

PRACTICAL FILE

**BE (CSE) 3 rd Semester
(Group:3)**

Algorithms Design and Analysis

Jan 2020- May 2020

**Submitted By
KSHITIJ SALUJA
Roll Number:UE183045**

**Submitted To
Dr. Ajay Mittal**

Computer Science and Engineering



**University Institute of Engineering and Technology
Panjab University, Chandigarh – 160014, INDIA**

INDEX

S.No	Name of practical/ Program	Date Of Submission	Page No	Sign	Remarks
1	REPORT 1 Linear Search	29/01/2020	3-12		
2	REPORT 2 Binary Search	05/02/2020	13-23		
3	REPORT 3 N-Array Search	19/02/2020	24-32		
4	REPORT 4 Merge Sort	19/02/2020	32-40		
5	REPORT 5 Quick Sort for unsorted list	19/02/2020	41-51		
6	REPORT 5 Quick Sort for sorted list	19/02/2020	41-51		
7	REPORT 5 Quick Sort with median as pivot	19/02/2020	41-51		
8	REPORT 6 Knap Sack Program	04/06/2020	52-54		
9	REPORT 7 Prims Algorithm	04/06/2020	55-58		
10	REPORT 8 Kruskal Algorithm	04/06/2020	59-62		
11	REPORT 9 Sum of Subsets	04/06/2020	63-66		
12	REPORT 10 M-Coloring	04/06/2020	67-70		
13	REPORT 11 N Queens	04/06/2020	71-74		
14	REPORT 12 Jolly Number	04/06/2020	75-77		
15	REPORT 13 WERTU	04/06/2020	78-79		

REPORT 1

Objective:

Analysis of time complexity of linear search by plotting a graph between execution time and number of elements in the list.

Input:

The input is :

- List of x random integers in the range (0 to $x + \text{int}(x * 0.1)$).
- A key which is generated randomly and is in the range (0 to $x + \text{int}(x * 0.1)$).
- Input the maximum size of the list which starts with 500 with a gap of 500.

Eg: If the maximum number of elements are 1500 then the analysis is done for 500, 1000 and 1500 elements.

Output:

- The algorithm either returns the index at which element is found or returns -1 if the element is not found.
- It outputs the percentage of the number of not found elements in the whole program.
- It plots the graph between the mean of 100 values of execution time and the number of elements in the list.
- It plots histograms for the frequency vs the number of elements in the graph. The frequency obtained is for 100 iterations on the same the number of elements.

- It plots the graph between the mean of 100 values of execution time vs the number of elements in the list and mean of x (number of elements in the list) values of execution time vs the number of elements in the list in the same graph.
- It gives a file as output which consists of following columns :
 1. Execution time of each iteration
 2. Index at which element is found(if not found -1)
 3. Status i.e found or not found.
 4. Number of elements in the list

Procedure:

Let arr be the list of size n in which the key(x) has to be found .

Algorithm:

```

Linear Search(arr ,n ,x){
    for t in range(n):
        if x==arr[i]:
            return i
        break
    return -1
}

```

In the above algorithm the list is traversed and each element of the list is compared with the key .The algorithm returns

index at which the key is found and if the key is not present it returns -1.

Program to analyse time complexity of linear search:

```
import time
import numpy as np
from matplotlib import pyplot as plt
import random
import pandas as pd
from tabulate import tabulate
from collections import Counter
i=0
list11=[]
result=[]
re=[]
runtime=[]
ri=[]
def linear1():
    for t in range(x):
        if key == list1[t]:
            break
    return t
def linear2():
    for t in range(x):
        if key == list12[t]:
            break
    return t
list112=[]
result2=[]
re2=[]
runtime2=[]
ri2=[]
for x in range(500,1000,500):
    list112.append(x)

    s=0
    for u in range (100):
        list12=[]
        for y in range(x):
            list12.append(random.randint(0,x+int(x*0.1)))
```

```

key=random.randrange(0,(x+int(x*0.1)))
start=time.perf_counter()
t=linear2()
if t==(x-1):
    result2.append(-1)
    re2.append("Not found")
else :
    result2.append(t)
    re2.append("found")
stop=time.perf_counter()
s=s+(stop-start)
ri2.append(stop-start)
runtime2.append(s/100)
for x in range(500,4000,500):
    list11.append(x)

s=0
for u in range (x):
    list1=[]
    for y in range(x):
        list1.append(random.randint(0,x+int(x*0.1)))
    key=random.randrange(0,(x+int(x*0.1)))
    start=time.perf_counter()
    t=linear1()
    if t==(x-1):
        result.append(-1)
        re.append("Not found")
    else :
        result.append(t)
        re.append("found")
    stop=time.perf_counter()
    s=s+(stop-start)
    ri.append(stop-start)
runtime.append((s/x))
data=[]
u=result.count(-1)
y=len(result)
print(u/y)
for h in range(len(ri)):
    data.append([ri[h],result[h],re[h]])
df = pd.DataFrame(data)
filename="analysis"

```

```
f=open(filename,"w")
f.write(tabulate(df))
f.close()
plt.title('analyze')
plt.xlabel('number')
plt.ylabel('time')
plt.plot(list11, runtime)
plt.plot(list112, runtime2)
plt.show()
```

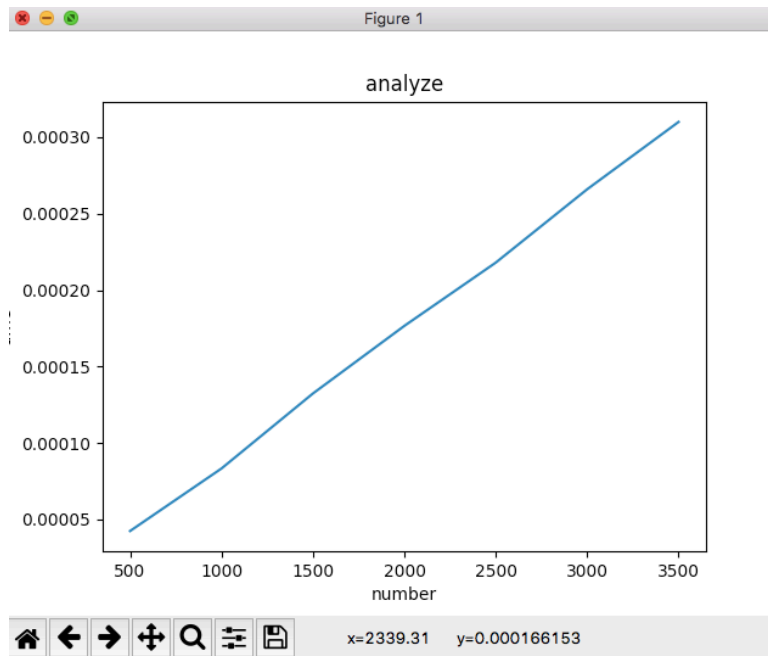
Explanation

The program defines a function linear which return the index at which the element is found else it returns -1 .The program calculates the execution time for searching for each iteration .The program starts with taking 500 elements upto 4000 with a stride of 500.It repeats on each stride 100 times and calculates the mean execution time for these iterations.It also finds the percentage of not found elements in the graph.Then it plots the graph using matplotlib library of mean execution time vs the number of elements in the graph. The file “analysis” is formed using pandas library which helps in making the table with columns as runtime time, index at which element if found and status.The program also plots a graph to compare mean execution time for 100 iterations as well as mean execution time for n number of iterations.

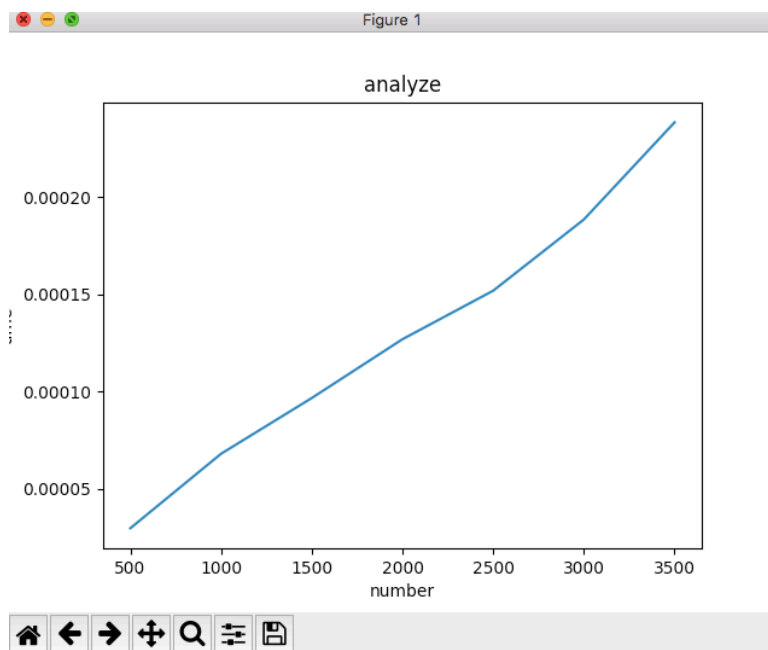
Output:

The output is as follows :

Graph: Run 1

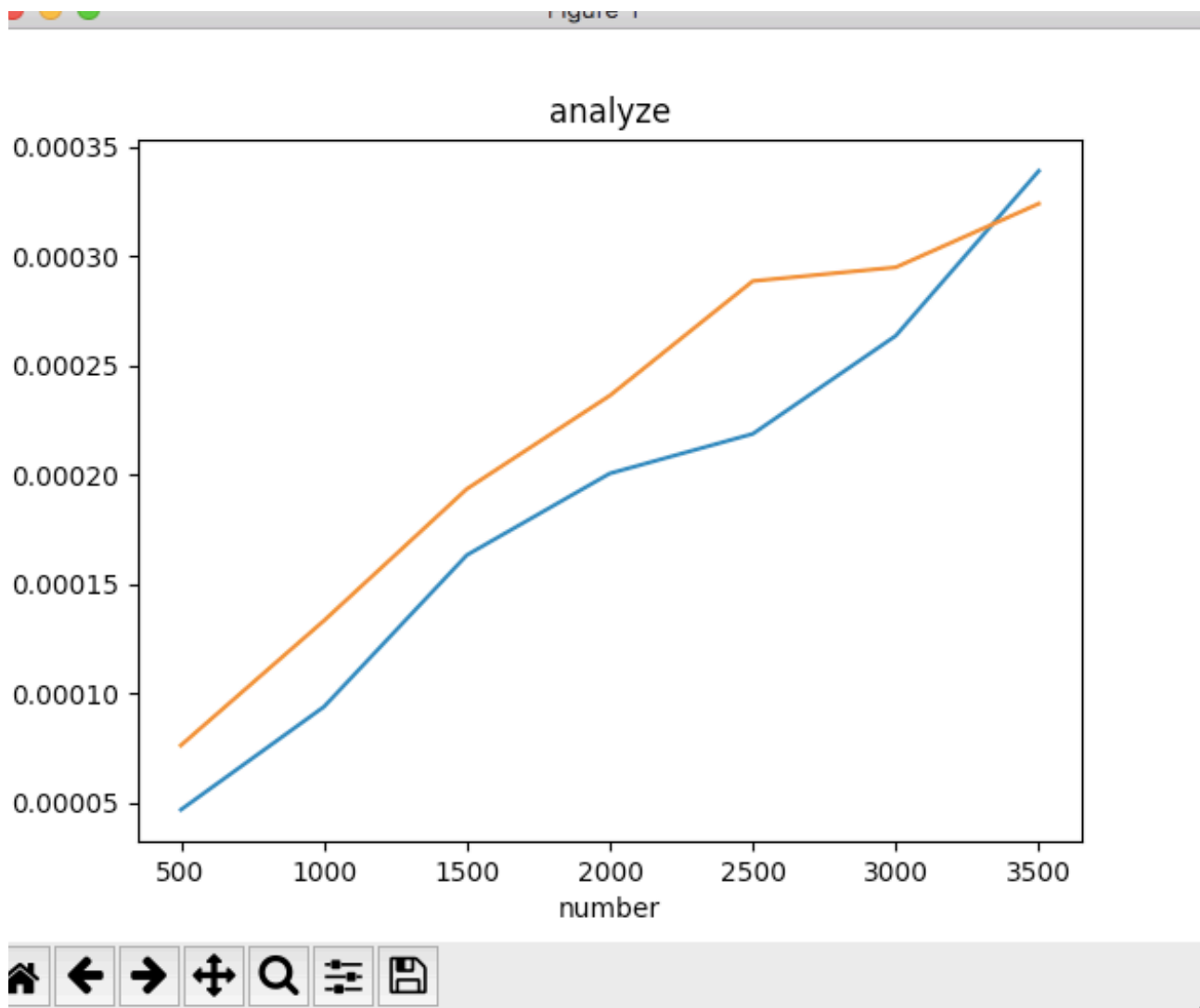


Run 2



Scale :- y-axis: 1 unit = 0.0005 sec
x-axis: 1 unit = 500 units

Comparison graph:



The blue line shows the plot when 100 iterations are taken and the orange line shows the plot when number of iterations are equal to the number of elements.

