} = \text{array[m:]}$$

3. Call MergeSort() on both subarray

$$\text{MergerSort(leftArray)}$$

$$\text{MergerSort(rightArray)}$$

4. Merge the two halves sorted in (2) and (3)

$$\text{Merge(leftArray, rightArray)}$$

```
import math

# This function merge the two sorted subarrays into one single
    sorted array
def merge(left_array, right_array):
    result = []
    # Left and right index initialized to zero
    L, R = 0, 0
    # Move the two pointer through the two arrays. Append the
        element to the resultant array whichever is less.
    while (L < len(left_array)) and (R < len(right_array)):
        if left_array[L] < right_array[R]:
            result.append(left_array[L])
            L += 1
        else:
            result.append(right_array[R])
            R += 1
```

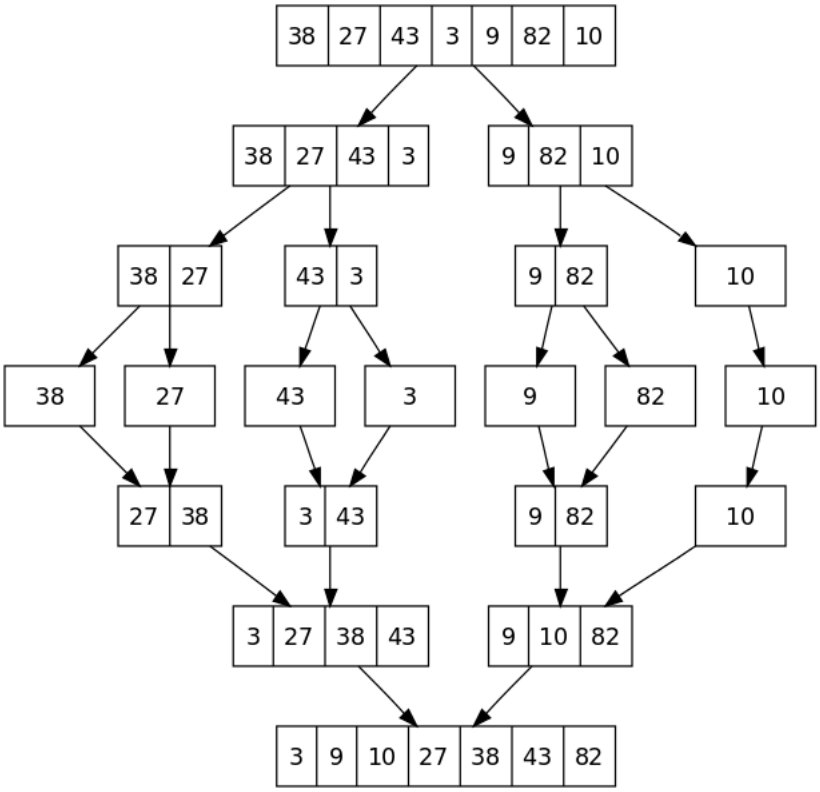


Figure 4.4: Merge sort

```
    # Remaining part of the subarrays
    left_remains = left_array[L:]
    right_remains = right_array[R:]

    # Concatenate the remaining to the resultant array
    return result + left_remains + right_remains


def merge_sort(arr):
    if len(arr) < 2:
        return arr
    m = math.ceil(len(arr)/2)
    left_array = arr[:m]
    right_array = arr[m:]

    return merge(merge_sort(left_array), merge_sort(right_array))


if __name__ == "__main__":
    arr = [29, 99, 27, 41, 66, 28, 44]
    print('Before sorting:', arr)
```

```
    arr = merge_sort(arr)
    print('After sorting:', arr)
```

Output

```
Before sorting: [29, 99, 27, 41, 66, 28, 44]
After sorting: [27, 28, 29, 41, 44, 66, 99]
```

5

Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given problem by breaking it into subproblems and save the result of subproblems for the future so that we will not have to compute that same problem again. Following are the two main properties of a problem that suggests that the given problem can be solved using dynamic programming.

1. **Overlapping subproblems**: Like divide-and-conquer, dynamic programming divide problems into subproblems. In divide-and-conquer strategy, the subproblems are independent of each other; however in dynamic programming, the same subproblems are needed again and again. In dynamic programming, the computed solutions to subproblems are stored in a table so that these don't have to be recomputed. This is called overlapping subproblems.

2. **Optimal substructure**: A given problems has optimal substructure property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

There are two ways dynamic programming can be applied:

1. **Top-down (Memoization) approach**: Top-down breaks the large problem into multiple subproblems. If the subproblem has already been solved, then just reuse the answer. Otherwise, solve the subproblem and store the result. Top-down uses recursion with memoization to avoid recomputing the same subproblem again.

2. **Down-top (Tabulation) approach**: Start computing result for the lowest possible subproblem (base) and keep it in a table. Use the subproblem result to solve other subproblems and subsequently keep the results in the table and thus finally solve the whole main problem.

Greedy	Divide & Conquer	Dynamic Programming
Optimises by making the best choice at the moment.	Optimises by breaking down a subproblem into simpler versions of itself and using multithreading and recursion to solve.	Same as Divide and Conquer, but optimises further by caching the answers to each subproblem as not to repeat the calculation twice.
Doesn't always find the optimal solution, but is very fast.	Always finds the optimal solution, but is slower than Greedy.	Always finds the optimal solution, but could be pointless on small datasets.
Requires almost no memory.	Requires some memory to remember recursive calls.	Requires a lot of memory for memoisation/tabulation.

Table 5.1: Difference between Greedy, Divide and Conquer and Dynamic Programming

5.1 Fibonacci sequence

Let's start with a basic example of the *Fibonacci sequence*. Fibonacci sequence is a sequence of positive integers in which the first two elements are 0 and 1 and each of the following elements is the sum of the two preceding ones.

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

First solution: Using ordinary **recursion method**.

```
def fib(n):
    if n <= 1:
       return n
   else:
       return fib(n-1) + fib(n-2)
```

Here, the program will call itself, again and again, to calculate further values as shown in the figure. The calculation of the *time complexity* of the recursion based approach is around $O(2^n)$. The *space complexity* of this approach is $O(n)$ as recursion can go max to n. In this method values like F_2 and F_3 are computed multiple times. Imagine the number of repetitions if you have to calculate for F_{100}. This method is ineffective for large values.

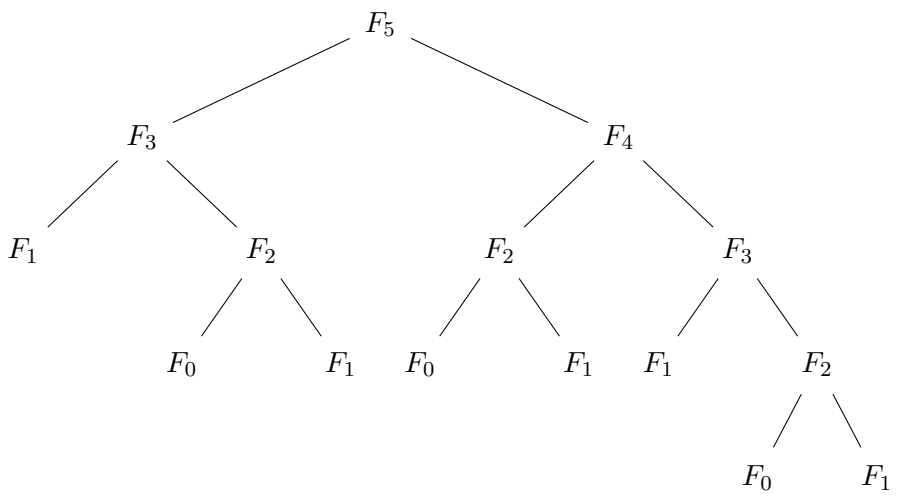


Figure 5.1: Recursion tree for Fibonacci sequence

Second solution: Using **Top-down method**

```
def fib(n, cache = {}):
    if cache.get(n) != None:
        return cache[n]
    if n <= 1:
        return n
    else:
        res = fib(n-1, cache) + fib(n-2, cache)
        cache[n] = res
        return res
```

Here, the computation time is reduced significantly as the outputs produced after each recursion are stored in a cache which can be reused later. This method is much more efficient than the previous one. Let's now calculate its *time complexity*.

$$T(n) = T(n-1) + T(n-2) + c$$

Now when we calculate `fib(n-1)`, we already have `fib(n-2)` in cache. So the number of operations for `fib(n-2)` is 1 (i.e. reading from the cache). Therefore

$$\begin{aligned} T(n) &= T(n-1) + 1 + c \\ &= [T(n-2) + 1 + c] + 1 + c \\ &= T(n-2) + 2(1+c) \\ &= \cdots \\ &= T(n-k) + k(1+c) \end{aligned}$$

Now let's find the value of k for which $n - k = 0$ i.e. $k = n$. Therefore,

$$\begin{aligned} T(n) &= T(0) + n(1+c) \\ &= 1 + n(1+c) \in O(n) \end{aligned}$$

The *space complexity* is $O(n)$ to store the cache.

Third solution: Using **Bottom-up method**

```
def fib(n):
    if n <= 1:
        return n
    dp = [0]*(n+1)
    dp[0] = 0
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

This code doesn't use recursion at all. Here, we create an empty list of length $(n+1)$ and set the base case of `fib(0)` and `fib(1)` at index positions 0 and 1. Then using a for loop for index values 2 up to n, the rest of the values are populated in the list. Unlike in the recursive method, the *time complexity* of this code is linear i.e., it runs in $O(n)$. The *space complexity* is $O(n)$ because of the requirement of storing the list.

5.2 Integers from a given list to add up to a number

Given a number n, find the number of ways of expressing n as a sum from a list of integers 1, 3, and 4.

If $n = 5$, then there are 6 ways to express n namely,

$$\{4,1\}, \{1,4\}, \{1,3,1\}, \{3,1,1\}, \{1,1,3\}, \{1,1,1,1,1\}$$

Divide the problem into sub-problems for solving it. Let $dp[n]$ be the be the number of ways to write n as the sum of 1, 3, and 4. Consider one

possible solution with $n = x_1 + x_2 + x_3 + \cdots + x_k$. If the last number is 1, then sum of the remaining numbers is $n - 1$. So the number that ends with 1 will contribute to $dp[n-1]$ number of ways. Similarly the number that ends with 3 and 4 will contribute to $dp[n-3]$ and $dp[n-4]$ number of ways respectively. The final recurrence would be:

$$dp[n] = dp[n-1] + dp[n-3] + dp[n-4]$$

with base cases:

$$dp[0] = dp[1] = dp[2] = 1 \text{ and } dp[3] = 2$$

First solution: Using Top-down approach
Time complexity: $O(n)$
Space complexity: $O(n)$

```
def number_sum_subset(n, memo = {}):
    # If the number is 0, 1, or 2, return 1.
    # The only ways we can represent: 0 = 0, 1 = 1, 2 = 1 + 1
    if n in (0, 1, 2):
        return 1
    # if the number is 3, return 2 because 3 = 1 + 1 + 1, 3 = 3
    elif n == 3:
        return 2
    else:
        if memo.get(n) != None:
            return memo[n]
        #  Recursively call the function for three sub-parts n-1,
            n-3, n-4.
        sub_part1 = number_sum_subset(n-1, memo)
        sub_part2 = number_sum_subset(n-3, memo)
        sub_part3 = number_sum_subset(n-4, memo)
        memo[n] = sub_part1 + sub_part2 + sub_part3
        return memo[n]
```

Second solution: Using Bottom-up approach
Time Complexity: $O(n)$
Space Complexity: $O(n)$

