Design and Analysis of Algorithms

Lab Sheet 2

Implement the following sorting algorithms and answer the associated questions.

1. Bubble sort

a. What is the time complexity of a simple bubble sort algorithm? Is there any difference between the best case and the worst case?

b. How can we change the best-case complexity to Ω(n)? Modify your algorithm accordingly. What is the worst-case complexity of the improved algorithm?

c. Give examples for best-case and worst-case inputs.

Solution:

Code:

import numpy as np

def bubbleSort(arr):

    n = len(arr)

    swapped = False

    for i in range(n-1):

        for j in range(0, n-i-1):

            if arr[j] > arr[j + 1]:

                swapped = True

                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:

            return

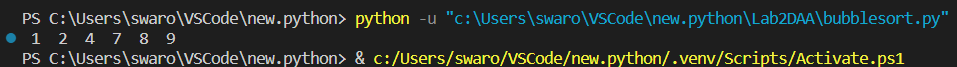
arr = [2,7,1,9,4,8]

bubbleSort(arr)

for i in range(len(arr)):

    print("% d" % arr[i], end=" ")

Output:



a. The time complexity of a simple bubble sort algorithm is O(n^2) in the worst case and the average case. This means the algorithm's performance degrades as the input size (n) increases. In the best case, when the input is already sorted, bubble sort has a time complexity of O(n), which is more efficient. So, there is a significant difference between the best-case and worst-case time complexities.

b. To change the best-case complexity to Ω(n) (making it more efficient in the best-case scenario), you can add a flag to track whether any swaps were made during a pass through the array. If no swaps were made during a pass, the array is already sorted, and you can terminate the algorithm early. Here's a modified bubble sort algorithm with this improvement:

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n - i - 1):

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

swapped = True

if not swapped:

break

With this modification, the best-case time complexity is Ω(n) because if the array is already sorted, we only need to make one pass to confirm that no swaps are needed, making the algorithm terminate early. The worst-case time complexity is still O(n^2) because in the worst case, we would still need to make n passes through the array.

c. Examples for best-case and worst-case inputs:

Best-case input: An array that is already sorted in ascending order. In this case, the algorithm will only make one pass through the array to confirm that no swaps are needed, resulting in the best-case time complexity of O(n).

Worst-case input: An array sorted in descending order. In this case, the algorithm will require the maximum number of passes and swaps to sort the array, resulting in the worst-case time complexity of O(n^2).

2. Selection sort

a. What is the time complexity of a simple selection sort algorithm? Is there any difference between the best case and the worst case?

b. How can we change the best-case complexity to Ω(n)? Modify your algorithm accordingly. What is the worst-case complexity of the improved algorithm?

c. Give examples for best-case and worst-case inputs.

Solution:

Code:

def selectionSort(array, size):

    for ind in range(size):

        min\_index = ind

        for j in range(ind + 1, size):

            if array[j] < array[min\_index]:

                min\_index = j

        (array[ind], array[min\_index]) = (array[min\_index], array[ind])

arr = [9,1,5,3,2,8,7,6]

size = len(arr)

selectionSort(arr, size)

print(arr)

Output:



a. The time complexity of a simple selection sort algorithm is O(n^2) in both the best case and worst case. This means that the performance of selection sort remains the same regardless of whether the input is already sorted or in reverse order, so there is no difference between the best case and worst case time complexities.

b. To change the best-case complexity to Ω(n) (making it more efficient in the best-case scenario), you can use a modified selection sort algorithm that checks if the current element is the minimum element in the remaining unsorted part of the array before swapping. This way, you can avoid unnecessary swaps when the current element is already the minimum. Here's a modified selection sort algorithm with this improvement:

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_idx = i

for j in range(i + 1, n):

if arr[j] < arr[min\_idx]:

min\_idx = j

if min\_idx != i:

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

With this modification, the best-case time complexity is Ω(n) because, in the best case, we only need to make one comparison for each element, and no swaps are needed. The worst-case time complexity remains O(n^2).

c. Examples for best-case and worst-case inputs:

Best-case input: An array that is already sorted in ascending order. In this case, the modified selection sort algorithm will still make the same number of comparisons but will perform fewer swaps, resulting in the best-case time complexity of Ω(n).

Worst-case input: An array sorted in descending order. In this case, the modified selection sort algorithm will make the same number of comparisons and swaps as the simple selection sort, resulting in the worst-case time complexity of O(n^2).