We also find the percentage of not found elements in the whole program. It comes out to be 39.51 % .It is significant because it tells around 40 % of the time the whole lists traversed which makes it take more execution time .

Table to show mean execution time:

S No	Iterations	Mean Execution Time
1	500	3.34036E-05
2	1000	7.11941E-05
3	1500	0.000118324
4	2000	0.000161976
5	2500	0.000198251
6	3000	0.00022828
7	3500	0.00026804

This table shows mean execution time for each particular number of iterations .There is another file which shows the execution time of each iteration and also tells the index of the key if found in the list.

S.No	Execution Time	Index	Status
0	1.2324E-05	68	found
1	1.0168E-05	-1	Not found
2	1.9119E-05	-1	Not found
3	1.675E-05	-1	Not found
4	7.008E-06	90	found
5	1.7763E-05	10	found
6	1.6306E-05	480	found
7	1.2781E-05	406	found
8	1.7602E-05	258	found
9	1.0494E-05	-1	Not found
10	7.602E-06	245	found
11	7.661E-06	37	found
12	7.592E-06	482	found
13	1.0224E-05	-1	Not found
14	7.235E-06	289	found
15	1.046E-05	287	found
16	8.645E-06	-1	Not found
17	1.1336E-05	-1	Not found
18	7.33E-06	201	found
19	7.013E-06	393	found
20	1.3326E-05	-1	Not found

Analysis

From the graph obtained we can conclude that relationship between execution time and size of the list is linear in nature . As the size of the list increases the execution time also increases linearly .

We can also see that the percentage of not found elements is around 40 % Hence 40 % of the times the list is fully traversed increasing the execution time .It can be seen there are some peaks in the graph also these are because of the variation in the execution time of the graph .Because the numbers and the key is generated randomly so the might be found in the beginning or in the end or never found .

We made two tables in order to show the analysis of execution time, index at each iteration and the size of the list .It also shows if the number of elements increase then the the mean execution time also increases.

REPORT 2

Objective:

Analysis of time complexity of binary search by plotting a graph between execution time and number of elements in the list.

Input:

The input is :

- List of x random integers in the range (0 to $x + \text{int}(x * 0.9)$).
- A key which is generated randomly and is in the range (0 to $x + \text{int}(x * 0.9)$).
- Input the maximum size of the list which starts with 500 with a gap of 500.

Eg: If the maximum number of elements are 1500 then the analysis is done for 500, 1000 and 1500 elements.

Output:

- The algorithm returns the number of comparisons of to be done to find the key .
- It outputs the percentage of the number of not found elements in the whole program.
- It plots the graph between the mean of 100 values of execution time and the number of elements in the list.
- It plots the graph to compare linear search and binary search in order to compare the mean execution time for them on set of values and the key .

- It plots the graph to compare execution time of iterative and recursive binary search .
- It plots the graph of %Not found elements vs the number of elements in the list .
- It gives a file as output which consists of following columns :
 1. Execution time of each iteration
 2. Number of comparisons for each iteration
 3. Status i.e found or not found.
 4. Number of elements in the list

Procedure:

Let arr be the list of size n in which the key(y) has to be found .

Algorithm:

```
def binary search (arr,l,r,y,i){
    while r >= l:
        mid=int((l+r)/2)

        if arr[mid] == y:
            return i
        elif arr[mid] > y:
            r=mid-1
        else:
            l=mid+1
        i=i+1
    return l
}
```

In the above algorithm the list is traversed and each element of the list is compared with the key .The algorithm returns the number of comparisons to find the key in the randomly generated list

Program to analyse time complexity of linear search:

```
import time
import numpy as np
from matplotlib import pyplot as plt
import random
import pandas as pd
from tabulate import tabulate
from collections import Counter
import math
i=0
no_of_elements=[]
result=[]
status=[]
runtime=[]
time_counter=[]
comparison=[]
n_found_arr=[]
def bsearch (arr,l,r,y,i):
    while r >= l:
        mid=int((l+r)/2)

        if arr[mid] == y:
            return i
        elif arr[mid] > y:
            r=mid-1
        else:
            l=mid+1
        i=i+1
    return i
for x in range(500,4001,500):
    no_of_elements.append(x)
    mean_time=0
```

```

n_found=0
for u in range (100):
    arr=[]
    for y in range(x):
        arr.append(random.randint(0,x+int(x*0.9)))
    arr.sort()
    #print(arr)
    key=random.randrange(0,(x+int(x*0.9)))
    #print(key)
    i=0
    start=time.perf_counter()
    t=bsearch(arr,0,(len(arr)-1),key,i)
    stop=time.perf_counter()
    if t == math.floor(math.log(x,2)+1):
        result.append(t)
        status.append("Not found")
        n_found=n_found+1
    else :
        result.append(t)
        status.append("found")

    mean_time=mean_time+(stop-start)
    time_counter.append(stop-start)
n_found_arr.append(n_found)
runtime.append(mean_time/100)
data=[]
o=status.count("Not found")
y=len(status)
print(o/y)
for h in range(len(time_counter)):
    data.append([time_counter[h],result[h],status[h]])
df = pd.DataFrame(data)
filename="bsearch3_analysis"
f=open(filename,"w")
f.write(tabulate(df))
f.close()
plt.title('analyze')
plt.xlabel('number')
plt.ylabel('time')
plt.plot(no_of_elements,runtime)
plt.show()
plt.title('analyze2')

```



```
plt.xlabel('number')
plt.ylabel('%not found')
plt.plot(no_of_elements,n_found_arr)
plt.show()
```

Explanation

The program defines a function `bsearch` which return the number of comparisons for each iteration .If the number of comparisons are equal to log of the number of elements then the key is not found in that particular iteration. The program calculates the execution time for searching for each iteration .The program starts with taking 500 elements upto 4000 with a stride of 500.It repeats on each stride 100 times and calculates the mean execution time for these iterations.It also finds the percentage of not found elements in the graph.Then it plots the graph using matplotlib library of mean execution time vs the number of elements in the graph. The file “analysis” is formed using pandas library which helps in making the table with columns as runtime time, index at which element if found and status.The program also plots a graph to compare mean execution time of iterative and recursive binary search .It also plots a graph in order to plot the percentage of not found elements for each stride of elements .

Output:

The output is as follows :

Graph: Run 1

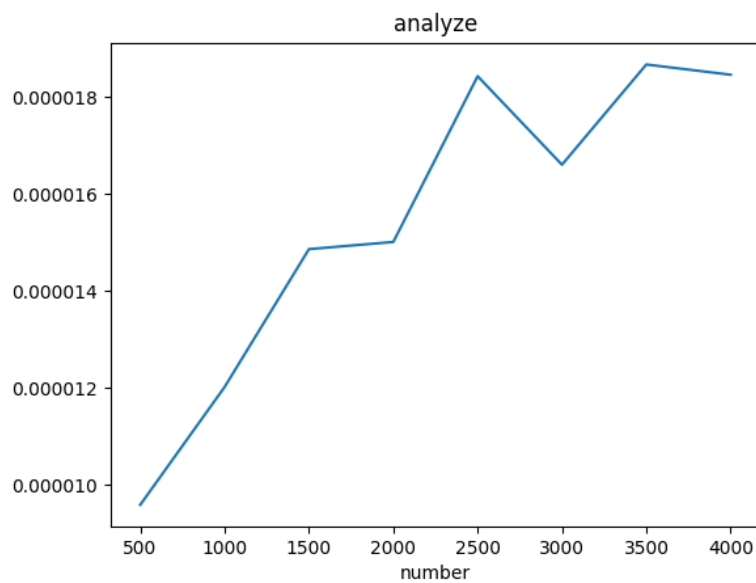


FIG-1

Run 2

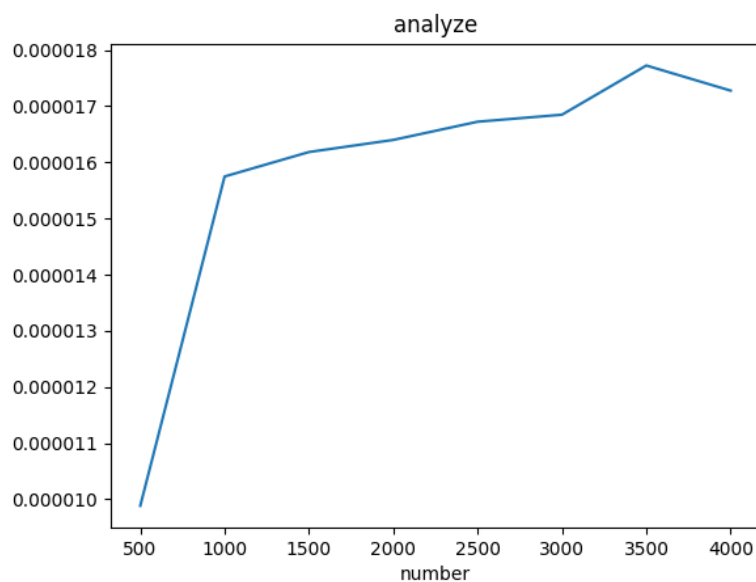


FIG-2

Scale :- y-axis: 1 unit = 0.00001 sec
x-axis: 1 unit = 500 units

Comparison graph:

