**SSN College of Engineering**

Department of Information Technology

UIT2201 — Programming and Data Structures

2022 – 2023

**Exercise — 06**

U. Pranaav | IT-B | 3122225002093

I. AIM:

To arrange n elements in ascending or order using merge sort and analyze the time complexity of the code and express the same in asymptotic notation. Also find the performance of the sorting algorithm in the average case, best case and worst case scenarios.

II. CODE:

# -\*- coding: utf-8 -\*-

"""

This module provides two functions that are used for implementing

merge sort. One of the functions is used for merging two sorted

lists and giving a third sorted list that combines the two lists

as output. Other function performs the merge sort by recursively

calling the first function and returns a sorted list. This is a

part of the exercises given under the course UIT2201 (Programming

and Data Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 10 2023

Revised on Wed May 10 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

"""

import random

import time

*def* merge(*A*,*B*):

    '''

    This function merges two sorted lists, A and B, into a single

    sorted list C. It compares the elements from A and B one by one,

    placing the smaller element in C. The function keeps track of the

    number of comparisons made using the global variable count.

    The input lists A and B are not modified, and there are no side

    effects.

    Args:

        A: First sorted list.

        B: Second sorted list.

    Returns:

        A new list C containing all elements from A and B, merged in

        sorted order.

    '''

    global count

    i = 0

    j = 0

    C = []

    while i < len(*A*) and j < len(*B*):

        count += 1

        if *A*[i] < *B*[j]:

            C.append(*A*[i])

            i += 1

        else:

            C.append(*B*[j])

            j += 1

    if i < len(*A*):

        C.extend(*A*[i:])

    else:

        C.extend(*B*[j:])

    return C

*def* merge\_sort(*data*):

    '''

    This function implements the merge sort algorithm to sort a

    given list of data in ascending order. It recursively divides

    the list into smaller sublists, sorts them individually, and

    then merges them back together using the merge() function.

    The input list is not modified, and there are no side effects.

    Args:

        data: The list of elements to be sorted.

    Returns:

        A new list containing the sorted elements of the input list.

    '''

    n = len(*data*)

    if n < 2:

        return *data*[:]

    else:

        mid = n // 2

        return merge(merge\_sort(*data*[:mid]), merge\_sort(*data*[mid:]))

*def* random\_list(*size*):

    '''

    The given function generates a random number of values

    and returns the values in a list.

    args:

        size: the number of point objects to be

        generated

    Returns:

        A list of random integer values.

    '''

    random\_list = []

    for case in range(*size*):

        x\_val = random.randint(-1000,1000)

        random\_list.append(x\_val)

    return random\_list

*def* worst\_scenario(*size*):

    '''

    The function provides the parameters required for

    printing a worst case scenario for sorting.

    The input is not modified and there are no side effects.

    args:

        size: the size of random list to be generated

    Returns:

        None

    '''

    random\_lst = [x for x in range(*size*,0,-1)]

    merge\_sort(random\_lst)

    print("Worst case scenario is:", count)

*def* best\_scenario(*size*):

    '''

    The function provides the parameters required for

    printing a best case scenario for sorting.

    The input is not modified and there are no side effects.

    args:

        size: the size of random list to be generated

    Returns:

        None

    '''

    random\_lst = [x for x in range(*size*)]

    merge\_sort(random\_lst)

    print("Best case scenario is:", count)

*def* avg\_scenario(*size*):

    '''

    The function provides the parameters required for

    printing an average case scenario for sorting.

