Experiment No: 1

Implement a function for each of following problems and count the number of steps executed/time taken by each function on various inputs (100 to 500) and write equation for the growth rate of each function. Also draw a comparative chart of number of input versus steps executed/time taken. In each of the following function N will be passed by user.

1. To calculate sum of 1 to N numbers using loop.
2. To calculate sum of 1 to N numbers using equation.
3. To calculate sum of 1 to N numbers using recursion.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1

Objectives: (a) Compare performance of various algorithms

(b) Judge best algorithm in terms of growth rate or steps executed

Equipment/Instruments: Computer System, Any C language editor–

Theory:

1. **Below are the steps to calculate sum of 1 to N numbers using loop**
    1. Take an input value for N.
    2. Initialize a variable sum to zero.
    3. Start a loop that iterates from i=1 to i=N.
    4. In each iteration, add the value of i to the sum variable.
    5. After the loop completes, output the value of sum.

2. **Below are the steps to calculate sum of 1 to N numbers using equation**
    1. Take an input value for N.
    2. Calculate sum as N*(N+1)/2.
    3. Output the value of sum.

3. **Below are the steps to calculate sum of 1 to N numbers using recursion**
    1. Take an input value for N.
    2. Define a recursive function sum(n) that takes an integer argument n and returns the sum of 1 to n.
    3. In the sum(n) function, check if n equals 1. If it does, return 1 (the base case).
    4. Otherwise, return the sum of n and the result of calling sum(n-1).
    5. Output the result.

Implement three functions based on above steps and calculate the number of steps executed by each functions on various inputs ranging from 100 to 500. Take a counter variable to calculate the number of steps and increment it for each statement in the function.

Observations:

- **Program:**

```
import java.util.*;
import java.lang.*;

public class Demo
{
   public int loop(int n)
   {
      int sum=0,counter=0;
      for(int i=1;i<=n;i++)
      {
         sum=sum+i;
         counter++;
      }
      System.out.println("Steps of loop:"+counter);
      return sum;
   }
   public int equation(int n)
   {
      int sum=0,counter=0;
      sum=(n*(n+1))/2;
      counter++;
      System.out.println("Steps of equation:"+counter);
      return sum;
   }
   int counter=0;
   public int recursion(int n)
```

```
    {
        int  sum=0;
        counter++;
        if(n<=1)
        {
            return n;
        }
        sum=n+recursion(n-1);
        return sum;
    }

    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        Demo d=new Demo();
        System.out.println("Enter a number: ");
        int n=sc.nextInt();
        System.out.println("Sum using loop is: "+ d.loop(n));
        System.out.println("Sum using equation is: "+ d.equation(n));
        System.out.println("Sum using recursion is: "+ d.recursion(n));
        System.out.println("Steps of recursion:"+d.counter);
}
}
```

Loop and Recursion Methods:

Both of these methods exhibit a direct correlation between the number of steps taken and the size of the input. For instance, when calculating the sum for 100, the methods might require around 100 steps. If the input increases to 200, the methods would then necessitate approximately 200 steps. This relationship is linear and follows a straightforward one-to-one pattern.

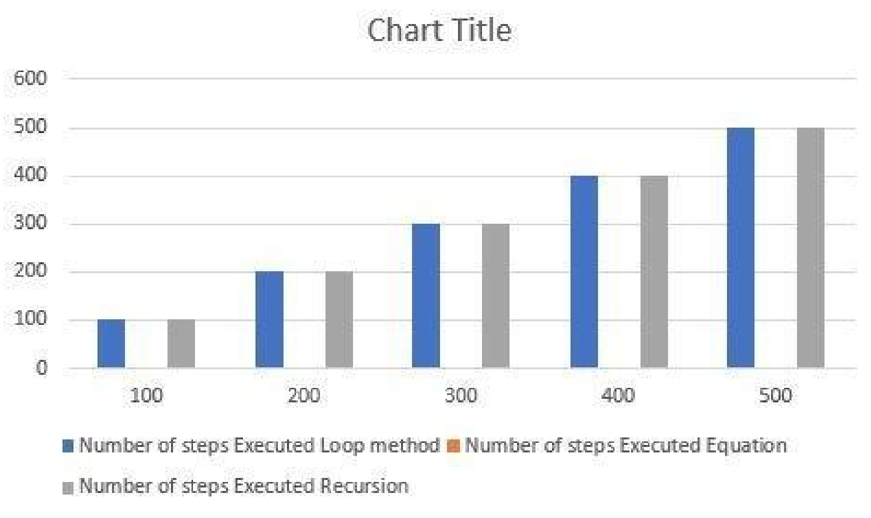
Equation Method:

This approach is incredibly efficient. Irrespective of the size of the input number, it consistently concludes its calculation within just one step. It's similar to possessing a convenient quick route.

Result: Complete the below table based on your implementation of functions and steps executed by each function.

Inputs	Number of Steps Executed		
	Loop method	**Equations**	**Recursion**
100	100	1	100
200	200	1	200
300	300	1	300
400	400	1	400
500	500	1	500
Equation	$f(N) = N$	$f(N) = 1$	$f(N) = N$

Chart:



Conclusion:

The loop-based and recursive approaches exhibited linear growth in the number of steps with increasing input size, while the equation-based method displayed a constant number of steps, providing its efficiency.

Quiz:

1. **What is the meaning of constant growth rate of an algorithm?**
   Constant growth rate of an algorithm implies that its execution time remains consistent as input size changes. This behavior, often seen in O(1) time complexity algorithms, ensures stable performance, making them efficient choices for various inputs.

2. **If one algorithm has a growth rate of n^2 and second algorithm has a growth rate of nthen which algorithm execute faster? Why?**
   If one algorithm has a growth rate of O(n^2) and the second algorithm has a growth rate of O(n), the algorithm with the growth rate of O(n) will execute faster for larger input sizes. This is because the growth rate of an algorithm indicates how its runtime increases as the input size (n) increases.

References used by the students:

"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
"Fundamentals of Algorithms" by E. Horowitz et al.

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 2

Write user defined functions for the following sorting methods and compare their performance by steps executed/time taken for execution on various inputs (1000 to 5000) ofrandom nature, ascending order and descending order sorted data.Also, draw a comparative chart of number of inputs versus steps executed/time taken for each cases (random, ascending, and descending).

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1, CO2

Objectives: (a) Compare performance of various algorithms.
(b) Judge best sorting algorithm on sorted and random inputs in terms of growth rate/time complexity.
(c) Derive time complexity from steps count on various inputs.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Write main function to use above sorting functions and calculate the number of steps executed by each functions on various inputs ranging from 1000 to 5000. Take a counter variable to calculate the number of steps and increment it for each statement in the function.