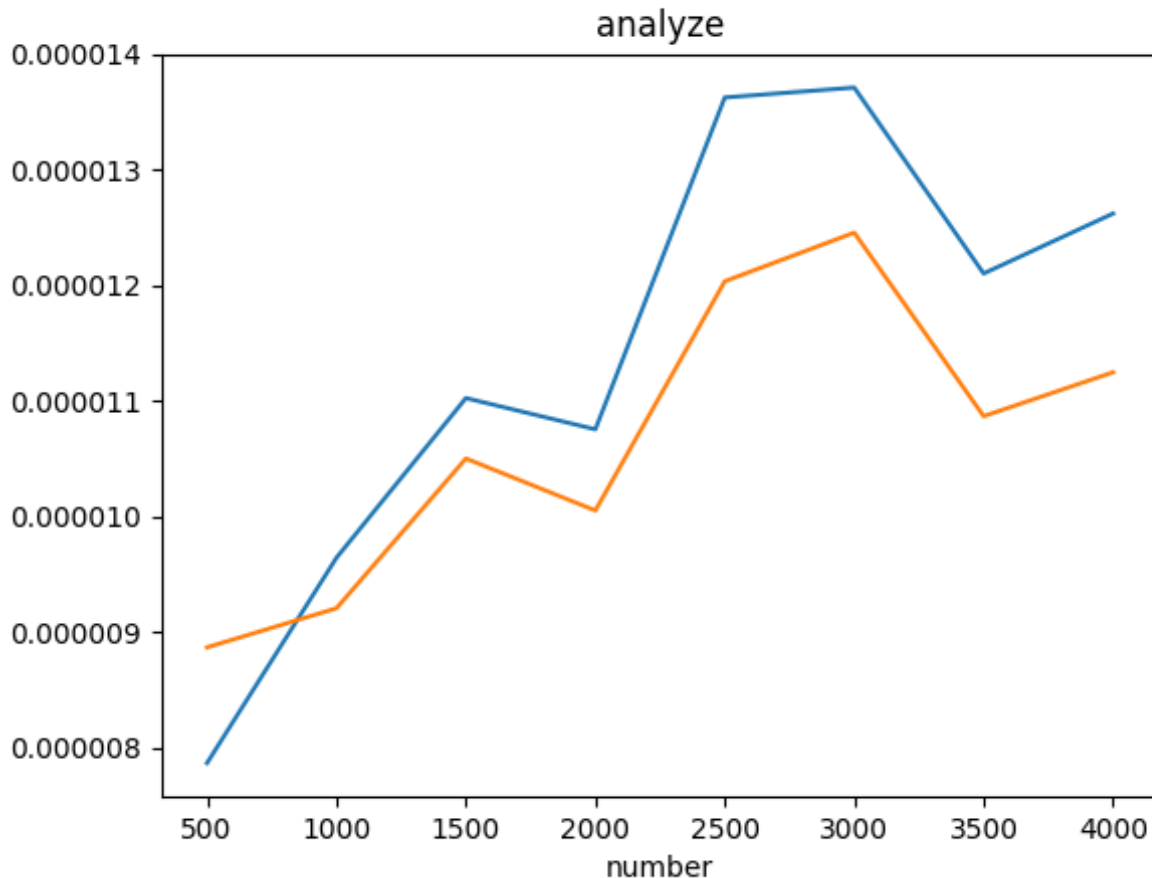


FIG-3

The blue line shows the mean execution time of 100 iterations of iterative binary search and the orange line shows the mean execution time of 100 iterations of recursive binary search . The plot clearly shows that the time taken for iterative binary search is more than the time taken for recursive binary search.

Not Found%



FIG-4

This plot shows the percentage of not found elements for **FIG-2**. Overall the percentage of not found elements is 49.5 %. This plot shows the number of not found keys for each particular number of elements for 100 iterations. It is noticed that in **FIG-4** the number of element decrease to a great extent when number of elements are 2500 but are on a higher side when number of elements are 500 or 3500 .

Table to show mean execution time:

S.NO	No Of elements	Mean Execution Time
0	500	7.8201E-06
1	1000	8.06488E-06
2	1500	8.41644E-06
3	2000	9.14833E-06
4	2500	9.38485E-06
5	3000	9.99299E-06
6	3500	1.13977E-05
7	4000	1.25058E-05

Table-1

This table shows mean execution time for each particular number of iterations .There is another file which shows the execution time of each iteration and also tells the number of comparisons in order to find the key in the list .Table-1 also shows that if the number of elements increase than the mean execution time also increases .

S.NO	Execution time	No. Of Comparisons	Status
0	7.371E-06	9	Not found
1	7.202E-06	9	Not found
2	4.604E-06	5	Found
3	8.521E-06	9	Not found
4	6.91E-06	9	Not found
5	7.115E-06	8	Found
6	5.525E-06	8	foun
7	6.518E-06	9	Not found
8	5.512E-06	9	Not found
9	6.498E-06	9	Not found
10	9.489E-06	8	Found
11	5.786E-06	8	Found
12	6.51E-06	8	Found
13	5.682E-06	9	Not found
14	5.583E-06	9	Not found
15	5.516E-06	9	Not found
16	5.429E-06	9	Not found
17	8.496E-06	8	Found
18	5.725E-06	6	Found
19	6.794E-06	9	Not found
20	6.267E-06	8	Found

Table-2

Analysis

From the graph obtained we can conclude that relationship between execution time and size of the list is logarithmic in nature . As the size of the list increases the execution time also increases logarithmically .

The percentage of not found elements is around 50 % for the case of **FIG-1**. Hence 50 % of the times the list is fully traversed increasing the execution time .

We can also see from **FIG-3** that the execution time for iterative binary search is more than the recursive binary search .

It is clear from **Table-1** that as the number of elements increase that the mean execution time also increases . This is because with more number of elements the the number of comparisons also increase .

REPORT 3

PROJECT-1

Objective:

Analysis of execution time taken by n-array search algorithm and its time complexity.

Input:

The input is :

- List of x random integers in the range $(0 \text{ to } x + \text{int}(x * 0.5))$.
- A key which is generated randomly and is in the range $(0 \text{ to } x + \text{int}(x * 0.5))$.
- Input the maximum size of the list which starts with 5000 with a gap of 5000.

Eg: If the maximum number of elements are 15000 then the analysis is done for 5000, 10000 and 15000 elements.

- The number of division to be performed on the list.

Output:

- The algorithm returns the index at which the key is found.
- It outputs the percentage of the number of not found elements in the whole program.
- It plots the graph between the mean of 100 values of execution time and the number of divisions on the list.
- It plots the graph of %Not found elements vs the number of elements in the list.
- It gives a file as output which consists of following columns :
 1. Execution time of each iteration
 2. Number of comparisons for each iteration
 3. Status i.e found or not found

Procedure:

Let arr be the list of size n in which the key(y) has to be found.

Algorithm:

```
def n_search (l,h,key,arr,m){
    partzn=[]
    for i in range(m-1):
        a=int((h-l)/m)
        #print(a)
        partzn.append(((i+1)*a)+l)

    if(h-l<=m-2):
        for i in range(l,(h+1)):
            if(keyx==arr[i]):
                return i
        return -1

    i=0
    while(i<(m-1)):
        if(keyx==arr[partzn[i]]):
            return partzn[i]

        elif(keyx<arr[partzn[i]]):
            return n_search(l,partzn[i]-1,keyx,arr,m)

        else:
            l=partzn[i]+1
            i=i+1
    return n_search(l,h,keyx,arr,m)
}
```

In the above algorithm (m-1) cut points are found. These are found using the formula $(i+1)*a+l$ where $a=\text{int}((h-l)/m)$ and l is the cut point. Then we find the

partition set to which the key lies. On finding the set recursion is implemented in order to find the key in that set.

Program to analyse time complexity of n-arry search:

```
import time
import numpy as np
from matplotlib import pyplot as plt
import random
import pandas as pd
from tabulate import tabulate
from collections import Counter
import math
i=0
list11=[]
result=[]
result2=[]
status=[]
ri=[]
runtime=[]
comparison=[]
def n_search(l,h,keyx,arr,m):
    partzn=[]
    for i in range(m-1):
        a=int((h-l)/m)
        #print(a)
        partzn.append(((i+1)*a)+l)

    if(h-l<=m-2):
        for i in range(l,(h+1)):
            if(keyx==arr[i]):
                return i
        return -1

    i=0
    while(i<(m-1)):
        if(keyx==arr[partzn[i]]):
            return partzn[i]

        elif(keyx<arr[partzn[i]]):
            return n_search(l,partzn[i]-1,keyx,arr,m)

        else:
            l=partzn[i]+1
            i=i+1
    return n_search(l,h,keyx,arr,m)
for x in range(2,10,1):
```

```

list11.append(x)
s=0
for u in range (100):
    arr1=[]
    for y in range(5000):
        arr1.append(random.randint(0,5000+int(5000*0.5)))
    key=random.randrange(0,(5000+int(5000*0.5)))
    arr1.sort()
    start=time.perf_counter()
    t=n_search(0,4999,key,arr1,x)
    stop=time.perf_counter()
    if t==-1:
        result2.append(-1)
        status.append("Not found")
    else:
        result2.append(t)
        status.append("found")
    s=s+(stop-start)
    ri.append(stop-start)
runtime.append(s/100)
data=[]
u=result.count(-1)
y=len(result)
print(u/y)
for h in range(len(list11)):
    data.append([list11[h],runtime[h]])
df = pd.DataFrame(data)
filename="n-arry-search2_analysis"
f=open(filename,"w")
f.write(tabulate(df))
f.close()
plt.title('analyze')
plt.xlabel('number')
plt.ylabel('time')
plt.plot(list11,runtime)
plt.show()

```

Explanation

The program defines a function `n_search` which returns the index at which the element is found depending upon the number of partitions the function has to make. The number of elements have been kept constant and the number of divisions vary from 2 to 20. The runtime for each iteration is calculated and the mean of 100 iterations is found. The number of divisions vs the mean execution time is plotted. A file is also created which consists of the number of divisions and the mean execution time of each division.

Output:

The output is as follows :

Graph: Run 1

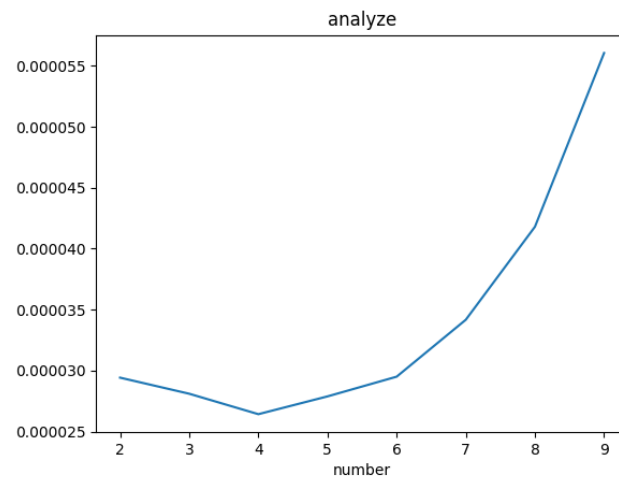


FIG-1

Scale :- y-axis: 1 unit = 0.00005 sec
x-axis: 1 unit = 1 unit

Run 2

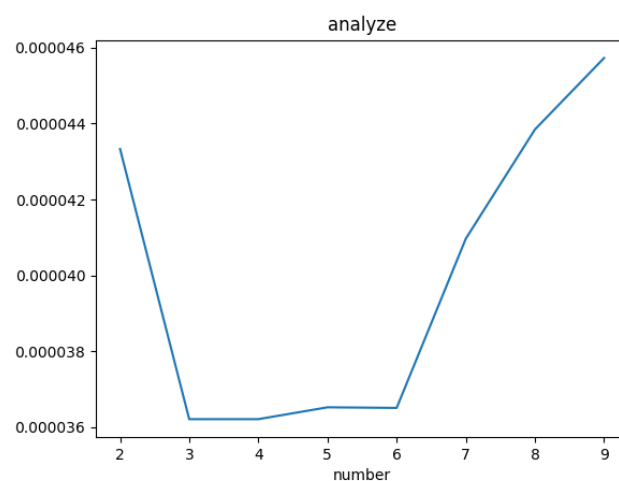
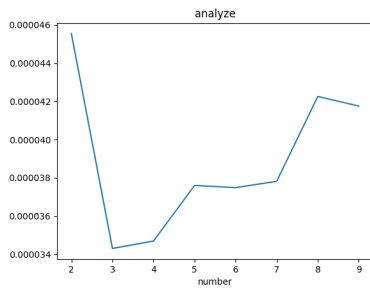


FIG-2

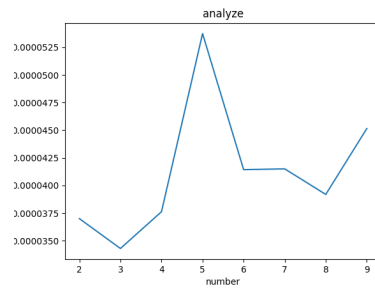
Scale :- y-axis: 1 unit = 0.00002 sec
x-axis: 1 unit = 1 unit

In **FIG-1** it is observed that the mean execution time is minimum at 4. In **FIG-2** it is observed that mean execution time is minimum at 3 but at 4 also the mean execution time is almost the same.