```
def number_sum_subset(n):
    dp = []
    for i in range(n+1):
        # If the number is 0, 1, or 2, append 1.
        # The only ways we can represent: 0 = 0, 1 = 1, 2 = 1 + 1
        if i in (0, 1, 2):
            dp.append(1)
        # if the number is 3, append 2 because 3 = 1 + 1 + 1, 3 = 3
        elif i == 3:
            dp.append(2)
```

```
        else:
            dp.append(dp[i-1] + dp[i-3] + dp[i-4])
    return dp[n]
```

5.3 House robber problem

We are given n number of houses along a street, each having some amount of money. A thief can not rob from adjacent houses. Find the maximum amount that can be robbed.

Example 1

```
Input: [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money =
    3). Total amount you can rob = 1 + 3 = 4.
```

Example 2

```
Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and
    rob house 5 (money = 1). Total amount you can rob = 2 + 9 + 1 =
    12.
```

First solution: Using Top-down approach
So, let's say we're indexing the houses from 0, and we're in 0th house at the start. Here, we've got two choices. Either we can rob house 0 and go to house 2 (leaving adjacent house 1) OR we can leave house 0 and rob house 1. We'll take the maximum of these two choices. We solve the problem recursively with the above two choices at each index i, and whenever we reach the end, we simply return 0 as there are no additional houses left. So, for the ith index, we have

- **Option1 (rob the house)**: Rob the current house at index i and move to index $i + 2$. The profit will be

  `money[i] + max_loot(money, i+2)`

- **Option2 (skip the house)**: Skip the house at index i and move to the next house at $i + 1$ and see the profit. The profit will be

  `max_loot(money, i+1)`

Decision:

```
max(money[i] + max_loot(money, i+2), max_loot(money, i+1))
```

Time Complexity: $O(n)$
Space Complexity: $O(n)$

```
# Maximum loot at the current index
def max_loot(money, currentIndex, memo = {}):
    if currentIndex >= len(money):
        return 0
    else:
        # Get the result from memory if available
        if memo.get(currentIndex) != None:
            return memo[currentIndex]
        # Option 1
        rob_house = money[currentIndex] + max_loot(money,
            currentIndex+2, memo)
        # Option 2
        skip_house = max_loot(money, currentIndex+1, memo)
        # Store the result in memory
        memo[currentIndex] = max(rob_house, skip_house)
        return memo[currentIndex]

if __name__ == "__main__":
    money = [6,7,1,30,8,2,4]
    print('Max loot:', max_loot(money, 0))
```

Output

```
Max loot: 41
```

Second solution
Here we have to look at the other direction. If we are at current index i, then what is the maximum money we can steal up to now. We have two options:

- **Option1 (rob the house)**: Rob the current house at index i and move back to index $i-2$. The profit will be

```
money[i] + max_loot(money, i-2)
```

- **Option2 (skip the house)**: Skip the next house at index i and move back to the previous house at i and see the profit. The profit will be

```
max_loot(money, i-1)
```

Decision:

```
max(money[i] + max_loot(money, i-2), max_loot(money, i-1))
```

```
def max_loot(money, currentIndex, memo = {}):
    if currentIndex <= 0:
        return 0
    else:
        if memo.get(currentIndex) != None:
            return memo[currentIndex]
        rob_house = money[currentIndex] + max_loot(money,
            currentIndex-2, memo)
        skip_house = max_loot(money, currentIndex-1, memo)

        memo[currentIndex] = max(rob_house, skip_house)
        return memo[currentIndex]

if __name__ == "__main__":
    money = [6,7,1,30,8,2,4]
    print(max_loot(money, len(money)-1))
```

Output

```
Max loot: 41
```

Third solution: Using Bottom-up approach

Time Complexity: $O(n)$
Space Complexity: $O(n)$

```
def max_loot(money):
    dp = []
    for i in range(len(money)):
        # Base case: when i = 0 (only one house) rob the house
        if i == 0:
            dp.append(1)
        # Base case: when i = 1 (two houses with indices 0 and 1)
            rob the house with max money
        elif i == 1:
            dp.append(max(money[0], money[1]))
        else:
            dp.append(max(money[i] + dp[i-2], dp[i-1]))
    return dp[-1]

if __name__ == "__main__":
    money = [6, 7, 1, 30, 8, 2, 4]
    print(max_loot(money))
```

Output

```
Max loot: 41
```

5.4 0/1 Knapsack problem

Given weights w_i and values (or profits) V_i of n items ($i = 1, 2, \cdots n$), put these items in a knapsack of capacity W to get the maximum total profit in the knapsack.

The problem is called 0/1 knapsack because we can either include an item as a whole or exclude it. That is to say, we can't take a fraction of an item. Let's take an example. Take the following input values.

$$\begin{aligned} V &= [50, 100, 150, 200] \\ w &= [8, 16, 32, 40] \\ W &= 64 \end{aligned}$$

Here we get the maximum profit when we include items 1, 2 and 4 giving us a total of 200 + 50 + 100 = 350.
To solve 0/1 knapsack using dynamic programming we construct a table with dimensions $(n + 1) \times (W + 1)$. The rows of the table correspond to items from 0 to n. The columns of the table correspond to weight limit from 0 to W. The cell with index ij contains a value $A[i][j]$ which represents the maximum profit possible when considering items from 0 to i and the total weight limit as j.
The table is filled up as follows: Let's start by setting the 0th row and column to 0. We do this because the 0th row means that we have no objects and the 0th column means that the maximum weight possible is 0. This is the base condition. Now for each cell ij, we have two options:

- Either we include ith item in our final selection.
- Or, we don't include ith item in our final selection.

Now how do we decide whether we include object i in our selection? There are two conditions that should be satisfied to include object i:

1. The total weight after including ith item should not exceed the weight limit. In this case the profit is $A[i - 1, j]$.
2. The profit after including ith item should be greater as compared to when the item is not included. In this case the profit is $V_{i-1} + A[i - 1, j - w_{i-1}]$ (since the indexing is started from 0, the weight of the ith item is w_{i-1} and value for the ith item is V_{i-1}).

With all the above conditions, the recursive formulation of knapsack problem is given as follows:

$$A[i][j] = \begin{cases} 0, & \text{if } i = 0 \\ 0, & \text{if } j = 0 \\ A[i-1, j], & \text{if } w_{i-1} > j \\ \max(A[i-1, j], V_{i-1} + A[i-1, j-w_{i-1}]), & \text{if } w_{i-1} \leq j \end{cases}$$

The algorithm basically works like this: If item (n, W) is kept in the knapsack, then recursively the $(n-1)$th item $(n-1, W-w_{n-1})$ is checked. If the item (n, W) is not kept, then the $(n-1)$th item $(n-1, W)$ is checked. This is repeated to keep track of the items in the knapsack.

Time complexity: $O(nW)$
Time complexity: $O(nW)$

Numerical example: Apply the algorithm to the following example with knapsack capacity $W = 3$.

Item	Weight	Value
A	w_0 =1	V_0 =1
B	w_1 =2	V_1 =6
C	w_2 =4	V_2 =4

Table 5.2: An example of 0/1 Knapsack problem

Initial table:

$$A[0][0] = A[1][0] = A[2][0] = A[3][0] = 0$$
$$A[0][0] = A[0][1] = A[0][2] = A[0][3] = 0$$

i\j	0	1	2	3
0	0	0	0	0
1	0			
2	0			
3	0			

Table 5.3: Initial table

Recursion for first row:

$$\begin{aligned} A[1,1] &= \max(A[1-1,1], V_0 + A[1-1,1-w_0]) \\ &= \max(A[0,1], 1 + A[0,0]) = 1 \\ A[1,2] &= \max(A[1-1,2], V_0 + A[1-1,2-w_0]) \\ &= \max(A[0,2], 1 + A[0,2]) = 1 \\ A[1,3] &= \max(A[1-1,3], V_0 + A[1-1,3-w_0]) \\ &= \max(A[0,3], 1 + A[0,3]) = 1 \end{aligned}$$

i\j	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0			
3	0			

Table 5.4: After adding the first row

Recursion for second row:

$$\begin{aligned} A[2,1] &= A[1,1] = 1 \\ A[2,2] &= \max(A[1,2], V_1 + A[1,2-w_1]) = 6 \\ A[2,3] &= \max(A[1,3], V_1 + A[1,3-w_1]) = 7 \end{aligned}$$

i\j	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	6	7
3	0			

Table 5.5: After adding the second row

Recursion for third row:

$$\begin{aligned} A[3,1] &= A[2,1] = 1 \\ A[3,2] &= A[2,2] = 6 \\ A[3,3] &= A[2,3] = 7 \end{aligned}$$

It can be observed the maximum possible value is 7 which is the bottom right entry of the table.

```
def knapSack(W, wt, val):
    n = len(val)
    # Initialize the table with all zeros.
```

i\j	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	6	7
3	0	1	6	7

Table 5.6: After adding the third row

```
    table = [[0 for j in range(W+1)] for i in range(n+1)]

    for i in range(n + 1):
        for j in range(W + 1):
            if i == 0 or j == 0:
                table[i][j] = 0
            elif wt[i-1] <= j:
                table[i][j] = max(val[i-1] + table[i-1][j-wt[i-1]],
                    table[i-1][j])
            else:
                table[i][j] = table[i-1][j]
    return table[n][W]

if __name__ == "__main__":
    val = [50, 100, 150, 200]
    wt = [8, 16, 32, 40]
    W = 64

    print(knapSack(W, wt, val))
```

Output

```
350
```

Second solution: Top-down approach with recursion

```
def knapSack(W, wt, val, n):
    # Base case
    if n == 0 or W == 0:
        memo[n][W] = 0
        return 0

    # Checking if the result is precalculated and returning it.
    if memo[n][W] != -1:
        return memo[n][W]

    # If weight of current element is less than or equal to
        capacity we can either include or exclude the item.
    if wt[n-1] <= W:
        memo[n][W] = max(val[n-1] + knapSack(W-wt[n-1], wt, val,
```

```
            n-1), knapSack(W, wt, val, n-1))
    # If weight of current element is greater than the capacity we
        will not include it thus returning just the ignoring part.
    else:
        memo[n][W] = knapSack(W, wt, val, n-1)
    return memo[n][W]

if __name__ == "__main__":
    val = [50, 100, 150, 200]
    wt = [8, 16, 32, 40]
    W = 64
    n = len(val)

    # The memory defined to store the result of the subproblem
        outside the function
    memo = [[-1 for j in range(W+1)] for i in range(n+1)]

    print(knapSack(W, wt, val, n))
```

Output

```
350
```

5.5 Longest common subsequence (LCS)

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", "acefg", ... etc are subsequences of "abcdefg".

For example, consider the two following sequences, X and Y:

$$X = \text{"ABCBDAB"}$$

$$Y = \text{"BDCABA"}$$

The length of the LCS is 4, LCS are "BDAB", "BCAB", and "BCBA"

A naive solution is to check if every subsequence of X to see if it is also a subsequence of Y. As there are 2^m subsequences possible of X, the *time complexity* of this solution would be $O(n.2^m)$, where m is the length of the first string and n is the length of the second string.
The LCS problem has optimal substructure. So we can apply the dynamic programming approach. Let's consider two sequences, $X = x_1x_2x_3 \cdots x_m$ and $Y = y_1y_2y_3 \cdots y_n$, of length m and n that both end in the same element.

- If X and Y both have the same element at the end, then we can say that:

$$LCS(x_1x_2\cdots x_m, y_1y_2\cdots y_n) = 1 + LCS(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_{n-1})$$

- If X and Y do not have the same element at the end, then the LCS of X and Y is the longer of the two sequences $LCS(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_n)$ and $LCS(x_1x_2\cdots x_m, y_1y_2\cdots y_{n-1})$

$$LCS(x_1x_2\cdots x_m, y_1y_2\cdots y_n) =$$
$$\max(LCS(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_n), LCS(x_1x_2\cdots x_m, y_1y_2\cdots y_{n-1}))$$

We continue with this recursion till we get to the first element.

First solution: Top-down approach
Time complexity: $O(nm)$: Since there are $(m+1)(n+1)$ entries, so the total number of recursive calls are at most $2(m+1)(n+1)+1$. So the *time complexity* is $O(mn)$.
Space complexity: $O(nm)$