1. **Selection Sort**

```
import java.util.*;
import java.lang.*;
public class selection
{
public static void main(String args[])
{
int a[]=new int[1000];
Random r=new Random();
for(int i=0;i<1000;i++)
{
a[i]=r.nextInt(10000);}
long start=System.currentTimeMillis();
for(int i=0;i<a.length;i++)
```

```
{
int min=i;
for(int j=i+1;j<a.length;j++)
{
if(a[j]<a[min])
{
min=j;
}
}
int temp=a[min];
a[min]=a[i];
a[i]=temp;
System.out.print(a[i] + " ");
}
long end=System.currentTimeMillis();
long time=end-start;
System.out.println("\nTime taken for
execution is:"+time);
}
```

Output:-

```
E:\Files_sem-5>javac selection.java

E:\Files_sem-5>java selection
1 9 41 46 51 61 113 123 132 141 144 146 147 186 189 199 229 249 265 266 267 269 269 328 335 376 377 378 381 396 407 417 420 433 439 444 452 463 478 486 493 498 500 512
516 523 527 528 532 544 546 571 589 611 618 671 683 717 723 734 747 758 771 798 812 814 816 824 876 884 891 903 916 924 948 957 962 968 993 1005 1008 1012 1031 1065 110
3 1115 1116 1128 1134 1135 1139 1151 1152 1154 1154 1163 1187 1204 1214 1222 1231 1232 1236 1245 1272 1276 1285 1300 1305 1312 1313 1332 1346 1382 1385 1386 1400 1408 1
413 1442 1442 1478 1498 1504 1505 1507 1540 1546 1553 1572 1580 1593 1618 1647 1650 1665 1670 1683 1736 1765 1793 1796 1796 1798 1803 1809 1822 1829 1836 1839 1842 1859
 1865 1877 1878 1879 1888 1888 1920 1930 1935 1937 1946 1952 1955 1976 2024 2039 2044 2053 2112 2118 2124 2125 2152 2167 2169 2172 2181 2184 2191 2193 2195 2218 2221 22
35 2242 2253 2259 2261 2265 2275 2284 2286 2298 2310 2318 2321 2339 2381 2382 2398 2400 2400 2411 2422 2428 2428 2428 2457 2468 2471 2481 2491 2504 2517 2521 2532 2549
2556 2565 2579 2589 2601 2613 2617 2635 2636 2637 2648 2651 2690 2703 2704 2724 2744 2760 2779 2782 2783 2787 2797 2807 2817 2818 2822 2825 2825 2836 2841 2849 2856 286
0 2885 2892 2893 2916 2918 2925 2940 2959 2962 2967 2980 2982 2983 2989 2989 2993 3060 3061 3061 3065 3066 3074 3077 3099 3104 3150 3164 3172 3177 3189 3224 3243 3244 3
246 3269 3295 3295 3296 3339 3374 3378 3407 3411 3412 3441 3446 3450 3482 3499 3507 3510 3518 3521 3558 3564 3568 3579 3596 3599 3612 3612 3618 3619 3622 3625 3628 3635
 3645 3648 3656 3672 3677 3701 3731 3747 3753 3756 3780 3800 3809 3828 3828 3837 3851 3855 3856 3893 3916 3927 3932 3933 3942 3943 3950 3954 3955 3981 3997 3997 3999 40
04 4005 4007 4008 4013 4023 4040 4048 4061 4070 4071 4080 4089 4097 4100 4106 4125 4132 4138 4139 4145 4157 4161 4185 4188 4209 4218 4219 4220 4224 4226 4233 4236 4247
4253 4291 4303 4303 4304 4319 4319 4325 4327 4328 4347 4350 4351 4352 4353 4366 4374 4398 4400 4408 4424 4434 4434 4436 4436 4451 4458 4467 4477 4485 4498 4515 4525 453
5 4552 4565 4588 4594 4595 4603 4607 4614 4625 4628 4629 4637 4639 4665 4682 4688 4705 4708 4720 4720 4733 4736 4739 4742 4751 4755 4760 4767 4784 4796 4801 4823 4826 4
827 4830 4845 4857 4858 4866 4896 4903 4906 4925 4945 4950 4954 4964 4975 4978 5022 5024 5036 5041 5058 5058 5073 5074 5092 5097 5106 5135 5140 5140 5141 5148 5151 5168
 5188 5210 5218 5241 5244 5246 5247 5253 5272 5283 5284 5288 5302 5310 5334 5337 5345 5354 5379 5389 5406 5425 5432 5436 5465 5470 5470 5473 5482 5518 5545 5559 5568 55
74 5609 5609 5615 5618 5621 5639 5642 5653 5699 5700 5718 5725 5733 5735 5743 5750 5756 5767 5774 5783 5816 5821 5827 5840 5845 5851 5853 5857 5881 5928 5946 5946 5948
5951 5962 5978 6000 6015 6021 6027 6031 6033 6044 6047 6048 6062 6081 6087 6110 6110 6111 6122 6128 6145 6153 6162 6164 6176 6181 6187 6205 6206 6208 6217 6233 6233 623
8 6247 6249 6251 6252 6252 6253 6262 6276 6285 6287 6287 6292 6301 6311 6314 6341 6368 6373 6379 6388 6390 6402 6403 6404 6413 6416 6417 6426 6448 6478 6478 6489 6502 6
550 6605 6617 6622 6623 6626 6631 6631 6644 6653 6655 6662 6663 6666 6677 6691 6702 6712 6726 6735 6741 6754 6758 6777 6788 6802 6804 6819 6823 6826 6856 6857 6857 6859
 6864 6879 6882 6882 6927 6937 6945 6949 6962 6965 6967 6971 6972 6984 6989 6998 7003 7008 7015 7019 7020 7031 7042 7045 7048 7068 7071 7072 7078 7079 7098 7100 7110 71
28 7138 7164 7165 7168 7174 7193 7214 7239 7243 7267 7270 7273 7279 7300 7318 7319 7324 7331 7331 7331 7331 7338 7362 7364 7365 7367 7381 7396 7399 7414 7422 7423 7452
7463 7465 7481 7498 7502 7548 7555 7556 7559 7560 7574 7576 7588 7595 7599 7610 7632 7633 7654 7662 7663 7671 7676 7714 7717 7731 7733 7737 7738 7738 7739 7742 7762 777
2 7775 7775 7782 7783 7797 7801 7807 7822 7826 7836 7839 7845 7851 7861 7865 7886 7889 7893 7897 7900 7900 7911 7911 7913 7915 7929 7973 7984 7986 7997 8000 8016 8027 8
039 8043 8068 8090 8091 8094 8094 8103 8113 8132 8155 8163 8168 8171 8172 8185 8195 8205 8220 8225 8230 8232 8244 8261 8262 8273 8273 8276 8280 8292 8297 8297 8297 8301
 8307 8322 8326 8330 8339 8345 8363 8371 8376 8390 8418 8432 8438 8451 8454 8460 8468 8490 8495 8505 8508 8509 8509 8512 8538 8544 8550 8565 8581 8588 8611 8615 8625 86
26 8642 8658 8667 8670 8670 8676 8695 8707 8725 8727 8734 8747 8754 8758 8759 8785 8792 8822 8829 8858 8874 8885 8891 8895 8896 8916 8923 8941 8954 8957 8968 8969 8969
8984 8990 8993 9010 9012 9020 9026 9042 9075 9076 9098 9144 9149 9178 9182 9186 9191 9200 9200 9215 9247 9247 9256 9258 9259 9260 9275 9285 9291 9294 9315 9318 9325 933
5 9339 9341 9355 9355 9371 9375 9383 9392 9405 9406 9448 9459 9466 9474 9493 9494 9500 9519 9529 9542 9546 9563 9573 9580 9588 9610 9618 9629 9639 9640 9644 9645 9646 9
646 9647 9651 9661 9663 9666 9671 9680 9683 9688 9695 9710 9717 9718 9721 9723 9724 9747 9755 9765 9773 9775 9804 9814 9827 9835 9837 9855 9869 9885 9889 9893 9896 9917
 9925 9925 9941 9945 9949 9968 9973 9978
Time taken for execution is:193
```