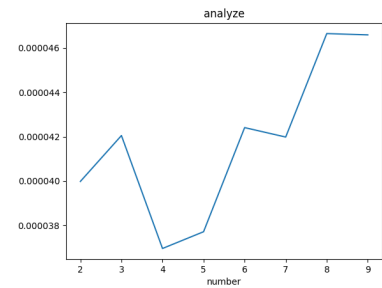
Comparison graph:



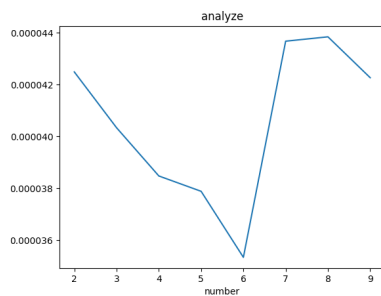
5000 elements.



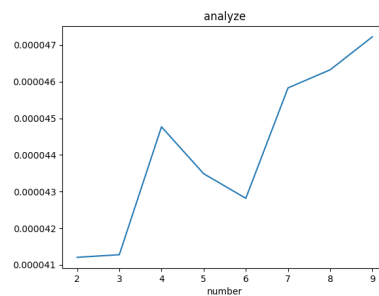
10000 elements



15000 element



20000 elements.



25000 elements

FIG-3

These five plots show the partition on varying the number of elements in the list. It can be seen that on caring the number of elements in the list the partition at which the min execution time arises also keeps on varying.

Not Found%

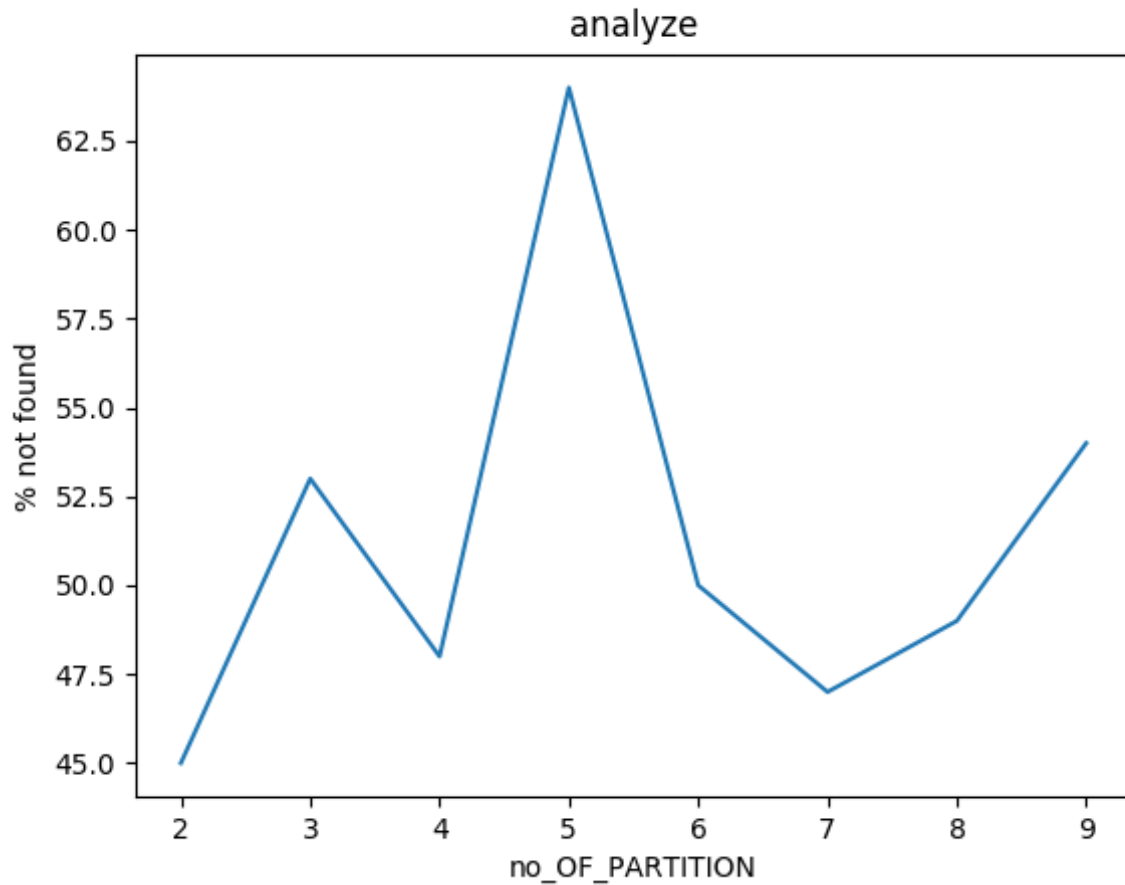


FIG-4

This plot shows the percentage of not found elements for **FIG-1**. Overall the percentage of not found elements is 51.25%. This plot shows the number of not found keys for each particular number of partitions for 100 iterations.

Table to show mean execution time:

S.No	No. Of partitions	Mean execution time
0	2	3.13178E-05
1	3	2.8786E-05
2	4	3.08003E-05
3	5	3.9589E-05
4	6	4.29813E-05
5	7	3.84924E-05
6	8	4.91227E-05
7	9	4.31845E-05
8	10	4.61053E-05
9	11	5.42517E-05
10	12	4.47932E-05
11	13	5.46133E-05
12	14	6.01761E-05
13	15	5.59147E-05
14	16	4.95268E-05
15	17	5.15667E-05
16	18	6.07993E-05
17	19	5.94008E-05

Table-1

This table shows mean execution time for each particular number of partitions. It can be seen that the mean execution time is minimum when the number of partitions are equal to 3. After that also it still gets some local minimal in between.

Analysis

From **FIG-3** it can be seen that the minimum mean execution time varies with the number of elements in the list. It can be seen that minimum execution time is reached when number of partitions are between 3 and 6.

From **Table-1** it is clear that global minima is reached when no of partitions are 3 and there are occasional local minima which is evident from seeing the table.

FIG-4 tells about the percentage of not found elements which keeps on varying from the range from 45 in binary search and 65 in 5-arry search

REPORT 4

Objective:

Analysis of execution time taken by merge sort by the varying the number of elements in the list.

Input:

The input is :

- List of x random integers in the range (0 to $x + \text{int}(x \cdot 0.5)$).
- Input the maximum size of the list which starts with 500 with a gap of 500
Eg: If the maximum number of elements are 1500 then the analysis is done for 500, 1000 and 1500 elements.

Output:

- The algorithm returns a sorted array.
- It plots the graph between the mean of 100 values of execution time and the number of elements in the list.
- It makes a table which consists of the mean execution time and the number of elements in the list.

Procedure:

Let arr be the list which is to be sorted.

Algorithm:

```
def merge(arr, l, m, r):  
    n1 = m - l + 1  
    n2 = r - m  
  
    Left = []  
    Right = []  
  
    for i in range(0, n1):  
        Left.append(arr[l + i])  
  
    for j in range(0, n2):
```

```

        Rightappend(arr[m + 1 + j])
    i = 0
    j = 0
    k = l

    while i < n1 and j < n2 :
        if Left[i] <= Right[j]:
            arr[k] = Left[i]
            i += 1
        else:
            arr[k] = Right[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = Left[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = Right[j]
        j += 1
        k += 1

def mSort(arr,l,r):
    if l < r:
        size=0
        m = int((l+r)/2)
        size=size+mSort(arr, l, m)
        size=size+mSort(arr, m+1, r)
        merge(arr, l, m, r)
        return size+len(arr)
    else:
        return 0

```

The above algorithm has two functions. The function msort helps in creating sorted sub arrays by dividing the array into smaller parts. The function merge is used to merge the sorted sub arrays which in end gives the full sorted array.

Program to analyse time complexity of merge sort:

```

import time
import numpy as np
from matplotlib import pyplot as plt

```

```

import random
import pandas as pd
from tabulate import tabulate
from collections import Counter
import math
no_of_elements=[]
runtime=[]
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    Left= []
    Right = []

    for i in range(0 , n1):
        Left.append(arr[l + i] )

    for j in range(0 , n2):
        Right.append(arr[m + 1 + j])
    i = 0
    j = 0
    k = l

    while i < n1 and j < n2 :
        if Left[i] <= Right[j]:
            arr[k] = Left[i]
            i += 1
        else:
            arr[k] = Right[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = Left[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = Right[j]
        j += 1
        k += 1

def mSort(arr,l,r):
    if l < r:
        size=0
        m = int((l+r)/2)
        size=size+mSort(arr, l, m)
        size=size+mSort(arr, m+1, r)

```

```

        merge(arr, l, m, r)
        return size+len(arr)
    else:
        return 0
for x in range(500,4001,500):
    no_of_elements.append(x)
    mean_time=0
    n_found=0
    mean_time2=0
    for u in range (10):
        arr=[]
        for y in range(x):
            arr.append(random.randint(0,x+int(x*0.5)))
        #print(arr)
        key=random.randrange(0,(x+int(x*0.5)))
        #arr.sort()
        #print(key)
        i=0
        start=time.perf_counter()
        t=mSort(arr,0,(len(arr)-1))
        stop=time.perf_counter()
        print(t)
        mean_time=mean_time+(stop-start)
    runtime.append(mean_time/100)
data=[]
for h in range(len(runtime)):
    data.append([no_of_elements[h],runtime[h]])

df = pd.DataFrame(data)
filename="mergesort_analysis"
f=open(filename,"w")
f.write(tabulate(df))
f.close()
plt.title('analyze')
plt.xlabel('number')
plt.ylabel('time')
plt.plot(no_of_elements,runtime)
plt.show()

```

Explanation

The program defines two functions i.e msort and merge which return a sorted list. The program runs on different number of elements and computes the

mean execution time for sorting the list. It also creates a file to store the mean execution time and the number of elements in the list.

Output:

The output is as follows :

Graph:

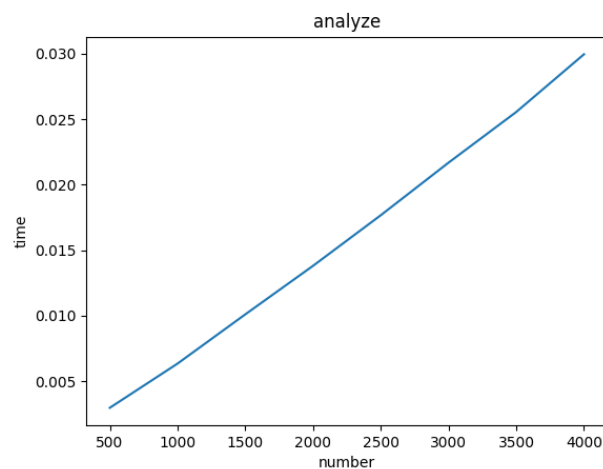


FIG-1

Scale :- y-axis: 1 unit = 0.005 sec
x-axis: 1 unit = 500 unit

The graph shown is plotted for mean execution time vs number of elements in the list. The complexity for merge sort is $O(n \log n)$. The graph shown above seems to be linear but actually represents $n \log n$.