    The input is not modified and there are no side effects.

    args:

        size: the size of random list to be generated

    Returns:

        None

    '''

    random\_lst = random\_list(*size*)

    merge\_sort(random\_lst)

    print("Average case scenario is:", count)

#driver code

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

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    count = 0

    random\_lst = random\_list(20)

    print(*f*"Random list is : {random\_lst}")

    sorted\_val = merge\_sort(random\_lst)

    print(*f*"Sorted list is: {sorted\_val}")

    print(*f*"Count is : {count}")

    k = 10

    while k <= 10000:

        count = 0

        start = time.time()

        merge\_sort(random\_list(k))

        end = time.time()

        print(*f*"k is {k} and count is : {count}")

        print(*f*"Time taken is : {end - start}")

        print()

        k \*= 10

    k = 20000

    count = 0

    start = time.time()

    merge\_sort(random\_list(k))

    end = time.time()

    print(*f*"k is {k} and count is : {count}")

    print(*f*"Time taken is : {end - start}")

    print()

    k = 50000

    count = 0

    start = time.time()

    merge\_sort(random\_list(k))

    end = time.time()

    print(*f*"k is {k} and count is : {count}")

    print(*f*"Time taken is : {end - start}")

    print()

    print("Test case size: 10000\n")

    count = 0

    avg\_scenario(10000)

    count = 0

    best\_scenario(10000)

    count = 0

    worst\_scenario(10000)

III. OUTPUT:

Random list is : [-868, -484, 886, -901, -490, 193, 22, -841, 695, -914, 932, -994, -975, 962, -562, 672, -352, 95, 12, 551]

Sorted list is: [-994, -975, -914, -901, -868, -841, -562, -490, -484, -352, 12, 22, 95, 193, 551, 672, 695, 886, 932, 962]

Count is : 66

k is 10 and count is : 21

Time taken is : 0.001001596450805664

k is 100 and count is : 549

Time taken is : 0.0

k is 1000 and count is : 8675

Time taken is : 0.005000591278076172

k is 10000 and count is : 120391

Time taken is : 0.06281590461730957

k is 20000 and count is : 260805

Time taken is : 0.13418030738830566

k is 50000 and count is : 718158

Time taken is : 0.3586909770965576

Test case size: 10000

Average case scenario is: 120431

Best case scenario is: 64608

Worst case scenario is: 69008

2. I. AIM:

To sort a list using a recursive binary search method and calculate the time taken to sort the given list. Also express the same in asymptotic notation.

II. CODE:

# -\*- coding: utf-8 -\*-

"""

This module provides a function that performs a binary search on

a sorted list in order to find a certain element, and displays

-1 if the given element is not found. This is a part of the

exercises given under the course UIT2201 (Programming and Data

Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 10 2023

Revised on Wed May 17 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

"""

import random

from time import time

*def* binary\_search(*data*,*s\_ele*,*high*,*low*=0):

    '''

    The binary\_search function performs a binary search on a sorted

    list of data to find a specified element.

    Args:

        data (list): A sorted list of elements to search in.

        s\_ele (int or float): The element to search for in the data.

        high (int): The index of the highest element in the data.

        low (int, optional): The index of the lowest element in the

        data. Default is 0.

    Assumptions:

        The input data list is sorted in ascending order.

        The input high value is valid and within the bounds

        of the data list.

        The input low value is valid and within the bounds

        of the data list.

    Returns:

        If the element s\_ele is found in the data, the

        function returns its index.

        If the element s\_ele is not found in the data, the

        function returns -1.

    '''

    if *low* <= *high*:

        mid = (*low* + *high*) // 2

        if *data*[mid] == *s\_ele*:

            return mid

        elif *data*[mid] > *s\_ele*:

            return binary\_search(*data*,*s\_ele*,mid-1,*low*)

        elif *data*[mid] < *s\_ele*:

            return binary\_search(*data*,*s\_ele*,*high*,mid+1)

    else:#1 2 3 4 5 6 7

        return -1

*def* random\_list(*size*):

    '''

    The given function generates a random number of values

    and returns the values in a list.

    args:

        size: the number of point objects to be

        generated

    Returns:

        A list of random integer values.