2. Bubble Sort

```
import java.util.*;
import java.lang.*;

public class bubble
{
public static void main(String args[])
{
int a[]=new int[1000];
```

```
Random r=new Random();

long start=System.currentTimeMillis();
for(int i=0;i<1000;i++)
{
a[i]=r.nextInt(10000);}

for(int i=0;i<a.length;i++)
{
for(int j=i+1;j<a.length;j++)
{
if(a[i]>a[j])
{
int temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
System.out.print(a[i] + " ");
}

long end=System.currentTimeMillis();
long time=end-start;

System.out.println("\nTime taken for execution is: "+ time);
}
}
```

Output:

```
E:\Files_sem-5>javac bubble.java

E:\Files_sem-5>java bubble
14 28 33 45 51 56 61 78 83 86 87 88 120 122 125 128 130 134 146 149 169 169 172 173 180 180 181 183 191 197 199 199 212 219 228 244 252 264 265 267 289 296 298 300 308
317 324 330 332 339 342 345 345 345 370 372 377 390 420 442 444 450 456 464 504 511 512 516 520 532 550 554 561 578 604 616 630 633 648 687 692 717 726 729 730 743 748
752 765 768 768 770 771 777 779 786 796 825 831 837 838 844 845 857 862 862 866 877 880 890 899 905 933 937 974 979 989 997 1007 1012 1013 1023 1037 1054 1055 1076 1079
 1084 1089 1105 1124 1133 1168 1180 1209 1212 1233 1247 1256 1265 1288 1305 1327 1334 1337 1366 1377 1379 1380 1394 1435 1436 1441 1442 1448 1456 1478 1483 1487 1509 15
09 1514 1522 1524 1527 1534 1563 1570 1579 1614 1614 1617 1619 1623 1625 1631 1649 1657 1661 1681 1681 1690 1695 1699 1701 1703 1707 1722 1727 1731 1732 1737 1758 1763
1769 1782 1788 1796 1805 1810 1815 1818 1829 1829 1840 1849 1849 1852 1860 1860 1876 1876 1884 1895 1917 1919 1922 1936 1946 1971 1973 1979 1997 1998 2000 2005 2021 202
4 2025 2026 2033 2059 2070 2071 2076 2078 2079 2081 2091 2113 2118 2149 2155 2162 2166 2175 2177 2178 2178 2207 2214 2232 2244 2249 2288 2323 2339 2339 2342 2344 2355 2
356 2366 2373 2374 2443 2452 2492 2496 2513 2532 2546 2565 2583 2660 2677 2684 2695 2711 2733 2755 2769 2773 2774 2775 2784 2798 2823 2859 2862 2863 2868 2874 2882 2894
 2903 2915 2916 2923 2924 2925 2928 2940 2949 2955 2957 2963 2989 2993 2997 3008 3018 3036 3062 3075 3081 3090 3092 3100 3148 3153 3156 3167 3169 3171 3175 3180 3193 32
36 3236 3251 3267 3269 3272 3281 3282 3289 3294 3301 3313 3333 3345 3346 3349 3359 3404 3421 3421 3423 3428 3432 3438 3440 3445 3449 3450 3456 3459 3464 3474 3475 3484
3492 3501 3502 3505 3507 3508 3514 3526 3528 3539 3544 3549 3553 3564 3567 3570 3594 3601 3609 3627 3635 3645 3671 3679 3681 3682 3683 3696 3728 3729 3749 3761 3768 377
1 3771 3782 3784 3794 3812 3816 3819 3822 3843 3853 3855 3860 3863 3863 3865 3865 3872 3883 3884 3887 3896 3904 3907 3982 3984 4008 4015 4037 4038 4043 4052 4068 4069 4
074 4082 4093 4101 4107 4108 4119 4122 4125 4129 4140 4143 4144 4155 4169 4170 4185 4198 4200 4201 4212 4228 4239 4255 4265 4309 4309 4321 4327 4336 4339 4357 4358 4373
 4376 4382 4394 4409 4423 4454 4461 4472 4477 4478 4488 4511 4511 4513 4519 4528 4529 4531 4546 4566 4569 4590 4592 4597 4621 4635 4672 4697 4704 4709 4714 4734 4752 47
66 4767 4769 4793 4813 4813 4817 4830 4831 4840 4855 4874 4884 4885 4889 4900 4909 4919 4941 4959 4977 4995 5000 5026 5042 5061 5065 5066 5068 5085 5088 5097 5108 5111
5116 5122 5125 5133 5145 5178 5179 5180 5194 5202 5206 5210 5212 5224 5228 5251 5265 5272 5272 5288 5299 5300 5318 5342 5344 5349 5351 5367 5395 5409 5428 5433 5469 546
9 5472 5474 5483 5484 5492 5495 5498 5498 5506 5509 5510 5511 5526 5541 5542 5576 5590 5610 5638 5646 5653 5660 5689 5693 5696 5715 5750 5754 5770 5771 5800 5807 5846 5
855 5856 5857 5862 5863 5869 5880 5892 5896 5901 5902 5903 5928 5945 5957 5974 5996 6014 6020 6023 6040 6047 6066 6085 6101 6108 6108 6116 6129 6152 6157 6189 6191 6227
 6234 6246 6251 6294 6310 6312 6315 6321 6324 6326 6343 6373 6374 6375 6378 6388 6390 6391 6394 6397 6413 6414 6418 6422 6437 6437 6438 6440 6456 6466 6476 6481 6506 65
19 6526 6532 6561 6566 6566 6591 6647 6655 6683 6685 6692 6701 6718 6752 6757 6778 6801 6815 6821 6828 6836 6855 6864 6865 6867 6871 6873 6880 6901 6929 6934 6948 6956
6959 6967 6973 6998 6999 7001 7020 7021 7030 7043 7044 7045 7057 7065 7070 7072 7105 7114 7118 7134 7160 7161 7166 7180 7200 7205 7223 7224 7231 7235 7238 7239 7244 724
9 7260 7261 7274 7289 7297 7297 7299 7300 7306 7315 7345 7392 7396 7400 7423 7434 7437 7439 7454 7470 7471 7477 7480 7482 7510 7522 7527 7537 7538 7549 7558 7562 7585 7
587 7590 7601 7602 7698 7702 7705 7724 7725 7734 7740 7742 7745 7754 7754 7769 7772 7779 7782 7784 7785 7787 7791 7814 7826 7827 7830 7838 7854 7866 7881 7886 7897 7927
 7931 7933 7941 7941 7947 7953 7958 7959 7962 7963 7968 7976 7980 7987 8021 8030 8034 8041 8047 8060 8066 8075 8076 8076 8084 8096 8109 8121 8131 8147 8155 8170 8170 81
73 8174 8191 8195 8203 8220 8250 8251 8278 8283 8316 8327 8329 8332 8375 8382 8384 8385 8394 8397 8398 8402 8416 8433 8435 8441 8459 8475 8476 8498 8499 8513 8516 8519
8522 8523 8529 8536 8538 8538 8544 8557 8570 8578 8585 8599 8602 8609 8617 8618 8619 8621 8624 8651 8652 8662 8670 8675 8682 8688 8698 8699 8700 8708 8715 8729 8735 874
4 8754 8755 8760 8787 8787 8790 8793 8809 8832 8858 8864 8876 8877 8879 8893 8894 8900 8923 8957 8959 8998 9001 9026 9038 9040 9061 9077 9088 9102 9108 9113 9124 9137 9
138 9152 9158 9174 9185 9196 9198 9203 9219 9236 9269 9280 9333 9347 9348 9376 9381 9439 9450 9461 9472 9486 9494 9507 9518 9523 9528 9544 9550 9575 9587 9626 9649 9650
 9651 9652 9652 9677 9698 9711 9712 9718 9726 9742 9772 9773 9776 9794 9812 9816 9819 9845 9864 9877 9903 9915 9922 9926 9944 9956 9959 9959 9964 9972 9972 9998 9999
Time taken for execution is: 73
```