```
def lcs(X, Y, m, n, memo = {}):
    key = str(m) + " " + str(n)
    if memo.get(key) is not None:
        return memo[key]

    # Base case
    if m == 0 or n == 0:
        memo[key] = 0
        return 0

    # If the last character of 'X' and 'Y' match
    if X[m - 1] == Y[n - 1]:
        memo[key] = lcs(X, Y, m - 1, n - 1) + 1
        return memo[key]
    # If the last character of 'X' and 'Y' don't match
    else:
        memo[key] = max(lcs(X, Y, m, n - 1, memo), lcs(X, Y, m - 1,
            n, memo))

    return memo[key]


if __name__ == '__main__':
    X = 'ABCBDAB'
    Y = 'BDCABA'

    print('The length of the LCS is', lcs(X, Y, len(X), len(Y)))
```

Output

```
The length of the LCS is 4
```

Second solution: Bottom-up approach (tabulation)
Numerical example: Find the length of common subsequence between X ='ACA' and Y ='CBDA'.

Let $dp[i][j]$ indicates the length of the longest sequence using first i characters of string X and starting j characters of string Y. Initial table:

$$dp[0][0] = dp[1][0] = dp[2][0] = dp[3][0] = 0$$
$$dp[0][0] = dp[0][1] = dp[0][2] = dp[0][3] = dp[0][4] = 0$$

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				

Table 5.7: Initial table

Recursion for first row:

$$dp[1][1] = \max(dp[0][1], dp[1][0])) = 0$$
$$dp[1][2] = \max(dp[0][2], dp[1][1])) = 0$$
$$dp[1][3] = \max(dp[0][3], dp[1][2])) = 0$$
$$dp[1][4] = dp[0][3] + 1 = 1$$

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				

Table 5.8: Recursion for first row

Similarly we can continue further with the following rule:

1. If the character correspond to the row and the character corresponding to the column are same, fill the cell with the **value in the cell diagonally top-left plus one**.

2. Otherwise, fill the cell with the **maximum value among the left cell and the just cell above this**.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0				

Table 5.9: Recursion for second row

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2

Table 5.10: Recursion for third row

It can be observed the LCS is 2 which is the bottom right entry of the table.

```
def solve(X, Y):
    m = len(X)
    n = len(Y)
    # Defining the table with all zero
    dp = [[0 for j in range(n+1)] for i in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):

            # Base case
            if i == 0 or j == 0:
                dp[i][j] = 0
            else:
                # If the last character of 'X' and 'Y' match
                if X[i-1] == Y[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                # If the last character of 'X' and 'Y' don't match
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

```
    return dp[m][n]


if __name__ == '__main__':
    X = 'ABCBDAB'
    Y = 'BDCABA'

    # print('The length of the LCS is', lcs(X, Y, len(X), len(Y)))
    print('The length of the LCS is', solve(X, Y))
```

Output

```
The length of the LCS is 4
```

6

More Algorithms

6.1 Find smallest and largest number

Given an input array of integers, find both smallest and largest number present in the array.

While traversing through the array keep track of maximum and minimum numbers found so far and when you reach the end of the array, then you will get smallest and largest numbers present in the array.

Time complexity: $O(n)$
Space complexity: $O(1)$

```
def findMinMax(arr):
    maxValue = 0
    minValue = 0

    for item in arr:
        if item > maxValue:
            maxValue = item
        if item < minValue:
            minValue = item

    print("Max value: ", maxValue)
    print("Min value: ", minValue)
```

6.2 Swap values without using temporary variable

Write an algorithm to swap two given numbers without using a temporary variable.

First solution: Using addition and subtraction:

```
a = 5
b = 10

a = a + b
b = a - b # This will act like (a+b) - b, and now b equals a.
a = a - b # This will act like (a+b) - a, and now a equals b.

print("a =", a)
print("b =", b)
```

Output

```
a = 10
b = 5
```

Second solution: In Python, there is a simple construct to swap variables.

```
a, b = b, a
```

6.3 Array chunking

Given an array `arr` and chunk size `L`, divide the array into many sub-arrays where each sub-array is of given chunk size.

Time complexity: $O(n)$
Space complexity: $O(n)$

```
def chunk(arr, L):
    output = []
    index = 0

    while index < len(arr):
        output.append(arr[index:index+L])
        index += L

    return output

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    L = 4
    output_list = chunk(arr, L)
    print(output_list)
```

Output

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9]]
```

6.4 Get nth prime number

Write a program to get the nth prime number.

Time complexity: The *prime number theorem*[1] is equivalent to the statement that the nth prime number p_n satisfies $p_n \sim n \log n$. We therefore need to have $n \log n$ iteration. In each iteration, we need to check if the number is a prime or not in $O(n)$ time. So the *time complexity* will be of the order

$$\sum_{k=1}^{n \log n} k = \frac{(n \log n)(n \log n + 1)}{2} \sim O(n^2 (\log n)^2)$$

Space complexity: $O(1)$

```
def isPrime(number):
    # Assuming 1 is not a prime
    if number == 1:
        return False
    for i in range(2, number//2 + 1):
        if number % i == 0:
            return False
    return True

def main(n):
    primeCounter = 0
    iteration = 0
    while primeCounter < n:
        iteration += 1
        if isPrime(iteration):
            primeCounter += 1
            # print(primeCounter, iteration)
    print(iteration)

if __name__ == "__main__":
    n = 1001
    main(n)
```

Output

```
7927
```

[1] The number of primes less than or equal to n (a.k.a. **prime counting function**) is given by

$$\Pi(n) \sim \frac{n}{\log n}$$

6.5 FizzBuzz problem

Write a program that prints the numbers from 1 to `N` and for multiples of 3 print 'Fizz' instead of the number, for the multiples of 5 print 'Buzz' and finally print 'FizzBuzz' at the multiple of both 3 and 5.

```
def fizzbuzz(N):
    for i in range(1, N+1):
        # For number divisible by both 3 & 5, print 'FizzBuzz'
        if (i % 3 == 0) and (i % 5 == 0):
            print("FizzBuzz")
            continue
        # For number divisible by 3, print 'Fizz'
        elif i % 3 == 0:
            print("Fizz")
            continue
        # For number divisible by 5, print 'Buzz'
        elif i % 5 == 0:
            print("Buzz")
            continue
        print(i)

if __name__ == "__main__":
    fizzbuzz(15)
```

Output

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

6.6 Anagram checker

Write a function to check if two given strings are anagrams[2]

First solution: By sorting the strings and comparing:
Time complexity: $O(n \log n)$ since the most efficient sorting will take $O(n \log n)$ time while the comparison will take $O(n)$ time.
Space complexity: $O(1)$

```
def anagram_check(str1, str2):
    # Check if length is same or not
    if len(str1) != len(str2):
        print(str1 + " and " + str2 + " are not anagram.")
        return

    # Sort the strings
    sorted_str1 = sorted(str1)
    sorted_str2 = sorted(str2)

    # If sorted char arrays are same
    if sorted_str1 == sorted_str2:
        print(str1 + " and " + str2 + " are anagram.")
    else:
        print(str1 + " and " + str2 + " are not anagram.")


if __name__ == "__main__":
    str1 = "race"
    str2 = "care"
    anagram_check(str1, str2)
```

Output

```
race and care are anagram.
```

Second solution: By building character map for each string and then comparing the maps.
Time complexity: $O(n)$ since building character map will take $O(n)$ time plus the comparison will take $O(n)$ time.
Space complexity: $O(n)$ for the requirements of the character maps.

```
def anagram_check(str1, str2):
    # Build the charMap of the two strings
    charMap1 = buildCharMap(str1)
    charMap2 = buildCharMap(str2)
```

[2]Two strings are said to be **anagram** if we can form one string by arranging the characters of another string. For example, 'Race' and 'Care'. Here, we can form 'Race' by arranging the characters of 'Care'.

```
    # Check if length is same
    if (len(charMap1.keys()) != len(charMap2.keys())):
        print(str1 + " and " + str2 + " are not anagram.")
        return

    # Check both the dicts
    for ch in charMap1.keys():
        if charMap1[ch] != charMap2[ch]:
            print(str1 + " and " + str2 + " are not anagram.")
            return

    print(str1 + " and " + str2 + " are anagram.")


# Function to build the character map for a string
def buildCharMap(input_str):
    charMap = {}
    for ch in input_str:
        if ch in charMap:
            charMap[ch] += 1
        else:
            charMap[ch] = 1

    return charMap


if __name__ == "__main__":
    str1 = "race"
    str2 = "care"
    anagram_check(str1, str2)
```

Output

```
race and care are anagram.
```

6.7 Missing number

Given an array of $n-1$ distinct integers in the range of 1 to n, find the missing number in it in linear time.

For example, consider array $\{1, 2, 3, 4, 5, 7, 8, 9, 10\}$ whose elements are distinct and within the range of 1 to 10. The missing number is 6.

We can solve the problem using the formula for sum of first n natural numbers.

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

The idea is to find the sum of integers between 1 and n using the above formula where n is the array's size + 1. Also calculate the actual sum of integers in the array. Now the missing number would be the difference between the two.
Time complexity: $O(n)$
Space complexity: $O(1)$

```
def getMissingNumber(arr):
    # n = array's length + 1
    n = len(arr) + 1

    # Compute the sum of integers between 1 and n
    total = n*(n+1)/2

    # The missing number is the difference between the expected sum
        and the actual sum of integers in the list
    return total - sum(arr)
```

6.8 Find pair that sums up to k

Given an array `arr` and target `k`, find pair of elements of the array `arr` that sums up to `k`

Brute force solution: By checking all pairs
Time complexity: $O(n^2)$
Space complexity: $O(1)$

```
def findPair(arr, k):
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] + arr[j] == k:
                return (i, j)
    return None
```

Second solution: By sorting the array
Time complexity: $O(n \log n)$
Space complexity: $O(1)$ if you are allowed to modify the input, otherwise $O(n)$

```
def findPair(arr, k):
    arr.sort()
    left = 0
    right = len(arr) - 1
    while left < right:
        if arr[left] + arr[right] == k:
            return (left, right)
        elif arr[left] + arr[right] < k:
            left += 1
        else:
            right -= 1
    return None
```

Third solution: By using a dictionary
Time complexity: $O(n)$
Space complexity: $O(n)$

```
def findPair(arr, k):
    '''
    nums_map will store the each element of arr as key and index as
        value in each iteration.
    If arr = [1,3,7,9,2], then nums_map = {1: 0, 3: 1, 7: 2, 9: 3,
        2: 4}
    '''
    nums_map = {}
    for (i, element) in enumerate(arr):
        if nums_map.get(k - element):
```

```
            return (nums_map.get(k - element), i)
        else:
            nums_map[element] = i
    return None
```

6.9 First repeating character

Given a string `a_str`, find the first repeating character (i.e. the character that we have seen before as we inspect the characters of the string from left)

Brute force solution:
Time complexity: $O(n^2)$
Space complexity: $O(1)$