3. Insertion sort

a. What is the time complexity of a simple selection sort algorithm? Is there any difference between the best case and the worst case?

b. How can we change the best-case complexity to Ω(n)? Modify your algorithm accordingly. What is the worst-case complexity of the improved algorithm?

c. Give examples for best-case and worst-case inputs.

Solution:

Code:

def insertionSort(arr):

    n = len(arr)

    if n <= 1:

        return

    for i in range(1, n):

        key = arr[i]

        j = i-1

        while j >= 0 and key < arr[j]:

            arr[j+1] = arr[j]

            j -= 1

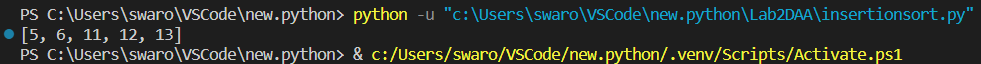
        arr[j+1] = key

arr = [12, 11, 13, 5, 6]

insertionSort(arr)

print(arr)

Output:



a. The time complexity of a simple insertion sort algorithm is O(n^2) in the worst case and average case. This means that the performance of insertion sort degrades as the input size (n) increases. In the best case, when the input is already sorted, insertion sort has a time complexity of O(n), which is more efficient. So, there is a significant difference between the best case and worst case time complexities.

b. To change the best-case complexity to Ω(n) (making it more efficient in the best-case scenario), you can use a modified insertion sort algorithm that checks whether an element is in its correct position without making unnecessary comparisons and swaps. Here's a modified insertion sort algorithm with this improvement:

def insertion\_sort(arr):

n = len(arr)

for i in range(1, n):

key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

With this modification, the best-case time complexity is Ω(n) because, in the best case, when the input is already sorted, we only need to make one comparison for each element, and no swaps are needed. The worst-case time complexity remains O(n^2).

c. Examples for best-case and worst-case inputs:

Best-case input: An array that is already sorted in ascending order. In this case, the modified insertion sort algorithm will make one comparison for each element, and no swaps are needed, resulting in the best-case time complexity of Ω(n).

Worst-case input: An array sorted in descending order. In this case, the modified insertion sort algorithm will require the maximum number of comparisons and swaps to sort the array, resulting in the worst-case time complexity of O(n^2).

4. Merge sort

Solution:

Code:

def mergeSort(array):

    if len(array) > 1:

        r = len(array)//2

        L = array[:r]

        M = array[r:]

        mergeSort(L)

        mergeSort(M)

        i = j = k = 0

        while i < len(L) and j < len(M):

            if L[i] < M[j]:

                array[k] = L[i]

                i += 1

            else:

                array[k] = M[j]

                j += 1

            k += 1

        while i < len(L):

            array[k] = L[i]

            i += 1

            k += 1

        while j < len(M):

            array[k] = M[j]

            j += 1

            k += 1

def printList(array):

    for i in range(len(array)):

        print(array[i], end=" ")

    print()

if \_\_name\_\_ == '\_\_main\_\_':

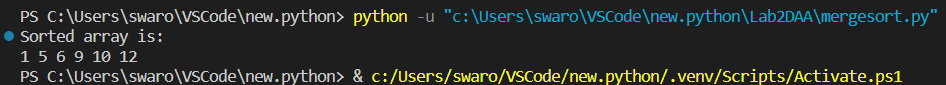
    array = [6, 5, 12, 10, 9, 1]

    mergeSort(array)

    print("Sorted array is: ")

    printList(array)

Output:



5. Quick sort

Solution:

Code:

def partition(array, low, high):

    pivot = array[high]

    i = low - 1

    for j in range(low, high):

        if array[j] <= pivot:

            i = i + 1

            (array[i], array[j]) = (array[j], array[i])

    (array[i + 1], array[high]) = (array[high], array[i + 1])

    return i + 1

def quickSort(array, low, high):

    if low < high:

        pi = partition(array, low, high)

        quickSort(array, low, pi - 1)

        quickSort(array, pi + 1, high)

data = [1, 7, 4, 1, 10, 9, -2]

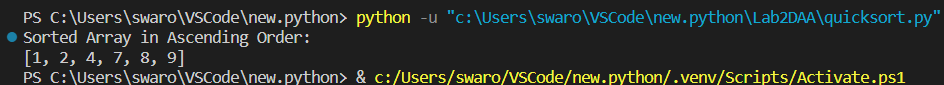
size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')

print(data)