3. Insertion Sort

```
import java.util.*;
import java.lang.*;

public class insertion
{
```

```
public static void main(String args[])
{
int arr[]=new int[1000];
Random r=new Random();
for(int i=0;i<1000;i++)
{
arr[i]=r.nextInt(10000);}

long start=System.currentTimeMillis();
for(int i=1;i<arr.length;i++)
{
int key=arr[i];
int j=i-1;

while((j>-1) && (arr[j]>key))
{
arr[j+1]=arr[j];
j--;
}
arr[j+1]=key;
}
long end=System.currentTimeMillis();

for(int i=0;i<arr.length;i++)
{
System.out.print(arr[i]+ " ");
}

long time=end-start;
System.out.println("\nTime taken for sorting is: " + time);
}
}
```

Output:-

```
E:\Files_sem-5>javac insertion.java

E:\Files_sem-5>java insertion
1 15 34 36 56 60 75 75 79 101 115 151 152 176 183 186 212 212 233 236 245 271 287 290 291 297 298 301 306 312 333 358 383 384 387 396 407 417 421 423 452 456 463 465 47
4 477 487 488 489 496 506 513 516 518 524 524 525 528 530 537 573 576 577 589 598 618 629 631 641 650 660 681 710 725 727 742 750 762 773 798 805 816 817 821 862 866 87
8 906 917 934 935 954 973 986 1012 1017 1027 1034 1053 1059 1086 1091 1128 1134 1134 1142 1144 1145 1145 1148 1154 1168 1180 1182 1183 1196 1199 1200 1207 1208 1213 122
1 1227 1234 1236 1242 1251 1263 1273 1291 1340 1342 1344 1355 1357 1360 1379 1398 1405 1407 1416 1425 1426 1430 1439 1464 1486 1487 1488 1510 1534 1539 1547 1555 1556 1
578 1586 1588 1610 1612 1623 1625 1631 1642 1651 1654 1669 1676 1682 1687 1700 1706 1773 1785 1791 1793 1798 1800 1833 1864 1884 1884 1892 1894 1896 1898 1901 1907 1910
 1913 1915 1917 1920 1923 1937 1939 1961 1976 1990 2011 2016 2019 2031 2048 2065 2071 2093 2113 2114 2122 2123 2141 2148 2155 2164 2165 2169 2190 2195 2206 2225 2233 22
34 2256 2265 2267 2284 2289 2294 2297 2300 2317 2326 2344 2346 2354 2379 2386 2394 2395 2421 2423 2458 2463 2465 2465 2473 2487 2495 2498 2503 2513 2529 2530 2539 2541
2549 2563 2570 2577 2579 2583 2603 2604 2611 2614 2624 2634 2635 2642 2643 2644 2653 2691 2693 2693 2704 2720 2723 2725 2729 2731 2736 2739 2739 2753 2763 2766 2771 277
4 2785 2786 2789 2790 2792 2797 2803 2808 2841 2848 2854 2862 2884 2887 2890 2904 2913 2918 2953 2953 2958 2962 2962 2964 2966 2966 2972 2975 2978 2981 3009 3014 3021 3
023 3069 3072 3081 3084 3087 3089 3091 3133 3148 3156 3169 3170 3172 3198 3198 3204 3215 3225 3230 3251 3279 3287 3289 3329 3334 3337 3341 3347 3353 3378 3386 3402 3408
 3418 3435 3437 3437 3440 3442 3459 3480 3491 3493 3506 3509 3521 3533 3567 3571 3574 3581 3591 3597 3636 3638 3643 3651 3657 3661 3668 3683 3705 3716 3744 3773 3774 37
82 3799 3874 3875 3881 3892 3897 3902 3914 3932 3935 3948 3970 3975 3977 3985 3988 3996 4005 4020 4033 4037 4039 4064 4072 4085 4096 4108 4121 4136 4149 4159 4161 4165
4167 4174 4181 4182 4189 4217 4222 4238 4246 4248 4263 4272 4285 4292 4293 4296 4296 4310 4327 4328 4329 4359 4370 4373 4388 4394 4418 4437 4491 4491 4491 4498 4508 452
0 4547 4549 4550 4557 4567 4573 4584 4589 4598 4606 4607 4623 4629 4649 4650 4654 4656 4674 4677 4712 4722 4736 4752 4752 4791 4793 4797 4803 4844 4873 4886 4894 4905 4
906 4928 4931 4938 4945 4946 4949 4950 4966 4982 4990 5001 5012 5023 5023 5030 5030 5044 5058 5059 5072 5076 5082 5088 5089 5093 5121 5143 5144 5145 5145 5146 5148 5158
 5173 5179 5202 5203 5207 5208 5218 5230 5247 5269 5271 5280 5291 5292 5300 5303 5306 5307 5314 5370 5370 5375 5379 5382 5384 5415 5487 5487 5497 5503 5515 5525 5525 55
36 5542 5542 5548 5550 5551 5558 5569 5583 5592 5607 5608 5624 5624 5646 5649 5655 5656 5657 5667 5672 5682 5718 5719 5748 5750 5751 5754 5756 5760 5768 5777 5778 5787
5788 5791 5799 5805 5806 5839 5840 5841 5860 5860 5901 5902 5911 5912 5919 5920 5925 5929 5932 5933 5937 5941 5957 5968 5984 6019 6032 6083 6085 6124 6139 6159 6166 617
0 6185 6201 6202 6209 6210 6213 6224 6254 6255 6259 6261 6264 6265 6268 6299 6299 6319 6323 6335 6341 6358 6359 6374 6384 6395 6402 6414 6415 6429 6437 6450 6455 6476 6
478 6482 6483 6488 6491 6499 6504 6504 6506 6510 6514 6532 6533 6549 6551 6558 6560 6561 6567 6578 6586 6592 6596 6608 6611 6613 6618 6655 6662 6668 6674 6703 6705 6735
 6736 6744 6751 6774 6774 6778 6781 6798 6804 6809 6814 6827 6835 6835 6849 6866 6895 6905 6910 6933 6944 6950 6957 6977 6984 6996 7021 7038 7040 7047 7051 7070 7094 71
09 7133 7145 7145 7155 7164 7181 7186 7212 7213 7222 7225 7258 7261 7264 7269 7274 7282 7283 7283 7299 7316 7322 7326 7340 7346 7348 7352 7355 7358 7362 7370 7385 7387
7390 7393 7398 7409 7409 7411 7423 7477 7480 7497 7501 7505 7528 7542 7550 7555 7559 7574 7577 7577 7587 7590 7594 7604 7615 7625 7642 7645 7652 7653 7710 7726 7729 775
6 7757 7767 7789 7793 7821 7822 7825 7841 7850 7851 7856 7858 7861 7876 7884 7951 7968 7975 7992 7992 8031 8036 8050 8095 8112 8142 8154 8155 8178 8205 8218 8233 8247 8
250 8252 8253 8254 8270 8284 8295 8304 8329 8334 8335 8342 8342 8362 8379 8400 8400 8446 8452 8468 8471 8498 8507 8515 8516 8542 8543 8546 8549 8558 8566 8569 8577 8597
 8598 8605 8608 8618 8634 8637 8638 8638 8643 8697 8708 8712 8714 8721 8728 8755 8797 8802 8813 8814 8820 8827 8844 8845 8850 8872 8882 8892 8913 8924 8938 8945 8947 89
66 8971 8990 9018 9026 9029 9043 9049 9065 9075 9075 9086 9091 9119 9130 9131 9162 9163 9181 9191 9209 9211 9221 9229 9234 9268 9277 9279 9287 9298 9316 9318 9318 9338
9345 9347 9351 9357 9361 9366 9370 9379 9385 9385 9404 9417 9423 9426 9428 9434 9438 9442 9461 9470 9471 9487 9496 9497 9523 9528 9529 9534 9541 9552 9571 9591 9599 961
0 9613 9615 9631 9635 9642 9659 9660 9667 9670 9680 9686 9690 9691 9696 9702 9705 9717 9717 9718 9722 9723 9725 9731 9741 9750 9754 9780 9796 9808 9811 9898 9908 9910 9
921 9942 9947 9951 9958
Time taken for sorting is: 0
```