```
def firstRepeatingCharacter(a_str):
    for i in range(len(a_str)):
        for j in range(i):
            if a_str[i] == a_str[j]:
                return a_str[i]
    return None
```

By using a dictionary:
Time complexity: $O(n)$
Space complexity: $O(n)$

```
def firstRepeatingCharacter(a_str):
    visited = {}
    for ch in a_str:
        if visited.get(ch):
            return ch
        else:
            visited[ch] = True
    return None
```

6.10 Remove duplicates

Given an array of integers `arr`, create a function that returns an array that contains the values of `arr` without duplicates.

Brute force solution:
Time complexity: $O(n^2)$
Space complexity: $O(n)$

```
def removeDuplicates(arr):
    output = []
    for element in arr:
        if element not in output:
            output.append(element)
    return output
```

Second solution: By sorting the array

After sorting the array all the duplicate elements are grouped together e.g. `[1, 1, 1, 3, 3, 4]`. So we only need to add the character to the output list if the current character does not match the previous character in the array.
Time complexity: $O(n \log n)$
Space complexity: $O(n)$

```
def removeDuplicates(arr):
    if len(arr) == 0:
        return []
    arr.sort()
    output = [arr[0]]
    for i in range(1, len(arr)):
        if arr[i] != arr[i - 1]:
            output.append(arr[i])
    return output
```

Third solution: By using a dictionary
Time complexity: $O(n)$
Space complexity: $O(n)$

```
def removeDuplicates(arr):
    visited = {}
    for element in arr:
        visited[element] = True
    return list(visited.keys())
```

6.11 Find duplicate value

Given an array `arr` of size $n+1$ which contains all integers between 1 and n, find one of the multiple repeating elements in the array.

Brute force solution:
Time complexity: $O(n^2)$
Space complexity: $O(1)$

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

Second solution: By sorting the array
Time complexity: $O(n \log n)$
Space complexity: $O(1)$ if you are allowed to modify the input, otherwise $O(n)$

```
def findDuplicate(arr):
    arr.sort()
    for i in range(1, len(arr)):
        if arr[i] == arr[i - 1]:
            return arr[i]
```

Third solution: By using a dictionary
Time complexity: $O(n)$
Space complexity: $O(n)$

```
def findDuplicate(arr):
    visited = {}
    for element in arr:
        if visited.get(element):
            return element
        else:
            visited[element] = True
```

Fourth solution: By using Floyd's cycle detection algorithm
Use the function $f(x) =$ `arr[x]` to construct the sequence:

```
arr[0], arr[arr[0]], arr[arr[arr[0]]], arr[arr[arr[arr[0]]]] ...
```

Starting from `arr[0]`, it will produce a linked list with a cycle. The cycle appears because `arr[x]` contains duplicate elements (at least one). The duplicate value is an entrance to the cycle. Given below is an example to show how cycle exists:

Let the array be `arr = [2, 6, 4, 1, 3, 1, 5]`

Index	0	1	2	3	4	5	6
arr	2	6	4	1	3	1	5

Starting from index 0, the traversal looks as follows:

`arr[0] = 2 --> arr[2] = 4 --> arr[4] = 3 --> arr[3] = 1 --> arr[1] = 6 --> arr[6] = 5 --> arr[5] = 1`

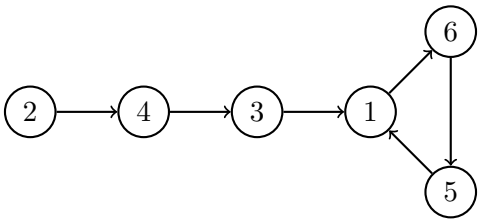


Figure 6.1: Floyd's cycle detection algorithm

- Algorithm consists of two parts and uses two pointers, usually called `tortoise` and `hare`.
- `hare` moves twice as fast as `tortoise` i.e. `tortoise = arr[tortoise]` and `hare = arr[arr[hare]]`.
- Since the `hare` goes fast, it would be the first one who enters the cycle and starts to run around the cycle.
- At some point, the `tortoise` enters the cycle as well, and since it's moving slower the `hare` catches the `tortoise` up at some intersection point.
- Note that the intersection point may not be the cycle entrance in the general case, but at somewhere middle in cycle.
- Move `tortoise` to the starting point of sequence and `hare` remains within cycle and both move with the same speed i.e. `tortoise = arr[tortoise]` and `hare = arr[hare]`. Now they intersect at duplicate element.

Time complexity: $O(n)$
Space complexity: $O(1)$

```
def findDuplicate(arr):
    # Initialise variables
    tortoise = arr[0]
    hare = arr[0]

    # Loop till we find the
    # duplicate element
    while True:
        tortoise = arr[tortoise]
        # Hare moves with twice
        # the speed of tortoise
        hare = arr[arr[hare]]
        if (tortoise == hare):
            break

    tortoise = arr[0]

    # Loop to get start point
    # of the cycle as start
    # point will be the duplicate
    # element
    while (tortoise != hare):
        tortoise = arr[tortoise]
        hare = arr[hare]

    # Return the duplicate element
    return tortoise
```

Why does Floyd's Cycle Detection Algorithm works?

Let us assume that the linked list has a cycle that starts at x_4. As per the algorithm, we have two traversal pointers `hare` and `tortoise` meet each other at x_7 for the first time. From there the `tortoise` starts from the starting point x_1 and both the `hare` and `tortoise` are then moving by one node. The next point they meet will be the start of the cycle i.e. x_4.

Let,

x = Path distance between the start of the linked list and the start of the cycle.

y = Path distance between the start of the cycle to the point where `hare` and `tortoise` first meet each other.

L = Total length of the cycle.

When they first meet each other, `hare` has covered a distance of

$$x + k_h \times L + y$$

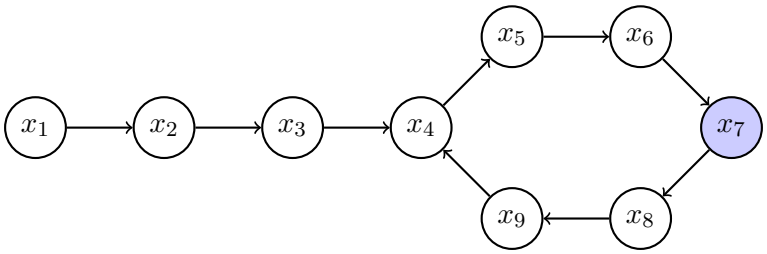


Figure 6.2: Floyd's cycle detection algorithm: How it works

where k_h is a non-negative integer and `tortoise` has covered a distance of

$$x + k_t \times L + y$$

where k_t is another non-negative integer.
Since `hare` travels twice the speed of `tortoise`, it would travel twice the distance. So,

$$\begin{aligned} x + k_h L + y &= 2 \times (x + k_t L + y) \\ \Rightarrow x + y &= (k_h - 2k_t)L \\ \Rightarrow x + y &= kL, \text{ where } k = k_h - 2k_t \text{ is an integer} \\ \Rightarrow x &= kL - y \end{aligned}$$

Now, if we move `tortoise` the start of the linked list, then after covering distance x it will come to the starting node of the cycle. The `hare` will cover $x = kL - y$ distance at the same time. The `hare` already covered y distance from the start of the cycle. After travelling x distance the `hare` will complete the cycle (may be multiple times) since $y + x = kL$ and reach the starting node where it will meet the `tortoise`.

6.12 Maximum sub-array sum

For a given array `arr` of integers (containing both positive as well as negative integers) find the largest possible sum of a contiguous sub-array.

For example, for the array `[-2, 1, 3, 4, -1, 2, 1, -5, 4]`, the contiguous sub-array with the largest sum is [4, -1, 2, 1], with sum 6.

Brute force solution:
Time complexity: $O(n^3)$
Space complexity: $O(1)$

```
def maximumSubarray(arr):
    maxSum = float('-inf')
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            # Compute the sum of sub-array starting at index i and
                ending at index j
            actualSum = 0
            for k in range(i, j + 1):
                actualSum += arr[k]
            maxSum = max(maxSum, actualSum)
    return maxSum
```

Second solution: Brute force with cumulative sum
Time complexity: $O(n^2)$
Space complexity: $O(1)$

```
def maximumSubarray(arr):
    maxSum = float('-inf')
    for i in range(len(arr)):
        cumulativeSum = 0
        for j in range(i, len(arr)):
            cumulativeSum += arr[j]
            maxSum = max(maxSum, cumulativeSum)
    return maxSum
```

Third solution: By using **Kadane's algorithm**
We use dynamic programming to solve the problem. Suppose we know the maximum sum of the sub-array ending on the $(i-1)$th position which is S_{i-1}.
Now, to find the maximum sum of the sub-array ending on the ith position i.e. S_i, it makes sense to add S_{i-1} to the ith element A_i only if

$$S_{i-1} + A_i > A_i$$

or,

$$S_{i-1} > 0$$

Otherwise, the ith element is good being on its own as the sub-array sum ending on the ith position.
So, the recurrence relation is:

$$S_i = \begin{cases} A_i + S_{i-1}, & \text{if } S_{i-1} > 0 \\ A_i, & \text{otherwise} \end{cases}$$

or,

$$S_i = \max(A_i, S_{i-1} + A_i)$$

Time complexity: $O(n)$
Space complexity: $O(1)$

```
def maximumSubarray(arr):
    globalSum = float('-inf')
    localSum = 0
    for x in arr:
        localSum = max(x, localSum + x)
        globalSum = max(globalSum, localSum)
    return globalSum
```

6.13 Rotation checker

How do you check if two strings are a rotation of each other?

For example, suppose we have two strings `str1 = "HELLO"`, and `str2 = "LOHEL"`. Both of them have equal lengths and have the same characters. By rotating `"HELLO"` three position to the left we can obtain `"LOHEL"`. So, these strings are rotation of each other.
In order to solve this problem, we will firstly check if the strings are of same length. After that we will concatenate the first string with itself, then check whether the second one is present in the concatenated string or not. If the second string exists in the concatenated string, the strings are rotations of each other. In our example, for `str1 = "HELLO"`, the concatenated String will be `"HELLOHELLO"`. As we can see, this concatenated string contains the string `"LOHEL"` and thus, these strings are rotations of each other.

Time complexity: $O(n^2)$, since it depends on the process of finding the string `str2` in the concatenated string. For this we need to run two loops: one along the concatenated string and the other through `str2`.

Space complexity: $O(1)$

```
def checkRotation(str1, str2):
    # Check if lengths of two strings are equal or not
    if len(str1) != len(str2):
        return False

    # Storing concatenated string
    temp = str1 + str1

    if str2 in temp:
        # Returning true if 2nd string is present in concatenated
            string
        return True
    else:
        return False

if __name__ == "__main__":
    str1 = "HELLO"
    str2 = "LOHEL"

    if checkRotation(string1, string2):
        print("Given Strings are rotations of each other.")
    else:
        print("Given Strings are not rotations of each other.")
```

6.14 Reverse words in a given sentence

Write a function to reverse the order of words in a given sentence

Examples:

```
Input: "getting good at coding needs a lot of practice"
Output: "practice of lot a needs coding at good getting"
```

First solution: using a stack

Time complexity: $O(n)$ where n is the number of characters in the sentence. The complexity is due to the splitting of words into an array.
Space complexity: $O(m)$ due to the requirement of the stack where m is the number of words in the sentence.