    '''

    random\_list = []

    for case in range(*size*):

        x\_val = random.randint(-1000,1000)

        random\_list.append(x\_val)

    return random\_list

#driver code

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

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    random\_lst = [x for x in range(1,10000001)]

    search\_val = random\_lst[0]

    start = time()

    ind = binary\_search(random\_lst,search\_val,len(random\_lst)-1)

    end = time()

    print(*f*"Actual index : 0 , index found via binary search : {ind}")

    print("Time taken is :", end - start)

    print()

    random\_lst = [x for x in range(1,10000001)]

    search\_val = random\_lst[-1]

    start = time()

    ind = binary\_search(random\_lst,search\_val,len(random\_lst)-1)

    end = time()

    print(*f*"Actual index : 9999999 , index found via binary search : {ind}")

    print("Time taken is :", end - start)

    print()

    random\_lst = [x for x in range(1,10000001)]

    search\_val = 100000000000000

    start = time()

    ind = binary\_search(random\_lst,search\_val,len(random\_lst)-1)

    end = time()

    print(*f*"Element is not present in list, index found via binary search : {ind}")

    print("Time taken is :", end - start)

    print()

III. OUTPUT:

Actual index : 0 , index found via binary search : 0

Time taken is : 0.0

Actual index : 9999999 , index found via binary search : 9999999

Time taken is : 0.0

Element is not present in list, index found via binary search : -1

Time taken is : 0.0

PART – B

3. I. AIM:

To arrange n elements in ascending or order using quick sort and analyze the time complexity of the code and express the same in asymptotic notation. Also find the performance of each sorting algorithm in their average case, best case and worst case scenarios.

II. CODE:

# -\*- coding: utf-8 -\*-

"""

This module provides two functions that are used for implementing

quick sort. One of the functions is used for merging two sorted

lists and giving a third sorted list that combines the two lists

as output. Other function performs the merge sort by recursively

calling the first function and returns a sorted list. This is a

part of the exercises given under the course UIT2201 (Programming

and Data Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed May 10 2023

Revised on Wed May 10 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

"""

import random

import time

*def* partition(*data*,*i*,*j*,*pivot*):

    '''

    This function partitions a given list 'data' based on a pivot

    element. It rearranges the elements in the list such that all

    elements smaller than the pivot are placed to the left of it,

    and all elements greater than the pivot are placed to the right

    of it.

    The input list 'data' is modified during the partitioning

    process.

    Args:

        data: The list of elements to be partitioned.

        i: The starting index of the partitioning range.

        j: The ending index of the partitioning range.

        pivot: The pivot element used for partitioning.

    Returns:

        The index of the pivot element after partitioning.

    '''

    global count

    while True:

        count += 1

        while *data*[*i*] < *pivot*:

*i* += 1

        while *data*[*j*] > *pivot*:

*j* -= 1

        if *i*>=*j*:

            break

*data*[*i*] , *data*[*j*] = *data*[*j*] , *data*[*i*]

*j* -= 1

*i* += 1

*data*[*i*] , *data*[len(*data*)-1] = *data*[len(*data*)-1] , *data*[*i*]

    return *i*

*def* getmid\_index(*arr*, *left*, *right*):

    '''

    This function determines the index of the middle element

    in a given list 'arr' between the indices 'left' and 'right'.

    It considers three elements: the element at the 'left' index,

    the element at the 'right' index, and the element at the

    calculated middle index.

    The input list 'arr' is not modified.

    Args:

        arr: The list of elements.

        left: The left index of the range.

        right: The right index of the range.

    Returns:

        The index of the middle element between 'left' and 'right'.

    '''

    global count

    mid = (*left* + *right*) // 2

    count += 1

    if *arr*[*left*] <= *arr*[mid] <= *arr*[*right*] or *arr*[*right*] <= *arr*[mid] <= *arr*[*left*]:

        return mid

    elif *arr*[mid] <= *arr*[*left*] <= *arr*[*right*] or *arr*[*right*] <= *arr*[*left*] <= *arr*[mid]:

        return *left*

    else:

        return *right*

*def* quick\_sort(*data*):

    '''

    This function implements the quicksort algorithm to sort a given

    list of data in ascending order. It recursively divides the list

    into smaller sublists, selects a pivot element, partitions the

    list based on the pivot, and then sorts the sublists individually.

    The input list 'data' is not modified. The function creates new

    lists during the sorting process.

    Args:

        data: The list of elements to be sorted.

    Returns:

        A new list containing the sorted elements of the input list.