4. Merge Sort

```
import java.util.*;
import java.lang.*;

public class merge
{
public static void main(String args[])
{
int arr[]=new int[1000];

Random r=new Random();
for(int i=0;i<1000;i++)
{
arr[i]=r.nextInt(10000);}

merge m=new merge();

long start=System.currentTimeMillis();
m.mergesort(arr,0,arr.length-1);


for(int i=0;i<arr.length;i++)
{
System.out.print(arr[i] + " ");
}
long end=System.currentTimeMillis();
long time=end-start;
System.out.println("\nTime taken for sorting is: " + time);
}

public void mergesorting(int a[],int start,int mid,int end)
{
int i,j,k;
int n1=mid-start+1;
int n2=end-mid;

int leftarray[]=new int[n1];
int rightarray[]=new int[n2];

for(i=0;i<n1;i++)
{
leftarray[i]=a[start+i];
}
for(j=0;j<n2;j++)
{
rightarray[j]=a[mid+1+j];
}

i=0;
j=0;
```

```
k=start;
while(i<n1 && j<n2)
{
if(leftarray[i]<=rightarray[j])
{
a[k]=leftarray[i];
i++;
}
else
{
a[k]=rightarray[j];
j++;
}
k++;
}
while(i<n1)
{
a[k] = leftarray[i];
i++;
k++;
}
while (j<n2)
{
a[k] = rightarray[j];
j++;
k++;    }   }
public void mergesort(int a[],int start,int end)
{
if(start<end)
{
int mid=(start+end)/2;
mergesort(a,start,mid);
mergesort(a,mid+1,end);
mergesorting(a,start,mid,end);}}}
```

Output:

```
E:\Files_sem-5>javac merge.java

E:\Files_sem-5>java merge
11 19 21 42 47 62 66 73 83 95 98 108 131 144 151 154 162 168 176 181 184 196 198 199 208 208 217 239 272 282 292 301 314 316 326 330 340 395 403 425 441 442 464 465 475
 480 486 494 499 518 528 537 546 547 548 557 564 571 606 606 619 623 660 677 690 697 720 729 732 736 746 755 761 768 771 785 785 791 792 799 801 801 813 814 818 819 826
 836 839 852 862 864 882 888 898 902 906 916 917 922 930 934 936 945 952 962 981 982 984 1056 1064 1089 1096 1096 1100 1107 1121 1121 1130 1146 1146 1147 1179 1214 1217
 1220 1224 1229 1234 1247 1259 1262 1294 1314 1325 1330 1333 1335 1338 1345 1359 1401 1407 1410 1414 1418 1419 1426 1428 1429 1429 1433 1447 1459 1476 1497 1517 1526 15
35 1544 1554 1579 1600 1610 1610 1659 1666 1690 1697 1706 1709 1709 1710 1717 1731 1738 1760 1764 1766 1767 1770 1776 1785 1788 1789 1792 1809 1851 1864 1866 1897 1910
1938 1942 1942 1953 1964 1965 1975 1990 2010 2019 2066 2086 2103 2112 2119 2129 2135 2150 2177 2186 2203 2209 2212 2214 2218 2223 2233 2239 2252 2253 2260 2278 2293 229
5 2341 2374 2375 2384 2386 2387 2389 2396 2413 2417 2428 2441 2443 2495 2509 2520 2540 2549 2552 2569 2582 2583 2586 2604 2641 2666 2694 2706 2719 2721 2734 2744 2748 2
752 2755 2758 2775 2777 2816 2816 2820 2820 2822 2839 2847 2854 2856 2858 2867 2890 2891 2895 2925 2958 2970 2987 2999 3008 3011 3012 3028 3061 3064 3089 3111 3126 3130
 3132 3157 3159 3181 3181 3189 3191 3199 3207 3218 3226 3228 3253 3266 3270 3270 3290 3291 3292 3297 3298 3316 3329 3332 3344 3359 3378 3402 3407 3465 3468 3483 3496 35
09 3515 3521 3533 3535 3542 3542 3544 3561 3572 3587 3588 3589 3592 3594 3596 3646 3666 3668 3694 3710 3715 3721 3733 3739 3742 3744 3747 3765 3768 3770 3779 3782 3786
3789 3796 3810 3833 3843 3843 3860 3861 3864 3873 3880 3880 3897 3914 3926 3960 3983 3994 3997 4001 4003 4027 4050 4070 4087 4089 4110 4122 4128 4133 4141 4154 4158 416
7 4170 4191 4205 4211 4221 4228 4247 4255 4288 4295 4299 4319 4321 4334 4335 4336 4338 4338 4339 4342 4353 4355 4386 4391 4395 4397 4402 4413 4430 4435 4441 4447 4447 4
456 4474 4477 4487 4489 4501 4511 4517 4528 4558 4566 4570 4596 4609 4610 4612 4615 4617 4623 4627 4650 4656 4657 4665 4678 4692 4706 4709 4709 4714 4754 4767 4806 4811
 4811 4841 4848 4853 4858 4858 4870 4876 4876 4880 4887 4900 4906 4911 4917 4953 4954 4989 4999 5013 5022 5024 5051 5054 5054 5059 5059 5065 5068 5072 5074 5089 5093 50
94 5096 5108 5113 5114 5116 5127 5145 5148 5160 5166 5170 5174 5174 5180 5181 5181 5183 5194 5200 5201 5203 5205 5220 5227 5228 5234 5247 5255 5260 5274 5294 5296 5303
5354 5354 5363 5366 5374 5383 5390 5425 5436 5442 5449 5454 5461 5474 5477 5480 5493 5521 5525 5546 5548 5571 5598 5615 5645 5662 5666 5679 5683 5685 5695 5711 5733 574
7 5756 5776 5785 5793 5799 5820 5844 5845 5848 5855 5860 5869 5878 5907 5908 5931 5955 5967 5969 5970 5997 6008 6012 6013 6026 6028 6032 6036 6042 6140 6157 6213 6235 6
260 6260 6260 6290 6304 6311 6313 6314 6324 6336 6368 6386 6397 6397 6399 6412 6419 6433 6439 6463 6467 6470 6493 6493 6526 6553 6556 6563 6565 6569 6579 6584 6588 6603
 6621 6621 6635 6652 6654 6668 6673 6678 6692 6693 6695 6713 6741 6776 6801 6828 6830 6834 6854 6858 6859 6865 6898 6910 6912 6924 6927 6929 6933 6943 6948 6948 6950 69
86 6990 6993 6999 7012 7014 7021 7034 7035 7037 7050 7057 7095 7096 7102 7107 7108 7111 7113 7123 7124 7146 7148 7152 7154 7178 7183 7202 7208 7209 7215 7218 7224 7234
7241 7248 7253 7284 7294 7297 7311 7338 7343 7348 7349 7359 7368 7373 7385 7397 7399 7400 7415 7420 7431 7455 7472 7478 7485 7488 7496 7511 7517 7539 7545 7548 7550 755
2 7556 7565 7567 7587 7593 7595 7598 7615 7615 7643 7643 7649 7662 7665 7727 7745 7749 7764 7770 7787 7815 7815 7838 7856 7862 7911 7913 7916 7966 7988 8010 8014 8024 8
031 8050 8055 8059 8073 8080 8087 8089 8092 8105 8118 8136 8139 8148 8148 8158 8164 8178 8204 8210 8212 8216 8225 8232 8232 8241 8250 8270 8271 8304 8319 8325 8329 8354
 8356 8366 8372 8388 8390 8394 8394 8401 8409 8409 8414 8424 8436 8458 8459 8474 8474 8487 8491 8497 8499 8503 8505 8507 8509 8511 8512 8516 8518 8525 8534 8541 8542 85
55 8556 8566 8577 8582 8583 8602 8614 8615 8618 8639 8639 8652 8657 8659 8659 8661 8680 8707 8708 8709 8745 8746 8772 8774 8785 8786 8799 8799 8799 8802 8815 8818 8831
8849 8854 8874 8879 8885 8892 8892 8899 8914 8921 8924 8925 8934 8951 8957 8960 8972 8977 8992 9000 9005 9022 9027 9029 9033 9038 9039 9060 9067 9076 9079 9082 9094 909
6 9110 9112 9112 9134 9136 9137 9157 9162 9166 9173 9180 9185 9188 9194 9194 9196 9197 9199 9204 9214 9232 9239 9254 9258 9285 9309 9314 9333 9336 9341 9346 9365 9368 9
376 9378 9408 9413 9419 9431 9456 9475 9489 9491 9500 9501 9507 9542 9542 9549 9557 9563 9565 9567 9569 9569 9579 9587 9588 9621 9625 9637 9657 9666 9672 9675 9680 9717
 9730 9758 9763 9770 9774 9779 9789 9799 9811 9816 9827 9833 9858 9862 9868 9869 9874 9879 9885 9890 9891 9906 9911 9926 9927 9937 9939 9949 9951 9952 9968 9980 9981 99
84 9999
Time taken for sorting is: 56
```