```
def reverseWords(sentence):
    word_array = sentence.split(" ")
    stack = [] # Defining a stack

    # Pushing words into stack
    for word in word_array:
        stack.append(word)

    result = []
    while len(stack) > 0:
        # Poping words from stack and adding into result
        result.append(stack.pop())

    return " ".join(result)

if __name__ == "__main__":
    sentence = "i like this program very much"
    print(reverseWords(sentence))
```

Output

```
much very program this like i
```

Second solution: using *two pointer technique* (in in constant *space complexity*)

1. Convert the string into an array of strings, which will store the words.
2. Initialize the 2 pointers `left` and `right` at first and last index of the array respectively.

3. While the `left` pointer does not exceed the `right` pointer, swap the elements at the `left` and `right` pointer, move the `left` pointer forward and the `right` pointer backward by 1 place.

4. Finally, construct the sentence from the array return the final calculated string.

Time complexity: Swapping of words takes $O(m)$ time where m is the number of words in the sentence and splitting of words into an array takes $O(n)$ time where n is the number of characters in the sentence. Since $m < n$, the complexity is due to the splitting of words into an array and is given by $O(n)$.
Space complexity: $O(1)$.

```
def reverseWords(sentence):
    word_array = sentence.split(" ")
    left = 0
    right = len(word_array) - 1
    while left <= right:
    # Swapping the value of left and right index
        word_array[left], word_array[right] = word_array[right],
            word_array[left]
        left += 1
        right -= 1

    result = " ".join(word_array)
    return result

if __name__ == "__main__":
    sentence = "i like this program very much"
    print(reverseWords(sentence))
```

Output

```
much very program this like i
```

6.15 Check subsequence

Given two strings `str1` and `str2`, find if `str1` is a subsequence of `str2`.

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

```
Input: str1 = "AXY", str2 = "ADXCPY"
Output: True (str1 is a subsequence of str2)

Input: str1 = "AXY", str2 = "YADXCP"
Output: False (str1 is not a subsequence of str2)
```

The idea is simple, we traverse both strings from one side to the other (say, from rightmost character to leftmost or vice versa). If we find a matching character, we move ahead in both strings. Otherwise we move ahead only in `str2`.

First solution: Iterative approach

Time complexity: $O(n)$
Space complexity: $O(1)$

```
def isSubSequence(str1, str2):
    m = len(str1)
    n = len(str2)

    i = 0   # Index of str1
    j = 0   # Index of str2

    # Traverse both str1 and str2
    # Compare current character of str2 with first unmatched
        character of str1
    # If matched, then move ahead in str1
    while i < m and j < n:
        if str1[i] == str2[j]:
            i += 1
        j += 1

    # If all characters of str1 matched, then j is equal to m
    return i == m


if __name__ == "__main__":
    str1 = "amnraon"
```

```
    str2 = "administration"

    if isSubSequence(str1, str2):
        print(f"'{str1}' is a subsequence of '{str2}'")
    else:
        print(f"'{str1}' is a not subsequence of '{str2}'")
```

Output

```
'amnraon' is a subsequence of 'administration'
```

Second solution: Recursive approach

Time complexity: $O(n)$ since there will be a maximum of n number of recursive calls in the worst case.
Space complexity: $O(1)$

```
def isSubSequence(str1, str2, m, n):
    # m and n are the length of the str1 and str2 respectively

    # Base Cases
    if m == 0:
        return True
    if n == 0:
        return False

    # If last characters of two strings are matching, move both the
        pointers
    if str1[m-1] == str2[n-1]:
        return isSubSequence(str1, str2, m-1, n-1)

    # If last characters are not matching, move only the pointer
        for str2
    return isSubSequence(str1, str2, m, n-1)


if __name__ == "__main__":
    str1 = "amnraon"
    str2 = "administration"

    if isSubSequence(str1, str2, len(str1), len(str2)):
        print(f"'{str1}' is a subsequence of '{str2}'")
    else:
        print(f"'{str1}' is a not subsequence of '{str2}'")
```

Output

```
'amnraon' is a subsequence of 'administration'
```

6.16 Reversing a binary tree

Given a binary tree of integers, create a function that reverses its left to right in place. Using recursion:

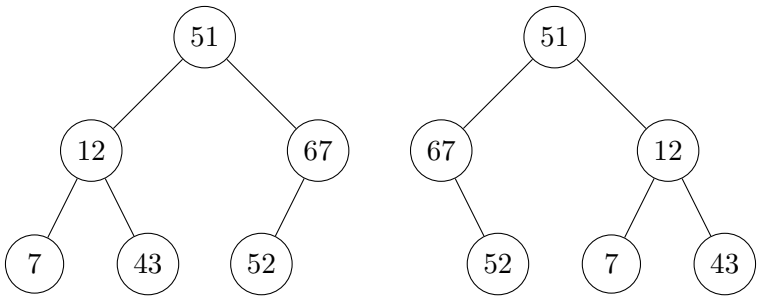


Figure 6.3: Reversing a binary tree

Time complexity: $O(n)$
Space complexity: $O(h)$ where h is the height (maximum depth from root) of the tree

```
class Tree_node:
    def __init__(self, data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right

def reverseTree(root):
   if root is None:
      return
   # Reversing the left and right sub-tree
   root.left, root.right = root.right, root.left
   reverseTree(root.left)
   reverseTree(root.right)

def main():
   # Creating the tree
   node_51 = Tree_node(51) # root node of the tree
   node_12 = Tree_node(12)
   node_67 = Tree_node(67)
   node_7 = Tree_node(7)
   node_43 = Tree_node(43)
   node_52 = Tree_node(52)
   node_51.left = node_12
   node_51.right = node_67
```

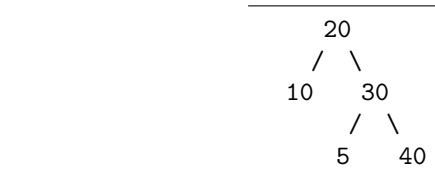
```
    node_12.left = node_7
    node_12.right = node_43
    node_67.left = node_52

    # Reversing the tree
    reverseTree(node_51)

if __name__ == "__main__":
    main()
```

6.17 Check for BST

Determine whether a given binary tree is a BST or not

The BST property "*Every node on the right subtree has to be larger than the current node and every node on the left subtree has to be smaller than the current node*" is the key to figuring out whether a tree is a BST or not.

The *greedy* algorithm – traverse the tree, at every node check whether the node contains a value larger than the value at the left child and smaller than the value on the right child – does not work for all cases. Consider the following tree:



In the tree above, each node meets the condition that the node contains a value larger than its left child and smaller than its right child hold, and yet it's not a BST: the value 5 is on the right subtree of the node containing 20, a violation of the BST property.

Instead of deciding based solely on a node's values and its children, we also need information flowing down from the parent. In the tree above, if we could remember about the node containing the value 20, we would see that the node with value 5 is violating the BST property contract. So, the condition we need to check at each node is:

1. If the node is the left child of its parent, it must be smaller than the parent, and it must pass down the value from its parent to its right subtree to make sure none of the nodes in that subtree is greater than the parent.

2. If the node is the right child of its parent, it must be larger than the parent, and it must pass down the value from its parent to its left subtree to make sure none of the nodes in that subtree is lesser than the parent.

Time complexity: $O(n)$ where n is the size of the tree.
Space complexity: $O(n)$ since there will be n number of recursive calls.

```
class Tree_node:
    def __init__(self, data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right
```

```
# Function to determine whether a given binary tree is a BST by
    keeping a valid range (starting from [-INFINITY, INFINITY]) and
    keep shrinking it down for each node as we go down recursively
def isBST(node, minKey, maxKey):

    # base case
    if node is None:
        return True

    # if the node's value falls outside the valid range
    if node.data < minKey or node.data > maxKey:
        return False

    # recursively check left and right subtrees with an updated
        range
    return isBST(node.left, minKey, node.data) and \
        isBST(node.right, node.data, maxKey)

# Function to determine whether a given binary tree is a BST
def checkForBST(root):
    if isBST(root, float("-Inf"), float("Inf")):
        print('The tree is a BST.')
    else:
        print('The tree is not a BST')

if __name__ == '__main__':
    '''
      _4__
     /    \
     2     6
    / \   / \
    1 3  5   7
    '''
    root = Tree_node(4)
    root.left = Tree_node(2)
    root.right = Tree_node(6)
    root.left.left = Tree_node(1)
    root.left.right = Tree_node(3)
    root.right.left = Tree_node(5)
    root.right.right = Tree_node(7)
    checkForBST(root)
```

Output

```
The tree is a BST.
```

6.18 Longest sub-string without repeating characters

Given a string `a_string` of alphanumeric characters, write a function that will return the longest sub-string without repeating characters.

Brute force solution:
Time complexity: $O(n^2)$
Space complexity: $O(n)$

```
def longestSubstringWithoutRepeating(a_string):
    # Maximum length
    max_len = 0

    for i in range(len(a_string)):
        # Default values in visited are false
        visited = [False] * 256

        for j in range(i, len(a_string)):
            # If current character is visited
            # Break the loop
            if (visited[ord(a_string[j])] == True):
                break
            # Else update the max_len if
            # this window is larger, and mark
            # current character as visited.
            else:
                max_len = max(max_len, j - i + 1)
                visited[ord(a_string[j])] = True
    return max_len
```

Second solution:

The idea is to scan the string from left to right, keeping track of the maximum length non-repeating character sub-string seen so far in a variable `max_len`. We also need a dictionary `last_idx` to store the last index of every character. When we traverse the string, to know the length of current window we need two pointers: (1) Starting pointer (`start`) and (2) Current pointer (`i`)

First we set the starting index to 0 and scan the string with the current pointer. During each character scan we store the index of that character in `last_idx`. In case we get a repeated character (which is already present in `last_idx`), we move the start pointer to `last_idx[a_string[i]] + 1` only if it is greater than previous start pointer. At each scan we also keep track of the maximum length `max_len`.
Time complexity: $O(n)$

Space complexity: $O(n)$

```
def longestSubstringWithoutRepeating(a_string):
    # last index of every character
    last_idx = {}
    max_len = 0
    # starting index of current
    # window to calculate max_len
    start = 0

    for i in range(0, len(a_string)):
        # Find the last index of str[i]
        # Update start (starting index of current window)
        # as maximum of current value of start and last
        # index plus 1
        if a_string[i] in last_idx:
            start = max(start, last_idx[a_string[i]] + 1)

        # Update result if we get a larger window
        max_len = max(max_len, i-start + 1)

        # Update last index of current char.
        last_idx[a_string[i]] = i

        # Print statement for understanding the concept
        print(a_string[i], 'i:', i, 'start:', start, 'max_len:',
            max_len, 'last_idx:', last_idx)

    return max_len

if __name__ == "__main__":
    a_string = "abxddcdbsd"
    print('Maximum length of longest sub-string =',
        longestSubstringWithoutRepeating(a_string))
```

Output:

```
a i: 0 start: 0 max_len: 1 last_idx: {'a': 0}
b i: 1 start: 0 max_len: 2 last_idx: {'a': 0, 'b': 1}
x i: 2 start: 0 max_len: 3 last_idx: {'a': 0, 'b': 1, 'x': 2}
d i: 3 start: 0 max_len: 4 last_idx: {'a': 0, 'b': 1, 'x': 2, 'd':
    3}
d i: 4 start: 4 max_len: 4 last_idx: {'a': 0, 'b': 1, 'x': 2, 'd':
    4}
c i: 5 start: 4 max_len: 4 last_idx: {'a': 0, 'b': 1, 'x': 2, 'd':
    4, 'c': 5}
d i: 6 start: 5 max_len: 4 last_idx: {'a': 0, 'b': 1, 'x': 2, 'd':
    6, 'c': 5}
b i: 7 start: 5 max_len: 4 last_idx: {'a': 0, 'b': 7, 'x': 2, 'd':
    6, 'c': 5}
```