Comparison graph:

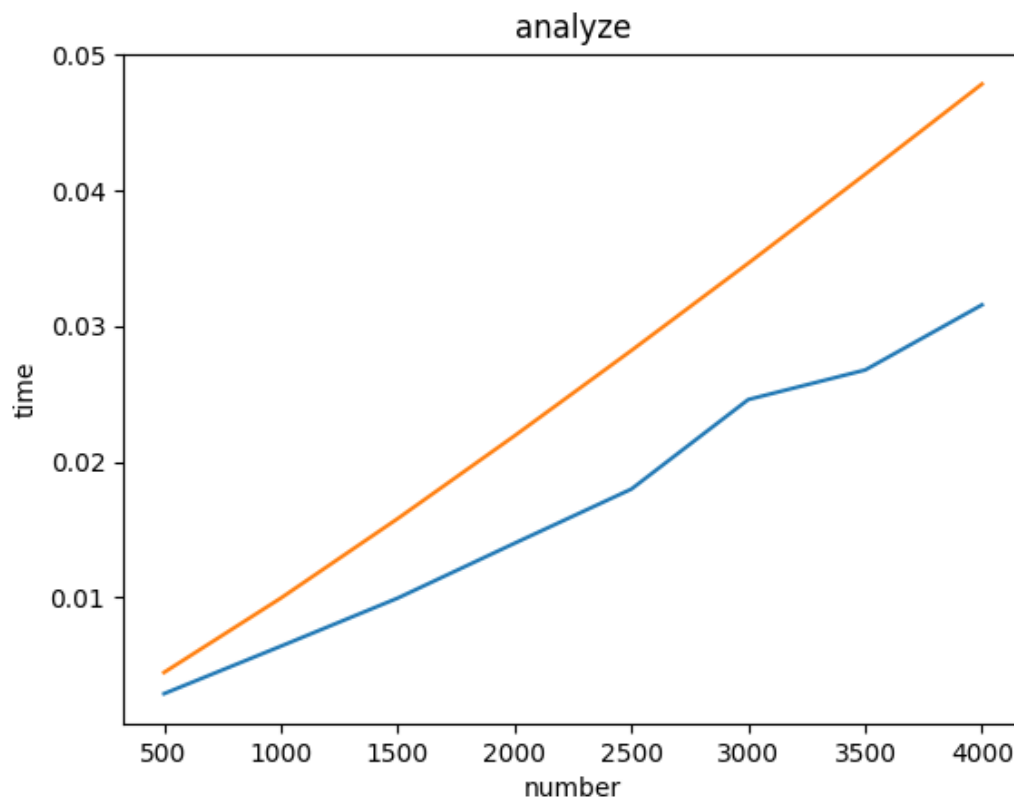


FIG-2

Scale :- y-axis: 1 unit = 0.01 sec

x-axis: 1 unit = 500 unit

The graph in **FIG-2** represents the comparison of merge sort and $n \log n$. The blue is a plot of $n \log n$ while the orange line shows the plot of merge sort. Hence it can be seen that both the graphs are very similar in nature.

AUXILIARY SPACE GRAPH:

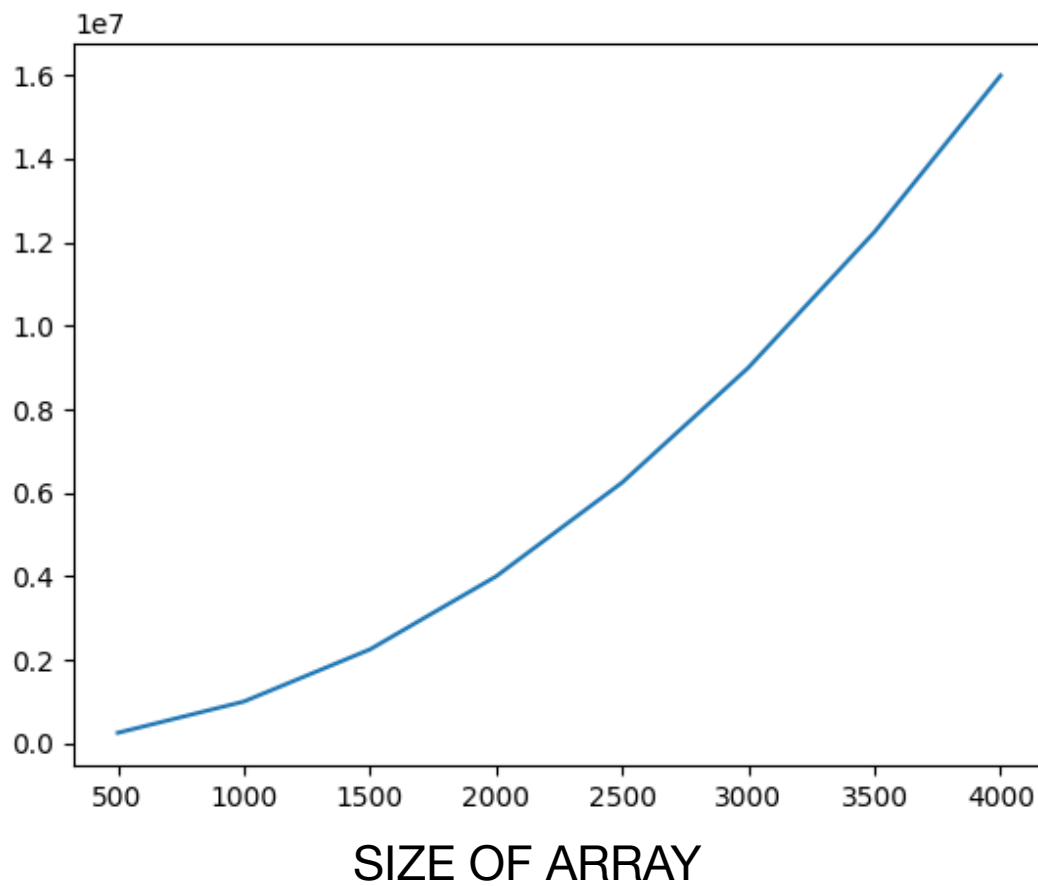


FIG-3

It can be seen that the auxiliary space required is nearly linear. The space complexity is of the order of $O(n)$

Table to show mean execution time:

S.No	No. Of elements	Mean execution time
0	500	0.00296597
1	1000	0.00634155
2	1500	0.0100999
3	2000	0.0138066
4	2500	0.01767
5	3000	0.0216832
6	3500	0.0255543
7	4000	0.0299615

As we can see from the table also, the time complexity rises almost linearly.

Analysis

From the graph obtained in **FIG-1** and **FIG-2** it is clear that the graph for merge sort is $n \log n$. Its graph is similar to linear only as a very small amount gets multiplied to a large value of n .

Table-1 also shows that the mean execution time keeps on increasing on increasing the number of elements in the list.

REPORT 5

Objective :

Analysis of time complexity of quick sort by plotting a graph between execution time and size of the list.

Input :

Input to a quick sort algorithm is a list of size n containing random integers within the range $(0 \text{ to } n + (n*0.5))$, low i.e the first index, high i.e the last index. ' n ' is generated from 500 till 4000 with a stride of 500.

E.g:- if $n = 500$, then the list is of the size of 500 and the random integers in the list from within the range $0 \text{ to } (500 + (500 * 0.5)) = 750$.

Output :

The algorithm returns a sorted list.

A list of n integers is randomly generated. The program calculates the execution time of the quick sort algorithm for each iteration i.e n by calculating 100 execution times for a particular iteration and then finding an average value for that iteration. It calculates execution time for each n from 100 to 1200 with a stride of 100. The mean execution time of 100 iterations for each value of n is stored in a list which is later used to plot a graph between execution time and size of the list.

A graph is plotted for a list of random integers in order to compare the three different quick sorts.

A graph is also plotted for a list of sorted integers in order to compare the three different types of quick sorts.

The program also gives a file as an output which consists of the following columns:

1. Iteration
2. Runtime of quick sort with pivot as last element
3. Runtime of quick sort with random pivot
4. Runtime of quick sort with pivot as median

Procedure :

Quick Sort Algorithm:

Let arr be a list of size n, low be the starting index and high be the last index.

Algorithm:

Algo QuickSort(arr, l, h) {

 if l < h:

 q = partition(arr,l,h)

 qsort_r(arr, l, q-1)

 qsort_r(arr, q+1, h) }

Algo partition(arr, l,h) {

 x=arr[h]

 i=l-1

 for j in range(l , h):

 if arr[j] < x:

 i = i+1

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

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

 return (i+1) }

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

}

Algo partition_random(arr, l, h) {

```

i = ( l-1 )
p=random.randint(l,h)
t=arr[p]
arr[p]=arr[h]
arr[h]=t
x=arr[h]

for j in range(l , h):
    if arr[j] < x:
        i = i+1
        arr[i],arr[j] = arr[j],arr[i]
arr[i+1],arr[h] = arr[h],arr[i+1]
return ( i+1 ) }
Algo RandomQuickSort(arr, l,h) {

```

```

    if l < h:

        q = r_partition(arr,l,h)
        qsort_r(arr, l, q-1)
        qsort_r(arr, q+1, h)
}
Algo partition_median(arr,l,h){

```

```

i = ( l-1 )
sub=[]
sub.append(arr[l:h+1])
#print(sub[0])
p=median(sub[0])
t=p
p=arr[h]
arr[h]=t
x=arr[h]

for j in range(l , h):
    if arr[j] < x:
        i = i+1
        arr[i],arr[j] = arr[j],arr[i]
arr[i+1],arr[h] = arr[h],arr[i+1]
return ( i+1 )
}

```

```

Algo median(l){

    if len(l) < 6:
        l.sort()
        return l[int(len(l)/2)]
    sorted_g=[]
    groups=[]
    median_g=[]
    k=0
    for i in range(0, len(l), 5):
        groups.append(l[i:i + 5])
        groups[k].sort()
        sorted_g.append(groups[k])

```

```

        if(len(groups[k]) !=5 ):
            break
        median_g.append(groups[k][2])
        k=k+1
    return median(median_g)
}

```

Algo MedianQuickSort(arr, l,h) {

```

    if l < h:

        q = partition_median(arr,l,h)
        qsort_r(arr, l, q-1)
        qsort_r(arr, q+1, h)
}

```

In the above-mentioned algorithm, the input is a list of size n , a low index and a high index to the function `quickSort(...)`. The l and h are used to calculate partition index ' q ' i.e an index before which all the values are smaller than it and after that all the values are bigger than it. The program checks if low is less than high, then it calls the function `partition(...)`. The function returns the partition index ' q '. The ' q ' divides the problem into two halves and this process continues until low becomes greater than high. The last recursive stack consists of a list of two elements which are in sorted order. After this point backtracking takes place and the recursive stack starts getting empty and the list elements start getting sorted. In the end, a sorted list is obtained.

For randomised quick sort, the functions `RandomquickSort(...)` and `partition_random(...)` are used. The only difference between normal quick sort and randomised quick sort is that in randomised quick sort, the pivot is chosen at random from the list of elements rather the high index.

For quick sort with median, the function `median` helps in calculating the median in linear time and this is used as a pivot. In this case partition is formed around the median.