5. Quick Sort

```
import java.util.*;
import java.lang.*;

public class quick
{
public static void main(String args[])
{
int arr[]=new int[1000];

Random r=new Random();
for(int i=0;i<1000;i++)
{
arr[i]=r.nextInt(10000);
}

quick q=new quick();

long start=System.currentTimeMillis();
q.quicksort(arr,0,arr.length-1);

for(int i=0;i<arr.length;i++)
{
System.out.print(arr[i] + " ");
}
long end=System.currentTimeMillis();

long time=end-start;
System.out.println("\nTime taken for sorting is: "+ time);

}
public int partition(int a[],int start,int end){
int pivot=a[end];

int j=start-1;

for(int i=start;i<=end;i++)
{
if(a[i]<pivot)
{
j++;
int temp=a[j];
a[j]=a[i];
a[i]=temp;
}
}
int temp=a[j+1];
a[j+1]=a[end];
a[end]=temp;
return (j+1);
```

```
}
public void quicksort(int a[],int start,int end)
{
if(start<end)
{
int p=partition(a,start,end);
quicksort(a,start,p-1);
quicksort(a,p+1,end);}}}
```

Output:

```
E:\Files_sem-5>javac quick.java

E:\Files_sem-5>java quick
0 2 5 18 40 52 70 81 87 89 93 101 105 131 131 151 200 220 238 242 261 264 266 269 281 292 302 310 328 339 344 359 361 365 385 394 395 420 424 451 465 472 488 489 499 50
1 504 521 535 548 557 562 568 592 608 610 627 631 645 651 656 662 664 667 695 697 701 722 724 740 755 758 767 779 788 795 801 812 836 838 850 868 875 904 921 921 941 94
3 962 963 963 964 981 988 1000 1000 1001 1026 1056 1059 1069 1073 1087 1092 1095 1112 1117 1119 1149 1171 1185 1185 1203 1206 1215 1243 1246 1259 1267 1269 1289 1307 13
24 1325 1334 1335 1341 1354 1365 1367 1370 1375 1384 1402 1423 1430 1442 1454 1454 1458 1481 1498 1498 1505 1510 1511 1517 1523 1524 1526 1527 1527 1544 1577 1578 1582
1587 1589 1596 1598 1625 1672 1677 1682 1685 1699 1720 1756 1763 1766 1769 1795 1806 1810 1854 1860 1868 1880 1891 1891 1892 1894 1910 1915 1920 1920 1940 1952 1954 196
3 1969 1974 1981 1986 1999 2003 2004 2046 2048 2055 2082 2085 2102 2106 2132 2133 2144 2148 2157 2169 2171 2189 2224 2239 2248 2258 2258 2259 2264 2268 2269 2269 2286 2
289 2290 2296 2304 2316 2333 2338 2352 2353 2362 2362 2377 2397 2403 2413 2440 2444 2453 2462 2477 2480 2484 2497 2506 2508 2514 2517 2518 2520 2520 2524 2535 2546 2551
 2557 2560 2561 2566 2642 2644 2648 2677 2692 2694 2696 2707 2716 2718 2738 2755 2756 2764 2765 2768 2776 2800 2804 2825 2827 2830 2834 2841 2842 2849 2863 2864 2865 28
78 2882 2882 2889 2906 2908 2925 2928 2949 2952 2952 2955 2962 2971 3001 3006 3008 3009 3009 3012 3013 3042 3045 3055 3067 3091 3095 3096 3116 3116 3120 3133 3141 3152
3161 3165 3166 3166 3171 3174 3178 3186 3189 3195 3205 3222 3227 3251 3255 3260 3265 3293 3297 3297 3299 3314 3319 3333 3351 3353 3359 3361 3372 3380 3385 3405 3405 340
7 3414 3414 3421 3423 3451 3460 3468 3493 3495 3499 3501 3525 3547 3547 3568 3569 3572 3578 3584 3587 3599 3630 3643 3659 3678 3686 3710 3715 3720 3736 3738 3751 3756 3
763 3782 3794 3794 3801 3804 3805 3807 3816 3816 3835 3840 3840 3863 3867 3867 3872 3874 3888 3899 3902 3903 3907 3914 3928 3956 3966 3981 3983 3983 3989 3992 3995 3998
 4004 4004 4054 4065 4065 4076 4083 4086 4100 4135 4141 4144 4148 4152 4156 4165 4169 4173 4203 4226 4227 4246 4270 4277 4284 4292 4310 4321 4345 4382 4386 4400 4405 44
11 4430 4431 4440 4474 4475 4478 4480 4502 4539 4540 4544 4551 4555 4560 4562 4588 4592 4624 4631 4635 4663 4670 4677 4703 4720 4727 4732 4763 4765 4768 4769 4782 4815
4819 4823 4829 4842 4850 4881 4884 4902 4902 4904 4904 4932 4963 4971 4972 4978 5002 5019 5039 5045 5067 5071 5075 5093 5107 5112 5139 5161 5173 5186 5187 5242 5255 525
6 5274 5275 5279 5296 5299 5314 5349 5351 5355 5378 5379 5388 5394 5413 5431 5448 5460 5460 5482 5530 5539 5546 5547 5554 5562 5572 5590 5592 5592 5633 5642 5646 5670 5
686 5696 5705 5716 5736 5744 5757 5778 5787 5804 5809 5810 5816 5848 5850 5870 5878 5879 5898 5905 5948 5948 5952 5966 5966 5969 5978 5982 5999 6004 6004 6005 6010 6019
 6025 6027 6027 6041 6047 6086 6088 6103 6158 6160 6175 6178 6185 6190 6221 6233 6237 6243 6282 6286 6312 6326 6328 6335 6357 6366 6367 6379 6384 6390 6394 6414 6430 64
48 6448 6463 6496 6506 6520 6523 6533 6534 6540 6551 6570 6580 6580 6582 6600 6610 6615 6615 6620 6623 6636 6637 6650 6658 6666 6681 6696 6720 6722 6727 6736 6741 6745
6766 6771 6774 6775 6776 6781 6784 6806 6814 6836 6838 6843 6846 6859 6868 6869 6884 6899 6908 6935 6938 6954 6963 6965 6971 6977 6989 6993 6999 7001 7003 7008 7027 704
1 7041 7044 7072 7079 7111 7116 7118 7123 7137 7137 7144 7178 7185 7202 7210 7230 7247 7260 7274 7295 7315 7326 7334 7335 7339 7340 7346 7352 7365 7384 7384 7385 7388 7
396 7398 7400 7405 7405 7411 7413 7422 7425 7432 7447 7447 7451 7460 7472 7482 7498 7498 7507 7530 7536 7566 7570 7579 7582 7587 7607 7614 7618 7621 7637 7649 7652 7659
 7661 7664 7688 7695 7701 7707 7733 7740 7749 7752 7753 7758 7765 7768 7776 7786 7815 7816 7835 7838 7852 7862 7866 7870 7879 7882 7898 7917 7919 7934 7936 7941 7948 79
58 7970 7972 7997 7999 8007 8017 8018 8019 8024 8027 8036 8040 8040 8041 8062 8070 8083 8097 8117 8149 8151 8158 8171 8178 8187 8188 8192 8192 8196 8201 8214 8224 8227
8229 8232 8234 8238 8248 8290 8335 8356 8357 8361 8362 8384 8397 8403 8403 8408 8426 8432 8434 8453 8487 8494 8513 8529 8531 8536 8537 8538 8556 8557 8562 8576 8583 860
9 8617 8630 8663 8683 8688 8709 8718 8740 8744 8752 8762 8765 8766 8785 8798 8807 8810 8814 8814 8831 8834 8835 8849 8851 8871 8872 8879 8886 8897 8912 8912 8930 8936 8
937 8949 8963 8970 8975 8979 8984 8989 9041 9047 9048 9049 9055 9063 9077 9080 9085 9102 9109 9126 9142 9161 9171 9172 9177 9187 9195 9196 9202 9299 9314 9315 9316 9318
 9319 9321 9327 9329 9332 9345 9355 9360 9365 9367 9367 9371 9381 9382 9390 9400 9404 9422 9437 9445 9473 9540 9546 9565 9572 9577 9579 9585 9587 9601 9627 9633 9634 96
46 9657 9665 9668 9673 9699 9707 9718 9719 9731 9736 9749 9750 9753 9758 9794 9799 9809 9815 9818 9827 9840 9841 9849 9853 9855 9857 9858 9866 9884 9897 9915 9916 9937
9954 9960 9970 9978
Time taken for sorting is: 72
```