```
s i: 8 start: 5 max_len: 4 last_idx: {'a': 0, 'b': 7, 'x': 2, 'd':
    6, 'c': 5, 's': 8}
d i: 9 start: 7 max_len: 4 last_idx: {'a': 0, 'b': 7, 'x': 2, 'd':
    9, 'c': 5, 's': 8}
Maximum length of longest sub-string = 4
```

6.19 Finding a peak element

Given an array of integers. Find the index of a peak element in it. An array element is a peak if it is not smaller than its neighbours. For corner elements, we need to consider only one neighbour.

For the array `arr = [5, 10, 20, 15]`, the element 20 is the peak which has neighbours 10 and 15 both less than 20. For the array `arr =[10, 20, 15, 2, 23, 90, 67]`, the peak can be 20 or 90. The given problem is to find only one such peak. Following corner cases give better idea about the problem.

- If input array is sorted in strictly increasing order, the last element is always a peak element.
- If the input array is sorted in strictly decreasing order, the first element is always a peak element.
- If all elements of input array are same, every element is a peak element.

It is clear that there is always a peak element in any given array.

Naive solution: By traversing the whole array.
Time complexity: $O(n)$
Space complexity: $O(1)$

```
# Find the peak element in the array
def findPeak(arr) :
    L = len(arr)
    # Single element array will always contain the peak element
    if (L == 1) :
      return 0

    # Checking if the first element is a peak element
    if (arr[0] >= arr[1]):
        return 0

    # Checking if the last element is a peak element
    if (arr[L - 1] >= arr[L - 2]):
        return L - 1

    # check for every other element
    for i in range(1, L - 1) :

        # check if the neighbors are smaller
        if (arr[i] >= arr[i - 1] and arr[i] >= arr[i + 1]) :
            return i
```

Second solution: The idea is based on the technique of Binary Search by finding the middle element. If the element on the right side is greater than

the middle element then there is always a peak element on the right side of the array `arr[mid+1:]`. On the other hand, if the element on the left side is greater than the middle element then there is always a peak element on the left side of the array `arr[:mid+1]`.

Time complexity: $O(\log n)$
Space complexity: $O(1)$

```
def findPeak(arr):
    left = 0
    right = len(arr)-1
    while left < right:
        mid = (left + right)//2
        if arr[mid] < arr[mid+1]:
            left = mid+1
        else:
            right = mid
    return left
```

Third solution: Using *Divide and Conquer* method. The idea is to find the middle element similar to the previous case. The middle element will divide the array into two sub-arrays: left sub-array `arr[:mid+1]` and right sub-array `arr[mid+1:]`. If the element on the right side is greater than the middle element then there is always a peak element on the right sub-array. On the other hand, if the element on the left side is greater than the middle element then there is always a peak element on the left sub-array. Apply the same function on the sub-array (which contains the peak element) recursively.

Time complexity: $O(\log n)$
Space complexity: $O(1)$

```
def findPeakRec(arr, left, right):
    if left >= right:
        return left
    mid = (left + right)//2
    if arr[mid] < arr[mid+1]:
        return findPeakRec(arr, mid+1, right)
    else:
        return findPeakRec(arr, left, mid)

def findPeak(arr):
    L = len(arr)
    return findPeakRec(arr, 0, L-1)
```

6.20 Ways to climb a stair

A person is climbing up a staircase having n steps in total and can hop either 1 step, 2 steps, or 3 steps at a time. Write a function to count how many possible ways the person can climb up the stairs.

There are n stairs, and a person is allowed to jump next stair, skip one stair or skip two stairs. To reach a stair n, a person has to jump either from $(n-1)$, $(n-2)$ or $(n-3)$th stair. So if $f(n)$ indicates the number of steps to climb upto n stairs, then

$$f(n) = f(n-1) + f(n-2) + f(n-3)$$

We can now just apply recursion to get the result.

First solution: Using normal recursion
Time complexity: $O(3^n)$
Space complexity: $O(n)$

```
def countWays(n):
    if n < 0:
        return 0
    # If the value of n is equal to zero then there is only one way
    elif n == 0:
        return 1
    else:
        return countWays(n - 3) + countWays(n - 2) + countWays(n - 1)
```

Second solution: Dynamic programming with memoized approach
Time complexity: $O(n)$
Space complexity: $O(n)$

```
def countWays(n, memo = {}):
    if n < 0:
        return 0
    # If the value of n is equal to zero then there is only one way
    elif n == 0:
        memo[n] = 1
        return memo[n]
    else:
        if memo.get(n) != None:
            return memo[n]
        memo[n] = countWays(n-3, memo) + countWays(n-2, memo) +
            countWays(n-1, memo)
        return memo[n]
```

Third solution: Dynamic programming with tabulation approach
Time complexity: $O(n)$

Space complexity: $O(n)$

```
def countWays(n):
    # Create a table with n+1 elements initialized to zero
    dp = [0] * (n + 1)
    # Base cases
    dp[0] = 1
    dp[1] = 1
    dp[2] = 2 # Since 2 = 1 + 1 or 2 = 2

    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]

    return dp[n]
```

If the number of possible steps is not explicitly specified but given as a parameter `possibleSteps`, then we can use the following script:

```
def countWays(n, possibleSteps):
    dp = [0] * (n+1)
    dp[0] = 1
    for i in range(1, n+1):
        numWays = 0
        for steps in possibleSteps:
            if (i-steps) >= 0: # To eliminate the possibility of
                negative index
                numWays += dp[i-steps]
        dp[i] = numWays
    return dp[n]

if __name__ == "__main__":
    possibleSteps = {1, 2, 3}
    print(countWays(6, possibleSteps))
```

Output

```
24
```

Time complexity: $O(mn)$
Space complexity: $O(n)$

6.21 Coin change

Given a value n and an infinite supply of coins with given denominations (m number of types) each, find how many ways can we make the change?

Suppose we are given a target value n and a set of denominations $S = \{d_1, d_2, d_3\}$. Let $f(n)$ indicates the number of ways we can make the change. Now let's pick the coin d_1. Then for the remaining target is $n - d_1$, for which we have number of ways to make the changes. Similarly if we pick the coin d_2. Then for the remaining target is $n - d_2$, for which we have number of ways to make the changes and so on. So we can write

$$f(n) = f(n - d_1) + f(n - d_2) + f(n - d_3)$$

We can now just apply recursion to get the result.
First solution: Dynamic programming with memoized approach

Time complexity: $O(mn)$
Space complexity: $O(n)$

```
def countWays(n, denominations):
    dp = [0] * (n+1)
    dp[0] = 1
    for i in range(1, n+1):
        numWays = 0
        for d in denominations:
            if (i-d) >= 0: # To eliminate the possibility of
                negative index
                numWays += dp[i-d]
        dp[i] = numWays
    return dp[n]

if __name__ == "__main__":
    denominations = {1, 2, 3}
    print('Number of ways:', countWays(6, denominations))
```

Output

```
Number of ways: 24
```

Second solution: Dynamic programming with tabulation approach

Time complexity: $O(mn)$
Space complexity: $O(n)$

```
def countWays(n, denominations, memo = {}):
```

```
    if n < 0:
        return 0
    # If the value of n is equal to zero then there is only one way
    elif n == 0:
        memo[n] = 1
        return memo[n]
    else:
        if memo.get(n) != None:
            return memo[n]
        numWays = 0
        for d in denominations:
            if (n-d) >= 0:
                numWays += countWays(n-d, denominations, memo)
        memo[n] = numWays
        return memo[n]

if __name__ == "__main__":
    denominations = {1, 2, 3}
    print('Number of ways:', countWays(6, denominations))
```

Output

```
Number of ways: 24
```

6.22 Minimum number of coins for change

Given a value n and an infinite supply of coins with given denominations (m number of types) each, find the number of minimum coins needed to make the change? If it's not possible to make a change, print '-1'

This problem is a variation of the **Coin change** problem. Here instead of finding the total number of possible solutions, we need to find the solution with the minimum number of coins. Suppose we are given a target value n and a set of denominations $S = \{d_1, d_2, d_3\}$. Let $\phi(n)$ indicates the minimum number of coins required to make the changes for value n. Then if we pick the coin d_1, then for the remaining value $n - d_1$, we need $\phi(n - d_1)$ number of coins. So if we select d_1, we can write

$$\text{Number of min coins given } d_1 \text{ is selected} = 1 + \phi(n - d_1)$$

Similarly if we select d_2, we can write

$$\text{Number of min coins given } d_2 \text{ is selected} = 1 + \phi(n - d_2)$$

Finally if we select d_3, we can write

$$\text{Number of min coins given } d_3 \text{ is selected} = 1 + \phi(n - d_3)$$

Therefore we can write for $\phi(n)$,

$$\phi(n) = 1 + \min_{i=1,2,3} (\phi(n - d_i))$$

We can now just apply recursion to get the result.

First solution: Using normal recursion
Time complexity: $O(m^n)$
Space complexity: $O(n)$

```
def coinChangeRec(n, denominations):
    if n == 0:
        return 0
    minCoins = float("inf")
    # Initializing with a very high value
    # minCoins will store the minimum of (1 + coinChangeRec(n-i,
        denominations))
    for coin in denominations:
        if (n-coin) >= 0:
            minCoins = min(minCoins, 1 + coinChangeRec(n-coin,
                denominations))
    return minCoins
```

```
def coinChange(n, denominations):
    # This function is used to return '-1' if no change is possible
    minCoins = coinChangeRec(n, denominations)
    if minCoins == float("inf"):
        return -1
    else:
        return minCoins

if __name__ == "__main__":
    denominations = {1, 2, 5}
    print('Minimum number of denominations:', coinChange(9,
        denominations))
```

Output

```
Minimum number of denominations: 3
```

Second solution: Using dynamic programming with memoized approach
Time complexity: $O(mn)$
Space complexity: $O(n)$

```
def coinChangeRec(n, denominations, memo = {}):
    if n == 0:
        return 0
    minCoins = float("inf")
    # Initializing with a very high value
    # minCoins will store the minimum of (1 + coinChangeRec(n-i,
        denominations))
    for coin in denominations:
        if (n-coin) >= 0:
            minCoins = min(minCoins, 1 + coinChangeRec(n-coin,
                denominations))
    memo[n] = minCoins
    return memo[n]

def coinChange(n, denominations):
    # This function is used to return '-1' if no change is possible
    minCoins = coinChangeRec(n, denominations)
    if minCoins == float("inf"):
        return -1
    else:
        return minCoins

if __name__ == "__main__":
    denominations = {1, 2, 5}
    print('Minimum number of denominations:', coinChange(9,
        denominations))
```

Third solution: Using dynamic programming with tabulation approach
Time complexity: $O(mn)$
Space complexity: $O(n)$

```
def coinChange(n, denominations):
    dp = [float("inf")] * (n+1)
    # dp[i] indicates the minimum number of coins required to make
        the changes for value i
    dp[0] = 0
    for i in range(1, n+1):
        minCoins = float("inf")
        for coin in denominations:
            if (i-coin) >= 0:
                minCoins = min(minCoins, 1 + dp[i-coin])
        dp[i] = minCoins

    if dp[n] == float("inf"):
        return -1
    else:
        return dp[n]
```

6.23 Cutting a rod

Given a rod of length n and a list of rod prices of length i, where $1 \leq i \leq n$, find the optimal way to cut the rod into smaller rods to maximize profit.

For example, if the length of the rod is 4 and the values of different pieces are given as the following, then the maximum obtainable value is 10 (by cutting in two pieces of lengths 2 each).