Output:



6. Sort a set of strings using Radix sort

Solution:

Code:

def radix\_sort(strings):

max\_length = len(max(strings, key=len))

strings = [s.ljust(max\_length) for s in strings]

for i in range(max\_length - 1, -1, -1):

strings = counting\_sort(strings, i)

return [s.rstrip() for s in strings]

def counting\_sort(strings, position):

count = [0] \* 256

for s in strings:

count[ord(s[position])] += 1

for i in range(1, 256):

count[i] += count[i - 1]

output = [''] \* len(strings)

for i in range(len(strings) - 1, -1, -1):

index = count[ord(strings[i][position])] - 1

output[index] = strings[i]

count[ord(strings[i][position])] -= 1

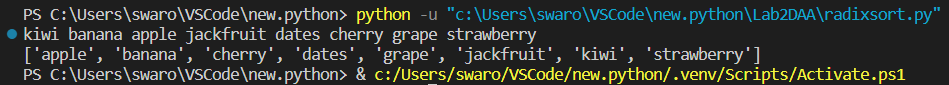
return output

input\_strings = input().split()

sorted\_strings = radix\_sort(input\_strings)

print(sorted\_strings)

Output:



Write Recursive algorithms for the following problems. Implement your algorithm, write the recurrence relation, solve it, and find the asymptotic time complexity

7. Print the sum of the first N natural numbers.

Solution:

def sum\_of\_natural\_numbers(N):

    if N == 0:

        return 0

    else:

        return N + sum\_of\_natural\_numbers(N - 1)

Recurrence relation: T(N) = T(N - 1) + O(1)

Solving the recurrence relation: T(N) = T(N - 1) + O(1) = T(N - 2) + O(1) + O(1) = T(N - 3) + O(1) + O(1) + O(1) = ... = T(1) + O(1) + O(1) + ... + O(1)

T(1) represents the base case, which is O(1).

So, T(N) = O(1) + O(1) + ... + O(1) (N times) = N \* O(1) = O(N)

The asymptotic time complexity of this algorithm is O(N). It's a linear time complexity since the number of recursive calls is directly proportional to N.

8. Print the product of the first N natural numbers

Solution:

def product\_of\_natural\_numbers(N):

if N == 0:

return 1

else:

return N \* product\_of\_natural\_numbers(N - 1)

Recurrence relation: T(N) = T(N - 1) + O(1)

Solving the recurrence relation: T(N) = T(N - 1) + O(1) = T(N - 2) + O(1) + O(1) = T(N - 3) + O(1) + O(1) + O(1) = ... = T(1) + O(1) + O(1) + ... + O(1)

T(1) represents the base case, which is O(1).

So, T(N) = O(1) + O(1) + ... + O(1) (N times) = N \* O(1) = O(N)

The asymptotic time complexity of this algorithm is O(N). It's a linear time complexity since the number of recursive calls is directly proportional to N.

9. Print the Nth Fibonacci number.

Solution:

def fibonacci(N):

if N == 0:

return 0

elif N == 1:

return 1

else:

return fibonacci(N - 1) + fibonacci(N - 2)

Recurrence relation: T(N) = T(N - 1) + T(N - 2) + O(1)

Solving the recurrence relation can be a bit complex, but the time complexity can be determined more intuitively:

Each call to the fibonacci function results in two additional calls (T(N - 1) and T(N - 2)), and this branching continues until the base cases are reached. The number of function calls in the worst case grows exponentially with N, which leads to an exponential time complexity.

Therefore, the time complexity of this naive recursive algorithm is exponential, approximately O(2^N), which can be quite slow for large values of N.

To optimize the computation of Fibonacci numbers and achieve a more efficient time complexity, you can use memoization (storing previously computed Fibonacci numbers to avoid redundant calculations) or an iterative approach. These approaches reduce the time complexity to O(N) or even O(log(N)) using matrix exponentiation.

10. Calculate xy.

Solution:

def power(x, y):

if y == 0:

return 1

elif y > 0:

return x \* power(x, y - 1)

Recurrence relation: T(y) = T(y - 1) + O(1)

Solving the recurrence relation: T(y) = T(y - 1) + O(1) = T(y - 2) + O(1) + O(1) = T(y - 3) + O(1) + O(1) + O(1) = ... = T(0) + O(1) + O(1) + ... + O(1)

T(0) represents the base case, which is O(1).