    '''

    global count

    if len(*data*) > 1:

        pivot\_ind = getmid\_index(*data*,0,len(*data*)-1)

        pivot = *data*[pivot\_ind]

*data*[pivot\_ind], *data*[len(*data*)-1] = *data*[len(*data*)-1], *data*[pivot\_ind]

        i = 0

        j = len(*data*) - 2

        count += 1

        i = partition(*data*,i,j,pivot)

        return quick\_sort(*data*[:i]) + [pivot] + quick\_sort(*data*[i+1:])

    else:

        return *data*

*def* random\_list(*size*):

    '''

    The given function generates a random number of values

    and returns the values in a list.

    args:

        size: the number of point objects to be

        generated

    Returns:

        A list of random integer values.

    '''

    random\_list = []

    for case in range(*size*):

        x\_val = random.randint(-1000,1000)

        random\_list.append(x\_val)

    return random\_list

*def* worst\_scenario(*size*):

    '''

    The function provides the parameters required for

    printing a worst case scenario for sorting.

    The input is not modified and there are no side effects.

    args:

        size: the size of random list to be generated

    Returns:

        None

    '''

    random\_lst = [x for x in range(*size*,0,-1)]

    quick\_sort(random\_lst)

    print("Worst case scenario is:", count)

*def* best\_scenario(*size*):

    '''

    The function provides the parameters required for

    printing a best case scenario for sorting.

    The input is not modified and there are no side effects.

    args:

        size: the size of random list to be generated

    Returns:

        None

    '''

    random\_lst = [x for x in range(*size*)]

    quick\_sort(random\_lst)

    print("Best case scenario is:", count)

*def* avg\_scenario(*size*):

    '''

    The function provides the parameters required for

    printing an average case scenario for sorting.

    The input is not modified and there are no side effects.

    args:

        size: the size of random list to be generated

    Returns:

        None

    '''

    random\_lst = random\_list(*size*)

    quick\_sort(random\_lst)

    print("Average case scenario is:", count)

#driver code

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

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    count = 0

    random\_lst = random\_list(20)

    print(*f*"Random list is : {random\_lst}")

    sorted\_val = quick\_sort(random\_lst)

    print(*f*"Sorted list is: {sorted\_val}")

    print(*f*"Count is : {count}")

    k = 10

    while k <= 10000:

        count = 0

        start = time.time()

        quick\_sort(random\_list(k))

        end = time.time()

        print(*f*"k is {k} and count is : {count}")

        print(*f*"Time taken is: {end - start}")

        print()

        k \*= 10

    k = 20000

    count = 0

    start = time.time()

    quick\_sort(random\_list(k))

    end = time.time()

    print(*f*"k is {k} and count is : {count}")

    print(*f*"Time taken is: {end - start}")

    print()

    k = 50000

    count = 0

    start = time.time()

    quick\_sort(random\_list(k))

    end = time.time()

    print(*f*"k is {k} and count is : {count}")

    print(*f*"Time taken is: {end - start}")

    print()

    print("For size 1000:\n")

    count = 0

    avg\_scenario(1000)

    count = 0

    best\_scenario(1000)

    count = 0

    worst\_scenario(1000)

III. OUTPUT:

Random list is : [545, -964, 732, -722, 42, -354, -425, 259, 691, 658, 864, 994, 919, 727, 330, -659, 222, -108, -12, -253]

Sorted list is: [-964, -722, -659, -425, -354, -253, -108, -12, 42, 222, 259, 330, 545, 658, 691, 727, 732, 864, 919, 994]

Count is : 47

k is 10 and count is : 21

Time taken is: 0.0

k is 100 and count is : 282

Time taken is: 0.0010023117065429688

k is 1000 and count is : 3613

Time taken is: 0.004009723663330078

k is 10000 and count is : 46902

Time taken is: 0.06917357444763184

k is 20000 and count is : 101747

Time taken is: 0.09500813484191895

k is 50000 and count is : 280753

Time taken is: 0.2534780502319336

For size 1000:

Average case scenario is: 3681

Best case scenario is: 1533

Worst case scenario is: 1999