Program to analyse the time complexity of Quick sort :

```
import time
import numpy as np
from matplotlib import pyplot as plt
import random
import pandas as pd
from tabulate import tabulate
from collections import Counter
import math
no_of_elements=[]
runtime=[]
runtime_m=[]
runtime_r=[]
def median(l):
    if len(l) < 6:
        l.sort()
        return l[int(len(l)/2)]
    sorted_g=[]
    groups=[]
    median_g=[]
    k=0
    for i in range(0, len(l), 5):
        groups.append(l[i:i + 5])
        groups[k].sort()
        sorted_g.append(groups[k])
        if(len(groups[k]) !=5 ):
            break
        median_g.append(groups[k][2])
        k=k+1
    return median(median_g)
def partition_m(crr,l,h):
    i = ( l-1 )
    sub=[]
    sub.append(crr[l:h+1])
    #print(sub[0])
    p=median(sub[0])
    t=p
    p=crr[h]
    arr[h]=t
    x=crr[h]

    for j in range(l , h):
        if arr[j] < x:
            i = i+1
            t=crr[i]
            crr[i]=crr[j]
            crr[j]=t
    t=crr[(i+1)]
    crr[(i+1)]=crr[h]
    crr[h]=t
```

```

    return ( i+1 )
def qsort_m(crr,l,h):
    if l < h:

        q = partition_m(crr,l,h)
        qsort_m(arr, l, q-1)
        qsort_m(arr, q+1, h)
def partition(arr,l,h):
    x=arr[h]
    i=l-1
    for j in range(l , h):
        if arr[j] < x:
            i = i+1
            t=arr[i]
            arr[i]=arr[j]
            arr[j]=t
    t=arr[(i+1)]
    arr[(i+1)]=arr[h]
    arr[h]=t
    return ( i+1 )
def r_partition(brr,l,h):
    i = ( l-1 )
    p=random.randint(l,h)
    t=arr[p]
    arr[p]=arr[h]
    arr[h]=t
    x=arr[h]

    for j in range(l , h):
        if brr[j] < x:
            i = i+1
            t=brr[i]
            brr[i]=brr[j]
            brr[j]=t
    t=brr[(i+1)]
    brr[(i+1)]=brr[h]
    brr[h]=t
    return ( i+1 )
def qsort_r(brr,l,h):
    if l < h:

        q = r_partition(brr,l,h)
        qsort_r(brr, l, q-1)
        qsort_r(brr, q+1, h)
def qsort(arr,l,h):
    if l < h:
        q = partition(arr,l,h)
        qsort(arr, l, q-1)
        qsort(arr, q+1, h)
for x in range(50,1301,50):
    no_of_elements.append(x)
    mean_time=0

```

```

mean_time_r=0
n_found=0
mean_time_m=0
for u in range (100):
    arr=[]
    brr=[]
    crr=[]
    for y in range(x):
        arr.append(random.randint(0,x+int(x*0.5)))
    for y in range(x):
        brr.append(arr[y])
        crr.append(arr[y])
    i=0
    start=time.perf_counter()
    qsort(arr,0,(len(arr)-1))
    stop=time.perf_counter()
    mean_time=mean_time+(stop-start)
    start=time.perf_counter()
    qsort_r(brr,0,(len(brr)-1))
    stop=time.perf_counter()
    mean_time_r=mean_time_r+(stop-start)
    start=time.perf_counter()
    qsort_m(crr,0,(len(crr)-1))
    stop=time.perf_counter()
    mean_time_m=mean_time_m+(stop-start)
    runtime.append(mean_time/100)
    runtime_r.append(mean_time_r/100)
    runtime_m.append(mean_time_m/100)
data=[]
for h in range(len(runtime)):
    data.append([no_of_elements[h],runtime[h],runtime_r[h],runtime_m[h]])
df = pd.DataFrame(data)
filename="quick_analysis"
f=open(filename,"w")
f.write(tabulate(df))
f.close()
plt.title('analyze')
plt.xlabel('number')
plt.ylabel('time')
plt.plot(no_of_elements,runtime)
plt.plot(no_of_elements,runtime_r)
plt.plot(no_of_elements,runtime_m)
plt.show()

```

Explanation :

The program defines functions `quickSort(...)` and `partition(...)` for performing quick sort on a list of elements. The program creates a list of size n where $n \in [500, 1200)$ incrementing by a factor of 200. The elements in the list are randomly generated from within the range of 0 to $[n + (n * 0.5)]$. 100 execution times are calculated for each value of n and the average of those 100 values is appended into a list '*quick_analysis*'. The program helps in performing all the three types of quick sort and also comparing them on the basis of mean execution time by varying the number of elements in the list.

Output :

The output is as follows:

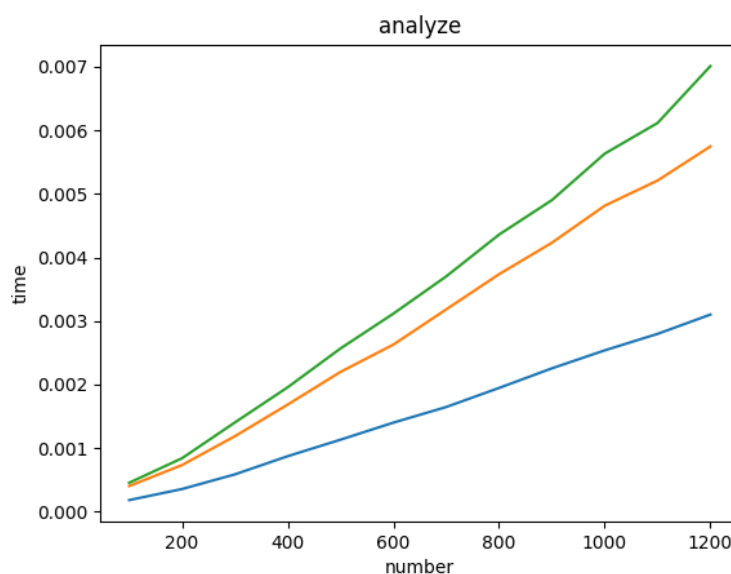


FIG-1

Scale :- y-axis: 1 unit = 0.001 sec
x-axis: 1 unit = 200 unit

The graph shows that the time taken in quick sort by finding the median is the greatest. Randomised quick sort also takes more time than the quick sort with last element as pivot. The list taken is a list of randomly generated numbers in the range $(n+0.5*n)$.

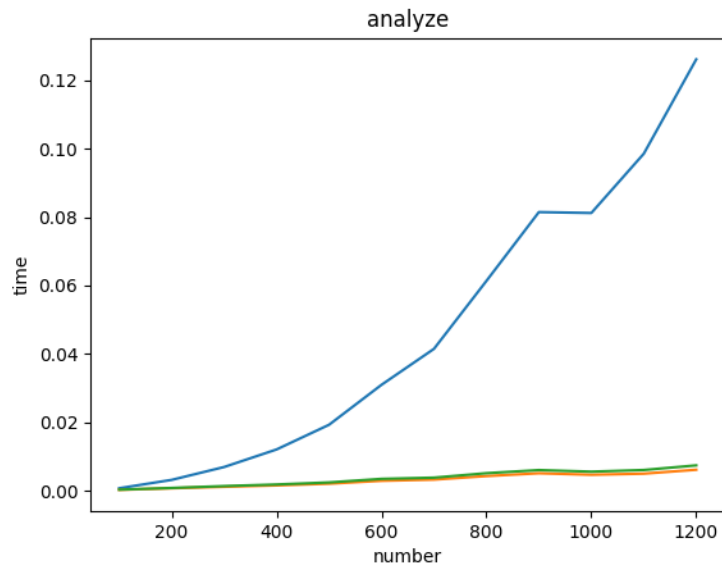


FIG-2

Scale :- y-axis: 1 unit = 0.002 sec

x-axis: 1 unit =500 unit

Here the list taken is sorted in ascending order. The time taken in normal quick sort is n^2 so the graph for it also takes the highest time. It can be seen that quick sort with median takes more time than randomised quick sort.

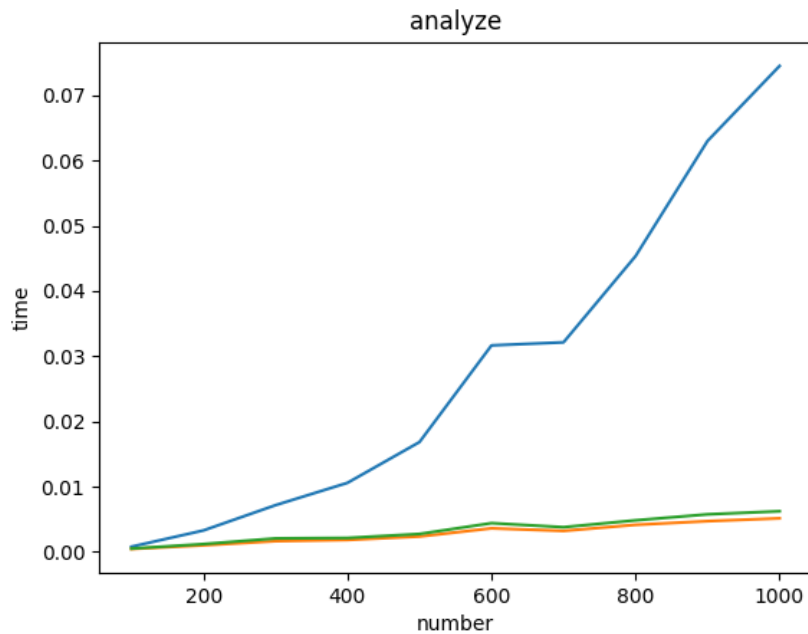


FIG-3

Scale :- y-axis: 1 unit = 0.01 sec

x-axis: 1 unit =200 unit

Here the list taken is sorted in descending order.The time taken in normal quick sort is n^2 so the graph for it also takes the highest time.It can be seen that quick sort with median takes more time than randomised quick sort.

A small observation her is that the number of recursive calls is more in case of sorted in deciding order than ascending order.The recursion depth was reached in 1200 elements in case of sorted in ascending order.

Table to show mean execution time:

S.No	No of elements	Runtime quick sort	Runtime quick sort(Random)	Runtime quick sort(median)
0	100	0.00019845	0.000371641	0.000416461
1	200	0.000400623	0.000853928	0.000957801
2	300	0.000653742	0.00127231	0.00149785
3	400	0.00095724	0.0019047	0.00227594
4	500	0.00157549	0.00301957	0.0037073
5	600	0.0016047	0.00315517	0.00373316
6	700	0.0018885	0.00355344	0.00417583
7	800	0.00228969	0.0042187	0.00485149
8	900	0.00279818	0.00507468	0.00613655
9	1000	0.00298869	0.00542405	0.00644215
10	1100	0.00309683	0.0056823	0.00691698
11	1200	0.00340718	0.00636524	0.00760343

Table-1

The table shows the mean execution time for all types of sorts on a randomly generated array.It can be observed that the mean execution in quick sort using median is greater than other two sorts.

Analysis

It can be seen from **FIG-1** that time taken for quick sort by finding the median is much more than the other two sorts. Than what is need to go through the tedious process of finding the median. It can be seen from **FIG-2** that taken in median quick sort is much less than normal quick sort. Finding the median and using it as an pivot gives the advantage of creating good partitions. When the list is already sorted normal quick sort creates a partition in which one partition there are all the elements and the other has zero as the pivot is the last element.

As seen from **FIG-1** randomised quick sort also takes more time than normal quick sort on a randomly generated list but when the list is sorted than normal quick sort takes much more time.

From the analysis it is quite clear that:

- Quick sort by finding the median is the best case.
- Quick sort by random pivot is average case
- Normal Quick sort represents the worst case.

REPORT - 6

Knapsack

Objective:

To solve the fractional knapsack problem using the Greedy approach.

Input :

A weight array and a value array of n items is given as input along with the maximum capacity of knapsack.

Output :

The maximum profit value and an array of items along with the fraction of each item to be included in the knapsack is given as output

Procedure :

Knapsack-Algorithm

Let wt be the weight array, val be the value array, m be the maximum capacity of knapsack

```
for i in sorted(ratio.items(), key = lambda x : x[1],reverse=True):
    j=list(i[0])
    if(j[1]<=cu_cap):
        cu_cap=cu_cap-j[1]
        res=res+j[0]
        s[j[1]]=[1]
    else:
        res=res+j[0]*(cu_cap/j[1])
        s[j[1]]=[cu_cap/j[1]]
        break
```

Explanation:

In this algorithm, the ratio of the value to the weight for each item is calculated. And items are sorted in decreasing order of the ratio. Then each item is picked from the sorted array and if the weight of the items is less than the capacity of the bag, the whole item is included in the bag and value is added to the profit. The weight is subtracted from the capacity. And if the weight exceeds the capacity, the ratio of capacity to the weight is calculated and only a fraction of item equal to this ratio is added.

Program:

```
n=int(input())
ratio={}
for i in range(n):
    val=int(input())
    weight=int(input())
    ratio[val,weight]=[val/weight]
    #print(val)
print(ratio)
res=0
s={}
gu_cap=int(input())
cu_cap=gu_cap
for i in sorted(ratio.items(), key = lambda x : x[1],reverse=True):
    j=list(i[0])
    if(j[1]<=cu_cap):
        cu_cap=cu_cap-j[1]
        res=res+j[0]
        s[j[1]]=1
    else:
        res=res+j[0]*(cu_cap/j[1])
        s[j[1]]=cu_cap/j[1]
        break
print(res)
print(s)
```

Explanation :

The above program shows the implementation of knapsack using greedy technique. It uses dictionaries in which the key is both profit and weight. Their value is the ratio

profit/weight. It sorts the dictionary in descending order according to its value i.e the ratio. Then the knapsack algorithm is implemented in order to find the desired result.