Observations:

Write observation based on amount of time executed by each algorithm.

- Bubble and selection sort takes much similar amount of time for complete it's execution While insertion sort takes less time as compared to above sorts. Quick and merge are considered highly efficient and are commonly used for sorting large inputs.

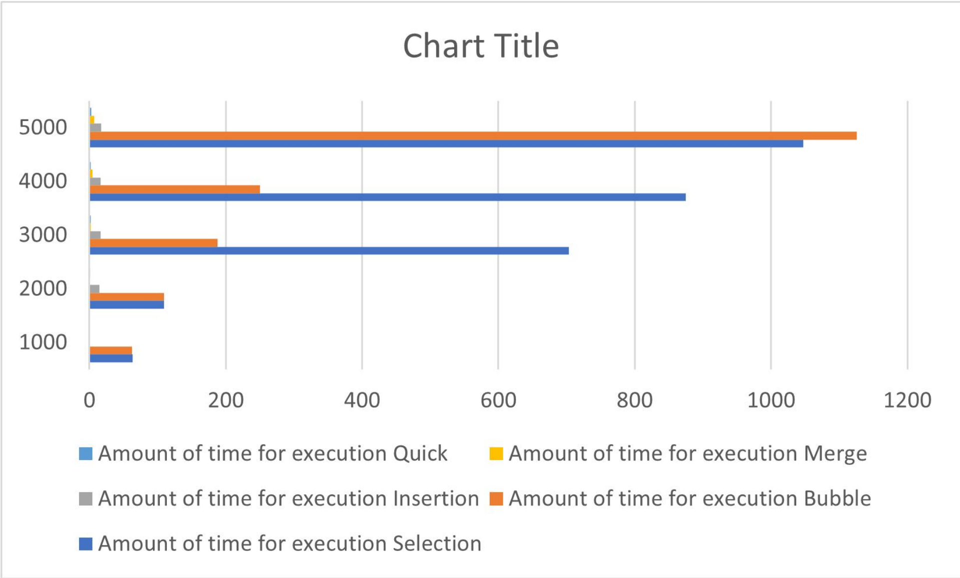
Result: Complete the below table based on your implementation of functions and steps executed by each function.**Also, prepare similar tables for ascending order sorted data and descending order sorted data.**

Inputs	Amount of time for execution (Random Data)				
	Selection	Bubble	Insertion	Merge	Quick
1000	63	62	0	0	0
2000	109	109	15	0	1
3000	703	188	16	2	2
4000	875	250	16	4	2
5000	1047	1126	17	7	3
Time Complexity	O(n^2)	O(n^2)	O(n^2)	O(n log(n))	O(n log(n))

Chart:

<Draw Comparative Charts of inputs versus number of steps executed on various data (Random, ascending

order sorted and descending order sorted data) by each algorithm.



Quiz:

1. Which sorting function execute faster (has small steps count) in case of ascending order sorted data?

Answer: If the data is already in ascending order, the sorting function that would execute the fastest (with the smallest step count) is Insertion Sort.

2. Which sorting function execute faster (has small steps count) in case of descending order sorted data?

Answer: In the case of data that is already sorted in descending order, the sorting function that would execute the fastest (with the smallest step count) is Selection Sort.

3. Which sorting function execute faster (has small steps count) in case of random data?

Answer: In the case of random data, the sorting function that generally executes faster (with a smaller step count) is Quick Sort.

4. On what kind of data, the best case of Bubble sort occurs?

Answer: The best-case scenario for Bubble Sort occurs when the input data is already sorted in ascending order.

5. On what kind of data, the worst case of Bubble sort occurs?

Answer: The worst-case scenario for Bubble Sort occurs when the input data is already sorted in descending order.

6. On what kind of data, the best case of Quick sort occurs?

Answer: The best-case scenario for Quick Sort occurs when the pivot chosen for partitioning the data is always close to the median value of the input. This leads to the most balanced partitioning.

7. On what kind of data, the worst case of Quick sort occurs?

Answer: The worst-case scenario for Quick Sort occurs when the chosen pivot element consistently results in highly unbalanced partitions.