```
Input:
length = [1, 2, 3, 4, 5, 6, 7, 8]
price = [1, 5, 8, 9, 10, 17, 17, 20]

Rod length: 4

Cut            Profit
4               9
1, 3           (1 + 8) = 9
2, 2           (5 + 5) = 10
3, 1           (8 + 1) = 9
1, 1, 2        (1 + 1 + 5) = 7
1, 2, 1        (1 + 5 + 1) = 7
2, 1, 1        (5 + 1 + 1) = 7
1, 1, 1, 1     (1 + 1 + 1 + 1) = 4

Best: Cut the rod into two pieces of length 2 each to gain revenue
    of 5 + 5 = 10
```

Let p_i is the price for rod of length i and r_i indicates the maximum profit that can be obtained by cutting a rod of length i. The idea is simple - one by one, partition the given rod of length n into two parts: i and $n-i$. Recur for the rod of length $n-i$ but don't divide the rod of length i any further. Finally, take the maximum of all values. This yields the following recursive relation:

$$r(n) = \max_{1 \leq i \leq n} (p(i) + r(n-i))$$

First solution: Dynamic programming using memoized approach
Time complexity: $O(n^2)$
Space complexity: $O(n)$

```
def rodCut(price, n, memo={}):
    # Base case
    if n == 0:
        return 0

    elif memo.get(n) != None:
        return memo[n]
```

```
    maxProfit = 0
    # One by one, partition the given rod of length 'n' into two
        parts of length (1, n-1), (2, n-2), (3, n-3), ... ,(n-1, 1),
        (n, 0) and take maximum
    for i in range(1, n+1):
        # Rod of length 'i' has a cost 'price[i-1]'
        cost = price[i-1] + rodCut(price, n-i, memo)
        # Getting the max value
        if cost > maxProfit:
            maxProfit = cost

    memo[n] = maxProfit
    return memo[n]

if __name__ == '__main__':
    price = [1, 5, 8, 9, 10, 17, 17, 20]
    n = 7 # Rod length
    print('Maximum profit is', rodCut(price, n))
```

Output

```
Maximum profit is 18
```

Second solution: Dynamic programming using tabulation approach
Time complexity: $O(n^2)$ because of two for loops
Space complexity: $O(n)$

```
def rodCut(price, n):
    # 'dp[i]' stores the maximum profit achieved from a rod of
        length 'i'
    dp = [0]*(n+1)

    # consider a rod of length 'i'
    for i in range(1, n+1):
        # Divide the rod of length 'i' into two rods of length 'j'
            and 'i-j' each and take maximum
        for j in range(1, i+1):
            dp[i] = max(dp[i], price[j-1] + dp[i-j])

    # 'dp[n]' stores the maximum profit achieved from a rod of
        length 'n'
    return dp[n]


if __name__ == '__main__':
    price = [1, 5, 8, 9, 10, 17, 17, 20]
    n = 7 # Rod length
    print('Maximum profit is', rodCut(price, n))
```

6.24 Numerical solution of an equation

Find a solution of the equation $f(x) = x^3 - 3x^2 - 7 = 0$ using numerical analysis

A numerical solution is an approximation to the solution of a mathematical equation, often used where analytical solutions are hard or impossible to find. There are various numerical methods to solve an equation numerically.

6.24.1 Bisection method

1. Choose two values a and b such that $a < b$ and $f(a)f(b) < 0$.
2. Find the midpoint $t = \dfrac{a+b}{2}$.
3. If $f(t) = 0$, then t is a solution.
4. If $f(t) \neq 0$, then
   (a) If $f(a)$ and $f(t)$ are of same sign, replace $a \leftarrow t$.
   (b) Else, replace $b \leftarrow t$.
5. Go back to step 2.
6. Iteration stops when $f(t) < \epsilon$ where ϵ is the specified tolerance and the solution is then given by the value of t.

This method always converges.

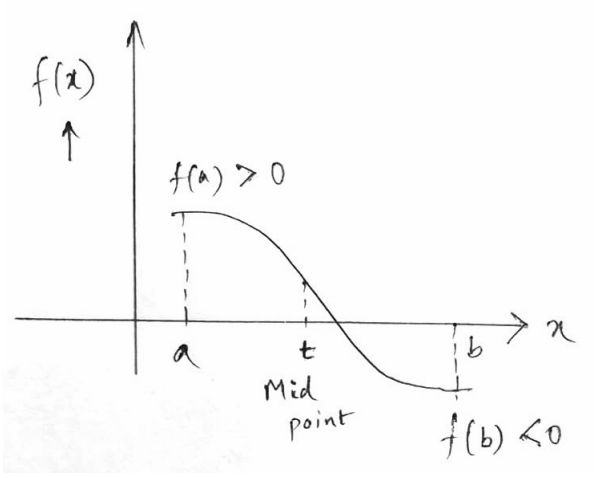


Figure 6.4: Bisection method

```
'''Bisection method'''
def f(x):
    # The equation to be solved is written in form f(x) = 0
    return x**3-3*x**2-7

def bisection_method(tolerance):
    a = int(input('Type the value of initial guess a = '))
    b = int(input('Type the value of initial guess b = '))

    while f(a)*f(b) > 0:
        print('f(a) = ', f(a))
        print('f(b) = ', f(b))
        print('Solution cannot be found. Enter new values of a and
            b.')
        a = int(input('Type the value of initial guess a = '))
        b = int(input('Type the value of initial guess b = '))
    print('f(a) = ', f(a))
    print('f(b) = ', f(b))
    if f(a)*f(b) < 0:
        t = (a + b)/2
        error = abs(f(t))
        iteration_counter = 1
        print('Iteration ',iteration_counter,': Value = ',t, ':
            Error = ', error)
        while error > tolerance:
            iteration_counter += 1
            if f(a)*f(t)>0:
                a = t
            else:
                b = t
            t = (a + b)/2
            error = abs(f(t))
            print('Iteration ',iteration_counter,': Value = ',t, ':
                Error = ', error)
    return t


if __name__ == "__main__":
    tolerance = 0.0001
    bisection_method(tolerance)
```

Output

```
Type the value of initial guess a = 2
Type the value of initial guess b = 6
f(a) = -11
f(b) = 101
Iteration 1 : Value = 4.0 : Error = 9.0
Iteration 2 : Value = 3.0 : Error = 7.0
```

```
Iteration 3 : Value = 3.5 : Error = 0.875
Iteration 4 : Value = 3.75 : Error = 3.546875
Iteration 5 : Value = 3.625 : Error = 1.212890625
Iteration 6 : Value = 3.5625 : Error = 0.138916015625
Iteration 7 : Value = 3.53125 : Error = 0.375457763671875
Iteration 8 : Value = 3.546875 : Error = 0.12013626098632812
Iteration 9 : Value = 3.5546875 : Error = 0.008922100067138672
Iteration 10 : Value = 3.55078125 : Error = 0.05572384595870972
Iteration 11 : Value = 3.552734375 : Error = 0.02343008667230606
Iteration 12 : Value = 3.5537109375 : Error = 0.007261299528181553
Iteration 13 : Value = 3.55419921875 : Error = 0.0008285733638331294
Iteration 14 : Value = 3.553955078125 : Error =
    0.0032168197649298236
Iteration 15 : Value = 3.5540771484375 : Error =
    0.0011942373766942183
Iteration 16 : Value = 3.55413818359375 : Error =
    0.00018286055114913324
Iteration 17 : Value = 3.554168701171875 : Error =
    0.00032284927007708575
Iteration 18 : Value = 3.5541534423828125 : Error =
    6.999257541195902e-05
```

6.24.2 Secant method

1. Choose two initial guesses x_0 and x_1.
2. For $n = 1, 2, 3, \cdots$
$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$
3. Iteration stops for $n = k$ when $f(x_k) < \epsilon$ where ϵ is the specified tolerance and the solution is then given by the value of x_k.

N.B. This method does not always converge.

Geometric interpretation: Let x_0 and x_1 be the initial approximations for the root s of $f(x) = 0$. $f(x_0)$ and $f(x_1)$ are their function values. Let x_2 be the point of intersection of the line joining $(x_0, f(x_0))$ and $(x_1, f(x_1))$.

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{0 - f(x_1)}{x_2 - x_1}$$

since $f(x_2) = 0$.

$$\Rightarrow x_2 = x_1 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_1)$$

General rule:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$

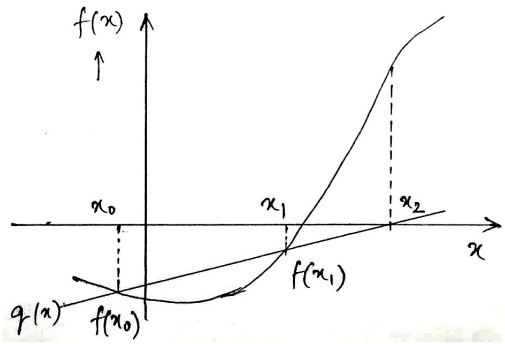


Figure 6.5: Secant method: geometric interpretation

```
'''Secant method'''
import sys

def f(x):
    # The equation to be solved is written in form f(x) = 0
    return x**3-3*x**2-7

def secant_method(tolerance):
    x0 = int(input('Type the value of the first initial guess x0 =
        '))
    x1 = int(input('Type the value of the second initial guess x1 =
        '))
    iteration_counter = 0
    while abs(f(x1)) > tolerance and iteration_counter < 100:
        try:
            denominator = float(f(x1) - f(x0))/(x1 - x0)
            x = x1 - float(f(x1))/denominator
        except ZeroDivisionError:
            print("Error! - denominator zero for x = ", x)
            sys.exit(1)    # Abort with error
        x0 = x1
        x1 = x
        iteration_counter += 1
        error = abs(f(x1))
        print('Iteration ', iteration_counter,': Value = ',x1, ':
            Error = ', error)
    return x1


if __name__ == "__main__":
    tolerance = 0.0001
    secant_method(tolerance)
```

Output

```
Type the value of the first initial guess x0 = 1
Type the value of the second initial guess x1 = 4
Iteration 1 : Value = 2.5 : Error = 10.125
Iteration 2 : Value = 3.2941176470588234 : Error = 3.808467331569311
Iteration 3 : Value = 3.7729200652528556 : Error =
    4.0024597927268815
Iteration 4 : Value = 3.527573079295066 : Error = 0.435000977724485
Iteration 5 : Value = 3.5516242617830494 : Error =
    0.04179231189964128
Iteration 6 : Value = 3.554180549595233 : Error =
    0.0005191945178779633
Iteration 7 : Value = 3.5541491819902817 : Error =
    6.06673715708439e-07
```

6.24.3 Newton-Raphson method

1. Choose an initial guess x_0.
2. For $n = 0, 1, 2, \cdots$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
provided $f'(x)$, the derivative of $f(x)$, exists at $x = x_n$.
3. Iteration stops for $n = k$ when $f(x_k) < \epsilon$ where ϵ is the specified tolerance and the solution is then given by the value of x_k.

N.B. This method does not always converge.

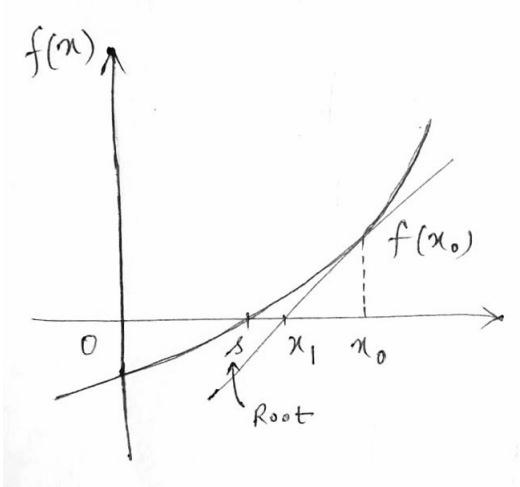


Figure 6.6: Newton-Raphson method: geometric interpretation

Geometric interpretation: Let x_0 be the initial guess for the root s of $f(x) = 0$. The tangent line to $y = f(x)$ at point $(x_0, f(x_0)$ has the following equation:

$$f'(x_0) = \frac{y - f(x_0)}{x - x_0}$$

If x_1 is the x-intercept of the tangent lin, then

$$f'(x_0) = -\frac{f(x_0)}{x_1 - x_0}$$
$$x_1 = x_0 - \frac{f(x_0)}{x_1 - x_0}$$

x_1 is the new Newton-Raphson estimate of the root. Next tangent line at $(x_1, f(x_1))$ will cut the x-axis much closer to the root.

$$\Rightarrow x_2 = x_1 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_1)$$

General rule:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$