Output :

Let the weight array be, $wt=[18,15,10]$ The value array, $val=[25,24,15]$. The maximum capacity of knapsack, $m=20$

Running the above program, we get the following solution.

$ratio=\{(25, 18): [1.3888888888888888], (24, 15): [1.6], (15, 10): [1.5]\}$

We use it to find the maximum profit generated :

$\{15: [1], 10: [0.5]\}$

It shows the weight with key as the ratio taken.

Profit=31.5

The maximum profit generated by including the above items in the knapsack.

Analysis:

It can be observed from the solution that the whole items are included in the knapsack followed by fraction of items and sometimes some items are not even included.

The items maximum ratio are put as whole as they have more value and have a higher value. Thus, greedy looks at the best solution at an instance and analyses the item with maximum profit and lesser weight and then include in the knapsack.

REPORT 7

Prims Algorithm

Objective :

To find the minimum spanning tree of a weighted undirected graph using Prim's algorithm.

Input:

The input to the algorithm are the edges and the weights of a graph.

Output:

The algorithm returns the edges that constitute the MST and the cost of the tree

Procedure :

Algorithm Prims :

```
int spanningTree(int V, int E, vector<vector<int>> &graph){
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++) key[i] = INT_MAX, mstSet[i] = false;
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet, V);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    return costofMST(parent, graph, V);
}
```

Explanation:

Function `spanningTree(int V, int E, vector<vector<int>> &graph):`

In this algorithm, the graph object formed by providing it with edges and weights . It is used to acquire the shortest edge from the graph . They are appended to the list *edges* and it's cost is added to *total_cost*.

These vertices are then used to determine the nearby vertices to k and l . After this step the nearby list is updated every single time a new edge is added to *edges*. In the end, the list of *edges* of the MST and the cost of the tree is returned.

Program:

```
#include <iostream>
using namespace std;
int spanningTree(int V,int E,vector<vector<int> > graph);

int main()
{
    int t;
    cin>>t;
    while(t-->0)
    {
        int V,E;
        cin>>V>>E;
        vector< vector<int> > graph(V,vector<int>(V,INT_MAX));

        while(E-->0)
        {
            int u,v,w;
            cin>>u>>v>>w;
            u--,v--;
            graph[u][v] = w;
            graph[v][u] = w;
        }

        cout<<spanningTree(V,E,graph)<<endl;
    }
    return 0;
}

int spanningTree(int V,int E,vector<vector<int> > graph)
{
    int s=INT_MAX;
    int h,r;
    int minc=0;
    int near[V];
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(i==j){
                graph[i][j]=INT_MAX;
                continue;
            }
            if(graph[i][j]<s){
                s=graph[i][j];
                h=i;
                r=j;
            }
        }
    }
    // cout<<h<<r<<" ";
    minc=minc+s;
    for(int i=0;i<V;i++){
        if(graph[i][h]<graph[i][r]){
            near[i]=h;
        }
        else
```



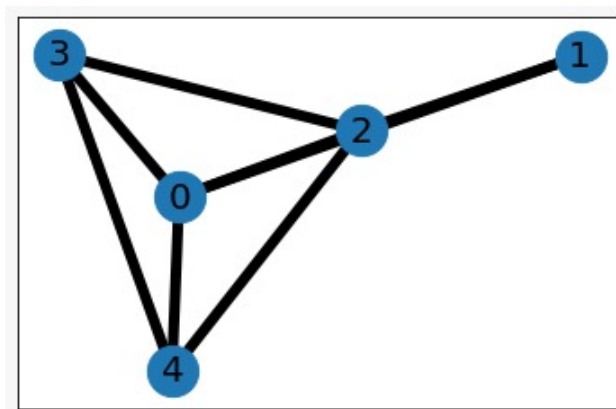
```

    near[i]=r;
}
// cout<<near[1];
near[h]=-1;
near[r]=-1;
for(int p=1;p<(V-1);p++){
    s=INT_MAX;
    for(int i=0;i<V;i++){
        if(near[i]!=-1 && i!=near[i]){
            if(graph[i][near[i]]<s){
                s=graph[i][near[i]];
                h=i;
                r=near[i];
            }
        }
    }
    minc=minc+s;
    near[h]=-1;
    for(int k=0;k<V;k++){
        if(near[k] !=-1 ){
            if(graph[k][near[k]]>graph[k][h])
                near[k]=h;
        }
    }
}
return minc;
}

```

Output:

The graph for which output is shown below:



```
1
58
1 2 1 5 1 19 1 3 17 3 5 6 4 3 6 5 4 2 3 2 16 1 4 16
```

25

The first image represents the input and second represents minimum spanning tree weight

REPORT 8

Kruskal Algorithm

Objective:

Implementation of a Kruskals's Algorithm to Create a minimum spanning tree for a graph. A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Input :

The algorithm takes total 3 parameters as input:

1. Number of vertices.
2. Edge adjacency matrix.
3. Cost adjacency matrix.

Procedure :

Kruskal Algorithm

```
int f(int i, int p[], int r[]) {  
    while(i!=p[i]) {  
        i = p[i];  
    }  
    return i;  
}
```

```
void u( int a, int b, int p[], int r[]) {  
    int i = f(a, p, r);
```

```

int j = f(b, p, r);

if(i==j) {
    return;
}

if(r[i]>=r[j]) {
    r[i]++;
    p[j] = p[i];
} else {
    r[j]++;
    p[i] = p[j];
}
return;
}

```

Explanation:

Sort all the edges in decreasing order of their weights then pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If no cycle the include edge. Repeat it for (V-1) steps.

Program:

```

int f(int i, int p[], int r[]) {

    while(i!=p[i]) {
        i = p[i];
    }
    return i;
}

void u( int a, int b, int p[], int r[]) {

    int i = f(a, p, r);
    int j = f(b, p, r);

    if(i==j) {
        return;
    }

    if(r[i]>=r[j]) {
        r[i]++;
        p[j] = p[i];
    } else {
        r[j]++;
        p[i] = p[j];
    }
    return;
}

```

```

}

int k(vector<pair<int,int>> adj[], int x, int y) {
    pair< int, pair<int,int> > w[2*x];
    int parent[y+1], r[y+1];
    for(int i = 1; i <= y; i++){
        parent[i] = i;
        r[i] = 1;
    }

    int t=0;
    for(int i=1; i<=n; i++) {
        for(int j=0; j<adj[i].size(); j++) {
            w[t++] = make_pair(adj[i][j].second, make_pair(i, adj[i][j].first));
        }
    }
    sort(w, w+t);

    int mst = 0;

    for(int i=0; i<t; i++) {
        int u = w[i].second.first;
        int v = w[i].second.second;
        int w1 = weight[i].first;
        if(f(u, parent, r) != f(v, parent, r)) {
            u(u, v, parent, r);
            mst+=w1;
        }
    }

    return mst;
}

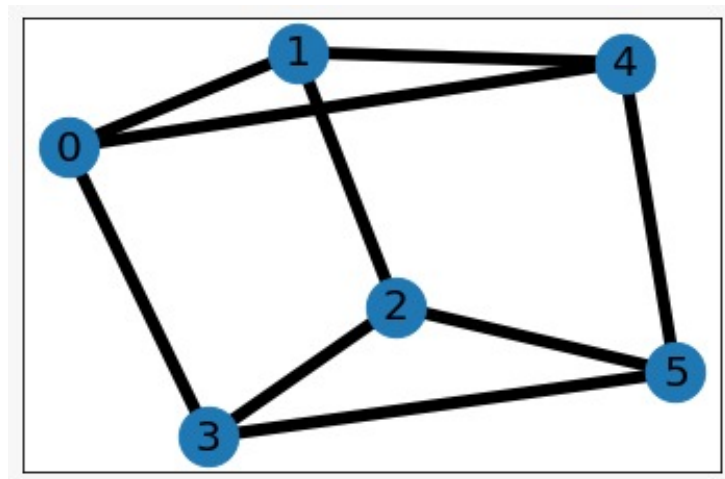
```

Explanation :

The program takes a number of vertices, edges, and weights as input from the user.

Output:

The graph for which output is shown below:



The output is:

```
1
6 12
1 4 10 1 2 20 2 3 16 1 4 16 4 3 20 5 2 12 1 2 15 6 5 9 5 1 6 1 5 8 3 6 16 6 4 8
```

53

The first image shows the input and second shows he weight.

REPORT 9

Sum Of Subsets

Objective:

Implementation of a Backtracking Algorithm to solve the Sum Of Subset Problem. Sum of Subset problem is to find the subsets of elements that are selected from a given set whose sum adds up to a given number m.

Input :

The algorithm takes total 5 parameters as input:

1. Array of n elements.
2. Index of the current element chosen.
3. Total sum generated from previous elements before current element in array.
4. Total sum of elements after current element including current element in array.
5. State array to keep track of elements chosen in the current set.

Procedure :

Algorithm sum of subsets:

```
void sumofsub(int s,int k,int r,int m,int n){
    x[k]=1;
    if((s+w[k])==m)
    {
        for(int i=0;i<n;i++){
            if(x[i]==1)
                cout<<w[i]<<" ";
        }
        cout<<"\n";
    }
    x[k]=0;
```

```

    }
    else if(s+w[k]+w[(k+1)]<=m){
        sumofsub((s+w[k]),(k+1),(r-w[k]),m,n);
    }
    if((s+r-w[k])>=m && (s+w[(k+1)]<=m)){
        x[k]=0;
        sumofsub(s,(k+1),(r-w[k]),m,n);
    }
}
}

```

Explanation:

Function void sumofsub(int s,int k,int r,int m,int n)

The function takes the original array, index of current element, sum of previous selected elements, total sum possible to increase the current sum, state array, and an array to store possible solutions. The algorithm checks if the index inputted is valid. Then it checks if sum till previous elements and the current element is equal to the desired value. If this is true then a solution is found and appended to the final answer array, else if this is false then it checks if the sum is less than the desired value. If this is true then the algorithm recursively tries to include the next element, otherwise it will exclude the current element and try with the next element.

Program:

```

#include<iostream>
using namespace std;
int w[7]={5,10,12,13,15,18};
int x[7];
void sumofsub(int s,int k,int r,int m,int n){
    x[k]=1;
    if((s+w[k])==m)
    {
        for(int i=0;i<n;i++){
            if(x[i]==1)
                cout<<w[i]<<" ";
        }
        cout<<"\n";
        // cout<<"aa";
    }
    x[k]=0;
}

```



```

else if(s+w[k]+w[(k+1)]<=m){
    sumofsub((s+w[k]),(k+1),(r-w[k]),m,n);
}
if((s+r-w[k])>=m && (s+w[(k+1)]<=m)){
    x[k]=0;
    sumofsub(s,(k+1),(r-w[k]),m,n);
//    cout<<"L1";
}
//cout<<"k";
}
int main(){
    int s=0;
    int k=0;
    int r=0;
    int n=7;
    for(int i=0;i<n;i++){
        r=r+w[i];
    }
    int m=30;
    sumofsub(s,k,r,m,n);
}

```

Explanation :

The program takes the array and the desired sum as input, and whether the user wants to print all possible solutions for the problem or just one. The program first sort the array and creates a state array to store the elements included in the subset. This state array is initialized to 0 and if the element is included then 1 is stored at the index of the included element. Then function `sumofsubsets()` which find all valid solutions and store the states in an array. After this the array is used to print all the valid solutions

Output :

```

5 10 12 13 15 18
30
5 10 15
5 12 13
12 18

```

```

5 10 12 13 15 18
47
no subset

```

In the output it can be seen that if the required is 30 then it generates 3 possible subsets and when the required sum is 47 then no subset is generated

REPORT 10

M-Coloring

Objective :

To solve the m-coloring problem using backtracking approach.