So, T(y) = O(1) + O(1) + ... + O(1) (y times) = y \* O(1) = O(y)

The asymptotic time complexity of this algorithm is O(y), where y is the exponent.

However, it's important to note that this algorithm is not very efficient for large values of y as it has exponential time complexity. You can optimize this by using a more efficient algorithm, such as the binary exponentiation method, which has a time complexity of O(log(y)).

11. Print the first N natural numbers

Solution:

def print\_natural\_numbers(N):

if N == 0:

return

else:

print(N)

print\_natural\_numbers(N - 1)

Recurrence relation: T(N) = T(N - 1) + O(1)

Solving the recurrence relation: T(N) = T(N - 1) + O(1) = T(N - 2) + O(1) + O(1) = T(N - 3) + O(1) + O(1) + O(1) = ... = T(1) + O(1) + O(1) + ... + O(1)

T(1) represents the base case, which is O(1).

So, T(N) = O(1) + O(1) + ... + O(1) (N times) = N \* O(1) = O(N)

The asymptotic time complexity of this algorithm is O(N). However, it's important to note that using recursion for this simple task is not the most efficient approach, and a loop would be more straightforward and efficient. Recursive algorithms are better suited for problems with more complex recursive structures.

12. Print the first N natural numbers in reverse order

Solution:

def print\_natural\_numbers\_reverse(N):

if N == 0:

return

else:

print(N)

print\_natural\_numbers\_reverse(N - 1)

Recurrence relation: T(N) = T(N - 1) + O(1)

Solving the recurrence relation: T(N) = T(N - 1) + O(1) = T(N - 2) + O(1) + O(1) = T(N - 3) + O(1) + O(1) + O(1) = ... = T(1) + O(1) + O(1) + ... + O(1)

T(1) represents the base case, which is O(1).

So, T(N) = O(1) + O(1) + ... + O(1) (N times) = N \* O(1) = O(N)

The asymptotic time complexity of this algorithm is O(N). However, just like with the previous problem, using recursion for this task is not the most efficient approach, and a loop would be more straightforward and efficient. Recursive algorithms are better suited for problems with more complex recursive structures.

13. Find the GCD(HCF) of two numbers.

Solution:

def gcd(a, b):

if b == 0:

return a

else:

return gcd(b, a % b)  
  
Recurrence relation:

T(a,b) = T(b,a mod b)+O(1)

Solving the recurrence relation: The Euclidean Algorithm performs O(log(min(a,b))) recursive calls in the worst case. The worst-case occurs when the two numbers are consecutive Fibonacci numbers, which minimizes the number of recursive calls.

The asymptotic time complexity is O(log(min(a,b))).

This algorithm is efficient for finding the GCD/HCF and is widely used due to its simplicity and effectiveness. It exploits the fact that the GCD of two numbers is the same as the GCD of the smaller number and the remainder of the division of the larger number by the smaller one.

14. Print the elements of an array.

Solution:

def print\_array(arr, index):

if index == len(arr):

return

else:

print(arr[index])

print\_array(arr, index + 1)

Recurrence relation: T(n)=T(n−1)+O(1)

Solving the recurrence relation: T(n) = T(n−1)+O(1) = T(n−2)+O(1)+O(1) = T(n−3)+O(1)+O(1)+O(1) = T(1)+O(1)+O(1)+…+O(1)

In each recursive call, we perform constant-time operations (O(1)). The number of recursive calls is n, so the total time complexity is O(n).

The asymptotic time complexity of this algorithm is O(n), where n is the number of elements in the array. It's important to note that using recursion for printing elements is less efficient than using a loop, as recursion involves additional function call overhead. Recursion is more suitable for problems with more complex recursive structures.

15. Print the elements of an array in reverse order

Solution:

def print\_array\_reverse(arr, index):

if index < 0:

return

else:

print(arr[index])

print\_array\_reverse(arr, index - 1)

Recurrence relation: T(n)=T(n−1)+O(1)

Solving the recurrence relation: T(n) = T(n−1)+O(1) = T(n−2)+O(1)+O(1) = T(n−3)+O(1)+O(1)+O(1) … = T(0)+O(1)+O(1)+…+O(1)

In each recursive call, we perform constant-time operations O(1)). The number of recursive calls is n, so the total time complexity is O(n).