```
'''Newton-Raphson method'''
def f(x):
# The equation to be solved is written in form f(x) = 0
    return x**3-3*x**2-7

def df_dx(x):
    # The derivative of f(x) w.r.t x
    return 3*x**2-6*x

def newton_raphson(tolerance):
    x0 = int(input('Type the value of the first initial guess x0 =
        '))
    iteration_counter = 0
    error = abs(f(x0))
    if error <= tolerance:
        return x0
    while error >= tolerance and iteration_counter <= 100:
        x1 = x0 - (f(x0)/df_dx(x0))
        x0 = x1
        error = abs(f(x0))
        iteration_counter += 1
        print('Iteration ', iteration_counter,': Value = ', x1, ':
            Error = ', error)
    return x1


if __name__ == "__main__":
```

```
tolerance = 0.0001
newton_raphson(tolerance)
```

Output

```
Type the value of the first initial guess x0 = 3
Iteration 1 : Value = 3.7777777777777777 : Error = 4.100137174211248
Iteration 2 : Value = 3.574278322440087 : Error = 0.3366729108147055
Iteration 3 : Value = 3.5543341033367417 : Error =
    0.003063993300784773
Iteration 4 : Value = 3.5541492344047616 : Error =
    2.618884451521808e-07
```

6.24.4 Regula-falsi method

1. Choose two values a and b such that $a < b$ and $f(a)f(b) < 0$.
2. Find the midpoint $t = \dfrac{af(b) - bf(a)}{f(b) - f(a)}$.
3. If $f(t) = 0$, then t is a solution.
4. If $f(t) \neq 0$, then
   (a) If $f(a)$ and $f(t)$ are of same sign, replace $a \leftarrow t$.
   (b) Else, replace $b \leftarrow t$.
5. Go back to step 2.
6. Iteration stops when $f(t) < \epsilon$ where ϵ is the specified tolerance and the solution is then given by the value of t.

Geometric interpretation: The equation of a straight line passing through the points $(a, f(a))$ and $(b, f(b))$ is

$$\frac{y - f(a)}{x - a} = \frac{f(b) - f(a)}{b - a}$$

At the intersection point with x-axis,

$$\begin{aligned} \frac{-f(a)}{x - a} &= \frac{f(b) - f(a)}{b - a} \\ \Rightarrow t\,(f(b) - f(a)) &= af(b) - af(a) - bf(a) + af(a) \\ \Rightarrow t &= \frac{af(b) - bf(a)}{f(b) - f(a)} \end{aligned}$$

Now if $f(t)f(a) > 0$, a is replaced by t.

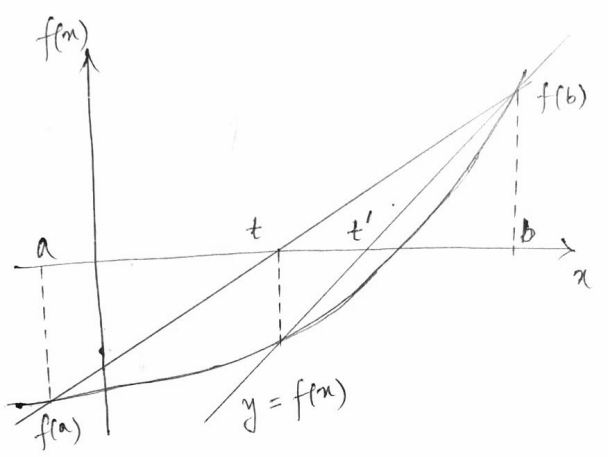


Figure 6.7: Regula-falsi method: geometric interpretation

```
'''Regula-falsi method'''
def f(x):
    # The equation to be solved is written in form f(x) = 0
    return x**3-3*x**2-7

def regulafalsi_method(tolerance):
    a = int(input('Type the value of initial guess a = '))
    b = int(input('Type the value of initial guess b = '))

    while f(a)*f(b) > 0:
        print('f(a) = ', f(a))
        print('f(b) = ', f(b))
        print('Solution cannot be found. Enter new values of a and
            b.')
        a = int(input('Type the value of initial guess a = '))
        b = int(input('Type the value of initial guess b = '))
    print('f(a) = ', f(a))
    print('f(b) = ', f(b))
    if f(a)*f(b) < 0:
        t = (a*f(b) - b*f(a))/(f(b) - f(a))
        error = abs(f(t))
        iteration_counter = 1
        print('Iteration ',iteration_counter,': Value = ',t, ':
            Error = ', error)
        while error > tolerance:
            iteration_counter += 1
            if f(a)*f(t)>0:
                a = t
            else:
                b = t
```

```
            t = (a*f(b) - b*f(a))/(f(b) - f(a))
            error = abs(f(t))
            print('Iteration ',iteration_counter,': Value = ',t, ':
                Error = ', error)
    return t


if __name__ == "__main__":
    tolerance = 0.0001
    regulafalsi_method(tolerance)
```

Output

```
Type the value of initial guess a = 1
Type the value of initial guess b = 4
f(a) = -9
f(b) = 9
Iteration 1 : Value = 2.5 : Error = 10.125
Iteration 2 : Value = 3.2941176470588234 : Error = 3.808467331569311
Iteration 3 : Value = 3.504004576659039 : Error = 0.8118075786435668
Iteration 4 : Value = 3.5450421571877415 : Error =
    0.1502786735172137
Iteration 5 : Value = 3.5525141111646144 : Error =
    0.02707494028368984
Iteration 6 : Value = 3.553856257297017 : Error =
    0.0048540143135582525
Iteration 7 : Value = 3.5540967484931585 : Error =
    0.0008694628781782399
Iteration 8 : Value = 3.5541398217013684 : Error =
    0.00015571566942895743
Iteration 9 : Value = 3.5541475357252508 : Error =
    2.7886966961432336e-05
```

6.25 Edit distance (Levenshtein distance)

Given two strings `str1` and `str2`, find the minimum number of edits (operations which include *insert*, *remove* and *replace*) required to convert `str1` into `str2`.

The **Levenshtein distance** between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

Example 1

```
Input:  str1 = "geek", str2 = "gesek"
Output: 1
We can convert str1 into str2 by inserting a 's'.
```

Example 2

```
Input:  str1 = "cat", str2 = "cut"
Output: 1
We can convert str1 into str2 by replacing 'a' with 'u'.
```

Example 3

```
Input:  str1 = "sunday", str2 = "saturday"
Output: 3
Last three and first characters are same. We basically
need to convert "un" to "atur". This can be done using
below three operations.
Replace 'n' with 'r', insert 't', insert 'a'
```

First solution: Naive recursive approach

1. Let m = the length of `str1` and n = the length of `str2`.
2. If last characters of two strings are same, ignore last characters and get count for remaining strings. So we recur for lengths $m-1$ and $n-1$.
3. Else (if last characters are not same), we consider all three operations on last character of first string `str1`, recursively compute minimum cost for all three operations and take minimum of three values.
   (a) Insert: Recur for lengths m and $n-1$.
   (b) Remove: Recur for lengths $m-1$ and n.
   (c) Replace: Recur for lengths $m-1$ and $n-1$.

Time complexity: $O(3^{\min(m,n)})$
Space complexity: $O(\min(m,n))$ for the requirement of call stack

```
def editDistance(str1, str2, m, n):
    # m = length of str1
    # n = length of str2
    # If first string is empty, the only option is to insert all
        characters of second string into first
    if m == 0:
        return n

    # If second string is empty, the only option is to remove all
        characters of first string
    if n == 0:
        return m

    # If last characters of two strings are same, ignore last
        characters and get count for remaining strings.
    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)

    # If last characters are not same, consider all three
        operations on last character of first string, recursively
    # Compute minimum cost for all three operations and take
        minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1), # Insert
                   editDistance(str1, str2, m-1, n), # Remove
                   editDistance(str1, str2, m-1, n-1) # Replace
                   )


if __name__ == "__main__":
    str1 = "sunday"
    str2 = "saturday"
    print (editDistance(str1, str2, len(str1), len(str2)))
```

Output

```
3
```

Second solution: Memoization (Top-down) approach

Time complexity: $O(mn)$
Space complexity: $O(mn)$

```
def editDistance(str1, str2, m, n, memo = {}):
    # m = length of str1
    # n = length of str2
    # If first string is empty, the only option is to insert all
        characters of second string into first
```

```
    if m == 0:
        return n

    # If second string is empty, the only option is to remove all
        characters of first string
    if n == 0:
        return m

    # Check if the recursive tree for given n & m has already been
        executed
    key = f'{m}_{n}' # An unique key for the set of values m and n
    if memo.get(key) != None:
        return memo[key]

    # If last characters of two strings are same, ignore last
        characters and get count for remaining strings.
    if str1[m-1] == str2[n-1]:
        memo[key] = editDistance(str1, str2, m-1, n-1, memo)
        return memo[key]

    # If last characters are not same, consider all three
        operations on last character of first string, recursively
    # Compute minimum cost for all three operations and take
        minimum of three values.
    memo[key] = 1 + min(editDistance(str1, str2, m, n-1, memo), #
        Insert
                editDistance(str1, str2, m-1, n, memo), # Remove
                editDistance(str1, str2, m-1, n-1, memo) # Replace
                )
    return memo[key]


if __name__ == "__main__":
    str1 = "sunday"
    str2 = "saturday"
    print (editDistance(str1, str2, len(str1), len(str2)))
```

Output

```
3
```

Second solution: Tabulation (Bottom-up) approach
We will define a table of size $(m+1) \times (n+1)$, where $dp[i][j]$ is the content of the (i, j)th cell, which indicates the edit distance of the substrings from `str1` and `str2` of length i and j respectively. Recursive equation for $dp[i][j]$:

- If $\min(i, j) = 0$:

$$dp[i][j] = \max(i, j)$$

- If ith character of str1 $=$ jth character of str2:

$$dp[i][j] = dp[i-1][j-1]$$

- If ith character of str1 $\neq$ jth character of str2:

$$dp[i][j] = 1 + \min\left(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]\right)$$

Time complexity: $O(mn)$
Space complexity: $O(mn)$

```
def editDistance(str1, str2, m, n):
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    # Fill dp[][] in bottom up manner
    for i in range(m+1):
        for j in range(n+1):

            # If first string is empty, only option is to
            # insert all characters of second string
            if i == 0:
                dp[i][j] = j  # Min. operations = j

            # If second string is empty, only option is to
            # remove all characters of second string
            elif j == 0:
                dp[i][j] = i  # Min. operations = i

            # If last characters are same, ignore last char
            # and recur for remaining string
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]

            # If last character are different, consider all
            # possibilities and find minimum
            else:
                dp[i][j] = 1 + min(dp[i][j-1],  # Insert
                                   dp[i-1][j],  # Remove
                                   dp[i-1][j-1])  # Replace

    return dp[m][n]


if __name__ == "__main__":
    str1 = "sunday"
    str2 = "saturday"
    print (editDistance(str1, str2, len(str1), len(str2)))
```

Output

```
3
```