Input :

The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. Each test case consists of four lines. The first line of each test case contains an integer N denoting the number of vertices. The second line of each test case contains an integer M denoting the number of colors available. The third line of each test case contains an integer E denoting the number of edges available. The fourth line of each test case contains E pairs of space separated integers denoting the edges between vertices.

Procedure :

Algorithm M-Coloring

```
void nextvalue(int k,int n,int m,int x[],bool graph[101][101]){
    int j;
    while(1){
        x[k]=(x[k]+1)%(m+1);
        if(x[k]==0){

            return ;
        }
        for(j=0;j<n;j++){
            if(graph[k][j] !=0 && x[k]==x[j])
                break;
        }
        if(j==n){
            return ;
        }
    }
}

bool mcoloring(int k,int n,int m,int x[],bool graph[101][101]){
    while(1){
        nextvalue(k,n,m,x,graph);
        // cout<<x[k]<<" ";
        if(x[k]==0){
```

```

        return 0;
    }
    if(k==(n-1)){
        return 1;
    }
    else {
        //cout<<k;
        return mcoloring((k+1),n,m,x,graph);
    }
}
}

```

Explanation:

1> Function **mcoloring(int k,int n,int m,int x[],bool graph[101][101]):**

The function takes the adjacency matrix as input. It calls the `nextvalue()` function in order to find the colour number for each edge one by one. It ends when it is able to find the required colour for the last vertex or when the number of edges exceed the value of `m`.

2> Function **nextvalue(int k,int n,int m,int x[],bool graph[101][101]):**

The function takes the adjacency matrix as input. It is used to find the required colour for each vertex so that it meets the condition that no two adjacent edges have the same color.

Program:

```

#include<iostream>
using namespace std;
int x[4]={0,0,0,0};
int g[4][4]= {{0, 1, 1, 1},
               {1, 0, 1, 0},
               {1, 1, 0, 1},
               {1, 0, 1, 0}};

void nextvalue(int k,int n,int m){
    int j;
    while(1){
        x[k]=(x[k]+1)%(m+1);
        if(x[k]==0)
            return ;
        for(j=0;j<n;j++){
            if(g[k][j] !=0 && x[k]==x[j])
                break;
        }
        if(j==n)
    }
}

```

```

        return ;
    }
}
void mcoloring(int k,int n,int m){
    while(1){
        nextvalue(k,n,m);
        if(x[k]==0){
            cout<<-1;
            return;
        }
        if(k==(n-1)){
            for(int i=0;i<n;i++){
                cout<<x[i]<<" ";
            }
            cout<<"\n";
            exit(0);
        }
        else {
            //=cout<<k;
            mcoloring((k+1),n,m);
        }
    }
}
int main(){
    int k=0;
    int n=4;
    int m;
    cin>>m;

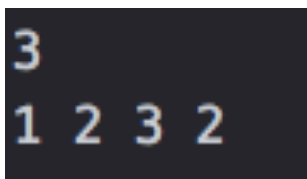
    mcoloring(k,n,m);
}

```

Explanation :

The program takes the adjacency matrix and the number of colours as input .It outputs the colour number of each vertex if it is possible which depends on the value of m.If it is not possible then it prints that.

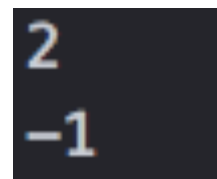
Output :



```

3
1 2 3 2

```



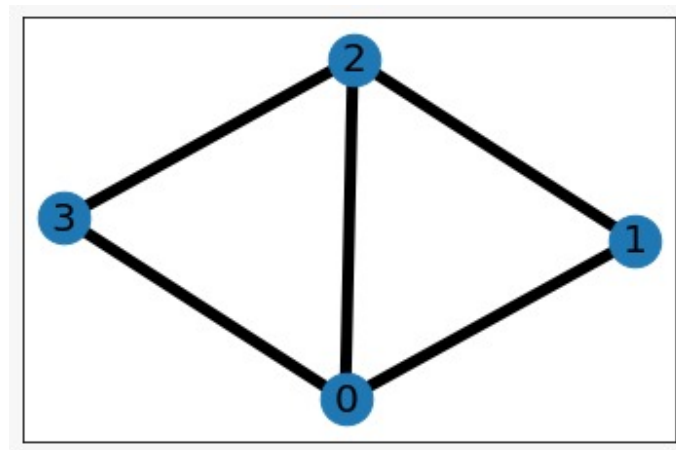
```

2
-1

```

These are the screenshots of the output when the number of colours are 3 and 2 respectively .In the first case it represents the number of colours and if the number of colours are 2 then it is not possible.

The graph for which these outputs are found is:



REPORT 11

N-Queens Problem

Objective :

To solve the n-queens problem using backtracking approach.

Input :

The algorithm takes total 2 parameters as input:

1> $N \times N$ array as chessboard to locate the position of the queens placed.

2> Dimension of the chessboard (N)

Procedure :

Algorithm N-queens

Let n be a side of the chess board and the number of queens.

```
int placed(int k,int i){  
  
    for(int j=0;j<(k);j++){  
        if((x[j]==(i)) || (abs(x[j]-(i))==abs(j-k))){  
            return 0;  
        }  
    }  
    return 1;  
}  
  
void nqueen(int k,int n){  
    for(int i=0;i<n;i++){  
        if(placed(k,i)){  
            x[k]=i;  
            if(k==(n-1)){  
                for(int u=0;u<n;u++){  
                    cout<<x[u]<<" ";  
                }  
                cout<<"\n";  
            }  
        }  
    }  
}
```

```

        else
            nqueen((k+1),n);
    }
}

```

Explanation:

1> Function **n_queens(in k, int n):**

The function takes chessboard matrix, current x coordinate of queen, dimension of chess board and an extra parameter to print one or all possible solutions for the problem. For ith Queen, we iterate through all possible y coordinates and check whether the y position is valid by using function **placed()**. If the position is possible then the function calls itself recursively for next queen. This will loop till either all possible y positions are not valid for any queen. In that case function will return False. The other case is when we find position for the nth queen. After that depending upon the optional parameter **all_possible_solutions**, algorithm will either find more solutions and store in ans array or it will return True.

2> Function **def placed(int k, int i):**

The function return False if there exist some queen before the current queen such that it attacks the current queen diagonally diagonally or vertically, otherwise this will return True.

Program:

```

#include<iostream>
using namespace std;
int x[4]={0,0,0,0};
int placed(int k,int i){
    for(int j=0;j<(k);j++){
        if((x[j]==(i)) || (abs(x[j]-(i))==abs(j-k))){
            return 0;
        }
    }
    return 1;
}
void nqueen(int k,int n){

```



```

for(int i=0;i<n;i++){
    if(placed(k,i)){
        x[k]=i;
        if(k==(n-1)){
            for(int u=0;u<n;u++){
                cout<<x[u]<<" ";
            }
            cout<<"\n";
        }
        else
            nqueen((k+1),n);
    }
}
}
int main(){
    int n,k;
    k=0;
    cin>>n;
    nqueen(k,n);
}

```

Explanation :

The program takes the dimension of the chessboard as the input. It outputs all the possible column number for each row in which the queen is to be placed in order to meet the conditions for n-queens problem.

QWERTY1

4			
1	3	0	2
2	0	3	1

5				
0	2	4	1	3
0	3	1	4	2
1	3	0	2	4
1	4	2	0	3
2	0	3	1	4
2	4	1	3	0
3	0	2	4	1
3	1	4	2	0
4	1	3	0	2
4	2	0	3	1

These are the screenshots of the output when the dimensions of board are 4×4 and 5×5 respectively .These represent the column of each row of the various cases which are possible.

REPORT 12

Jolly Number Problem

Objective :

To solve the Jolly number problem.

Input :

The algorithm takes total 2 parameters as input:

- 1> The number of elements n.
- 2> The array consisting of n elements.

Procedure :

Algorithm Jolly Number

```
string check(int arr[],int n){
    unordered_set <int> s;
    for(int i=0;i<(n-1);i++){
        int j=abs(arr[(i+1)]-arr[i]);
        s.insert(j);
        if(j<1 || j>(n-1)){
            return "Not jolly";
        }
    }
    if(s.size()==n-1)
        return "Jolly";
    else
        return "Not jolly";
}
```

Explanation:

Function string check(in k, int n):

In this function it is checked that every difference should lie within the range of 1 to $n-1$.

Program:

```
#include<iostream>
#include<unordered_set>
using namespace std;
string check(int arr[],int n){
    unordered_set <int> s;
    for(int i=0;i<(n-1);i++){
        int j=abs(arr[(i+1)]-arr[i]);
        s.insert(j);
        if(j<1 || j>(n-1)){
            return "Not jolly";
        }
    }
    if(s.size()==n-1)
        return "Jolly";
    else
        return "Not jolly";
}
int main(){
    int t;
    cin>>t;
    while(t--){
        int n;
        cin>>n;
        int arr[n];
        for(int i=0;i<n;i++){
            cin>>arr[i];
        }
        cout<<check(arr,n);
    }
}
```

Explanation :

It maintains an unordered set to store set of absolute difference of successive elements.

a) If absolute difference between two elements is more than $n-1$ or 0, return false.

b) If an absolute difference repeated, then all absolute differences from 1 to $n-1$ can't be present then set size will reduce, return false.

Output :

```
2
4 1 4 2 3
Jolly
5 1 4 2 -1 6
Not jolly
```

It can be seen in the output that first set of integers meet the condition of being a jolly number and the second doesn't.

REPORT - 13

WERTYU

Objective:

To implement a solution for the WERTYU problem. The task in the problem is to decode a string in a manner that for every character entered, the output should be the character at the immediate left in a keyboard.

Input :

The number of test cases and the string are the input in this algorithm

Procedure :

Algorithm WERTYU

```
c = "1234567890-=QWERTYUIOP[]\ASDFGHJKL;'ZXCVBNM,./"
ans = str()
for ch in string:
    index = -1
    if c.find(ch) != -1:
        index = c.find(ch)
        ans += c[index-1]
    else:
        ans += ch
```

Explanation:

In the Algorithm, for each character in the string inputted by the user is searched through the template. If the character is present in the template

then the character at immediate left is added to the result string. If the character is not present then it is added as it is in the resultant string.

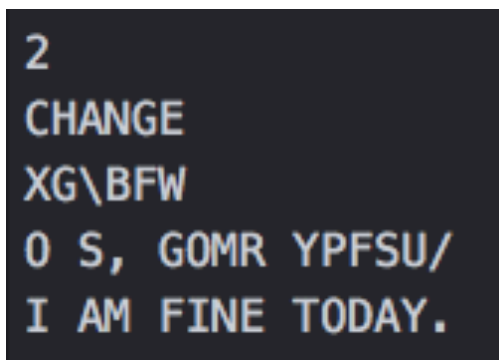
Program:

```
i = int(input())
for _ in range(1, i+1):
    string = input()
    c = "1234567890-=QWERTYUIOP[]\ASDFGHJKL;'ZXCVBNM,./"
    ans = str()
    for ch in string:
        index = -1
        if c.find(ch) != -1:
            index = c.find(ch)
            ans += c[index-1]
        else:
            ans += ch
    print(ans)
```

Explanation :

The program takes the number of test cases as input. Then for the number of test cases the program asks the user to enter the string, and the changed string is output in the next line.

Output :

A screenshot of a terminal window with a dark background and light-colored text. The output shows the results of the program for 2 test cases. The first test case input is 'CHANGE' and the output is 'XG\BFW'. The second test case input is 'O S, GOMR YPFSU/I AM FINE TODAY.' and the output is '0 S, GOMR YPFSU/I AM FINE TODAY.'.

```
2
CHANGE
XG\BFW
O S, GOMR YPFSU/
I AM FINE TODAY.
```