The asymptotic time complexity of this algorithm is O(n), where n is the number of elements in the array. Just like the previous case, using recursion for this task is not the most efficient approach, and a loop would be more straightforward and efficient. Recursive algorithms are better suited for problems with more complex recursive structures.

16. Reverse a given number

Solution:

def reverse\_number(num):

if num >= 0 and num <= 9:

return num

else:

last\_digit = num % 10

remaining\_digits = num // 10

reversed\_remaining = reverse\_number(remaining\_digits)

reversed\_num = int(str(last\_digit) + str(reversed\_remaining))

return reversed\_num

Recurrence relation: T(n)=T(n/10)+O(1)

Solving the recurrence relation: The algorithm divides the number by 10 in each recursive call, so the number of recursive calls is logarithmic (base 10) to the input number. The recurrence relation becomes: T(n) = T(n/10)+O(1) = T(n/100)+O(1)+O(1) = … =T(1)+O(1)+O(1)+ … +O(1)

In each recursive call, we perform constant-time operations O(1)). The number of recursive calls is logarithmic to the input number, so the total time complexity is O(log 10​n).

The asymptotic time complexity of this algorithm is O(log 10​n). Note that the actual base of the logarithm doesn't matter for big O notation.

17. Check if an array is sorted or not

Solution:

def is\_sorted(arr, n):

if n <= 1:

return True

else:

return (arr[n - 1] >= arr[n - 2]) and is\_sorted(arr, n - 1)

Recurrence relation: T(n)=T(n−1)+O(1)

Solving the recurrence relation: T(n) = T(n−1)+O(1) = T(n−2)+O(1)+O(1) = T(n−3)+O(1)+O(1)+O(1) … = T(1)+O(1)+O(1)+…+O(1)

In each recursive call, we perform constant-time operations O(1)). The number of recursive calls is n, so the total time complexity is O(n).

The asymptotic time complexity of this algorithm is O(n). However, it's important to note that using recursion for this task is not the most efficient approach, and a loop would be more straightforward and efficient. Recursive algorithms are better suited for problems with more complex recursive structures.

18. Write a recursive algorithm to find the median of median in O(n) time

Solution:

def find\_median\_of\_medians(arr):

n = len(arr)

groups = [arr[i:i + 5] for i in range(0, n, 5)]

sorted\_groups = [sorted(group) for group in groups]

medians = [group[len(group) // 2] for group in sorted\_groups]

if len(medians) <= 5:

return sorted(medians)[len(medians) // 2]

else:

return find\_median\_of\_medians(medians)

arr = [3, 2, 1, 5, 4, 9, 8, 7, 6, 10]

result = find\_median\_of\_medians(arr)

print("Median of medians:", result)

Recurrence relation: T(n)=T(n/5​)+T(7n/10​)+O(n)

Solving the recurrence relation can be complex, but it can be shown that the time complexity of the median of medians algorithm is linear (O(n)).

The asymptotic time complexity of the median of medians algorithm is O(n), making it efficient for finding the median in linear time. However, the algorithm has a higher constant factor compared to simpler linear-time algorithms, and in practice, randomized algorithms like QuickSelect are often preferred.

19. Write a recursive algorithm to find the kth largest element

Solution:

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] >= pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

def quick\_select(arr, low, high, k):

if low <= high:

pivot\_index = partition(arr, low, high)

if pivot\_index == k:

return arr[pivot\_index]

elif pivot\_index > k:

return quick\_select(arr, low, pivot\_index - 1, k)

else:

return quick\_select(arr, pivot\_index + 1, high, k - pivot\_index - 1)

arr = [3, 2, 1, 5, 4, 9, 8, 7, 6, 10]

k = 3

result = quick\_select(arr, 0, len(arr) - 1, len(arr) - k)

print(f"{k}th largest element:", result)

Recurrence relation: T(n)=T(n/2)+O(n)

Solving the recurrence relation can be complex, but it can be shown that the time complexity of the QuickSelect algorithm is linear on average, making it efficient for finding the kth largest element in linear time.

The asymptotic time complexity of the QuickSelect algorithm is O(n), on average, where n is the size of the array. The worst-case time complexity is O(n2), but the worst case is rare in practice due to the random choice of pivots in most implementations.