8. Which sorting algorithms are in-place sorting algorithms?

Answer: Bubble sort , selection sore , insertion sort , quick sort

9. Which sorting algorithms are stable sorting algorithms?

Answer: Insertion sort , merge sort , bubble sort

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	**Understanding of problem (3)**			**Program Implementation (5)**			**Documentation &Timely Submission (2)**			**Total (10)**
	Good (3)	**Avg. (2-1)**	**Poor (1-0)**	**Good (5-4)**	**Avg. (3-2)**	**Poor (1-0)**	**Good (2)**	**Avg. (1)**	**Poor (0)**	
Marks										

Experiment No: 3

Implement a function of sequential search and count the steps executed by function on various inputs (1000 to 5000) for best case, average case and worst case. Also, write time complexity in each case and draw a comparative chart of number of input versus steps executed by sequential search for each case.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1

Objectives: (a) Identify Best, Worst and Average cases of given problem.
(b) Derive time complexity from steps count on various inputs.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Steps to implement sequential search is as below:

1. Take an input array A of n elements and a key value K.
2. Define a variable pos, initially set to -1.
3. Iterate through the array A, starting from the first element and continuing until either the key value is found or the end of the array is reached.
4. For each element, compare its value to the key value K.
5. If the values match, set pos to the index of the current element and exit the loop.
6. If the end of the array is reached and the key value has not been found, posremain equal to -1.
7. Output the value of pos.

The algorithm works by sequentially iterating through the elements of the array and comparing each element to the target value. If a match is found, the algorithm exits the loop.

Implement above functions and calculate the number of steps executed by each functions on various inputs ranging from 1000 to 5000. Take a counter variable to calculate the number of steps and increment it for each statement.Based on algorithm's logic, decide best, worst and average case inputs for the algorithm and prepare a table of steps count.

Program:

```
import java.util.*;
import java.lang.*;

public class sequential
{
public static void main(String args[])
{
```

```
int arr[]=new int[1000];
Random r=new Random();
for(int i=0;i<1000;i++)
{
arr[i]=r.nextInt(10000);
System.out.print(arr[i] + " ");
}
Scanner sc=new Scanner(System.in);
System.out.println("\nEnter element that you want to search for:");
int num=sc.nextInt();
int flag=0, index=-1;
long start=System.nanoTime();
for(int i=0;i<arr.length;i++)
{
if(arr[i]==num)
{
flag=1;
index=i;
break;
}
}
long end=System.nanoTime();
if(flag==1)
{
System.out.println("Element is found at " + index + " position");
}
else
{
System.out.println("Element not found in array");
}
long time=end-start;
System.out.println(time);
}
}
```

Output:

```
E:\Files_sem-5>javac sequential.java

E:\Files_sem-5>java sequential
4957 4966 5331 3519 4491 4414 705 9364 6763 59 1657 1715 1161 7625 6107 3376 3083 3953 4631 756 1435 576 8900 9624 2080 7282 4656 4492 6411 1984 4961 5 3165 5027 1609 2
513 5350 7844 9992 9176 9704 5587 4203 977 618 5295 2307 8078 1624 5544 4561 6165 2360 1631 1533 939 9639 2182 5285 4559 5298 2265 9138 1857 9617 3945 9143 2157 4461 17
29 3699 8936 8435 9895 2357 9606 9040 6865 495 5 9980 3424 6867 6153 99 6398 8459 2501 4590 2256 2421 4355 2097 7879 4970 4160 381 3145 6432 2695
Enter element that you want to search for:
7282
Element is found at 25 position
```

Observations:

Write observation based on amount of time for execution executed by algorithm.

- Sequential search takes less time in best case as element is present at starting of array. It takes more time when the element is present at last position of array.

Result: Complete the below table based on your implementation of sequential search algorithm and steps executed by the function.

Inputs	Number of Steps Executed		
	Best Case	**Average Case**	**Worst Case**
1000	600	8400	26900
2000	600	24800	40700
3000	600	32800	59200
4000	600	41400	76200
5000	600	50700	93600
Time Complexity	O(1)	O(n)	O(n)

Chart:

<Draw Comparative Chart of inputs versus number of steps executed by algorithm in various cases>

Quiz:

1. Which is the best case of an algorithm?

Answer: The best case of an algorithm refers to the scenario in which the algorithm performs with the lowest possible computational cost or resource usage.

2. Which is the worst case of an algorithm?

Answer: The worst case of an algorithm refers to the scenario in which the algorithm performs with the highest possible computational cost or resource usage.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

<table>
<tr><th rowspan="2">Rubrics</th><th colspan="3">Understanding of problem
(3)</th><th colspan="3">Program Implementation
(5)</th><th colspan="3">Documentation &Timely Submission
(2)</th><th>Total
(10)</th></tr>
<tr><th>Good
(3)</th><th>Avg.
(2-1)</th><th>Poor
(1-0)</th><th>Good
(5-4)</th><th>Avg.
(3-2)</th><th>Poor
(1-0)</th><th>Good
(2)</th><th>Avg.
(1)</th><th>Poor
(0)</th><th></th></tr>
<tr><td>Marks</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td></td></tr>
</table>

Experiment No: 4

Compare the performances of linear search and binary search for Best case, Average case and Worst case inputs.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis, and Mathematical skills

Relevant CO: CO1, CO2

Objectives: (a) Identify Best, Worst and Average cases of given problem.
(b) Derive time complexity from steps count for different inputs.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Steps to implement binary search are as below:

1. Take an input **sorted** array A of n elements and a target value T.
2. Define variables start and end to represent the start and end indices of the search range, initially set to 0 and n-1 respectively.
3. Repeat the following steps while start <= end:
    a. Calculate the midpoint index mid as (start + end) / 2.
    b. If the value of the midpoint element A[mid] is equal to the target value T, return the value of mid.
    c. If the value of the midpoint element A[mid] is greater than the target value T, set end to mid-1.
    d. If the value of the midpoint element A[mid] is less than the target value T, set start to mid+1.
4. If the target value T is not found in the array, return -1.
5. Output the value returned in Step 3, representing the position of the target value T in the array.

Implement function of binary search algorithm and use linear search function implemented in previous practical. Compare the steps count of both the functions on various inputs ranging from 100 to 500 for each case (Best, Average, and Worst).

Program:

```
import java.util.*;
import java.lang.*;

public class binarysearch
{
public static void main(String args[])
{

int arr[]=new int[200];
Random r=new Random();
for(int i=0;i<200;i++)
{
arr[i]=r.nextInt(10000);
}
Arrays.sort(arr);
for(int i=0;i<arr.length;i++)
{
System.out.print(arr[i] + " ");
```

```
}

int midd=(0+(arr.length-1))/2;
System.out.println("\n" + arr[midd]);

int low=0;
int high=arr.length-1;
int index=-1,flag=0;
Scanner sc=new Scanner(System.in);

System.out.println("\nEnter element that you want to search for:");
int num=sc.nextInt();

long start=System.nanoTime();
while(low<=high)
{
int mid=(high+low)/2;

if(arr[mid]==num)
{
index=mid;
flag=1;
break;
}
else if(arr[mid]<num)
{
low=mid+1;
}
else if(arr[mid]>num)
{
high=mid-1;
}
}
long end=System.nanoTime();
if(flag==1)
{
System.out.println("Element found at " + index + " position");
}
else
{
System.out.println("Element not found in array");
}
long time=end-start;


System.out.println("Time taken for search is: "+ time);
}
}
```

Observations:

Write observation based on amount of time for execution by both algorithms.

- Linear search takes less amount of time as compared to binary search in best case. And time complexity of linear search is also less.

Result: Complete the below table based on your implementation of sequential search algorithm and steps executed by the function.

Best case:

Inputs	Number of Steps Executed (Best Case)	
	Linear Search	**Binary Search**
100	600	800
200	600	900
300	600	1000
400	600	1100
500	600	1200
Time Complexity	O(1)	O(1)

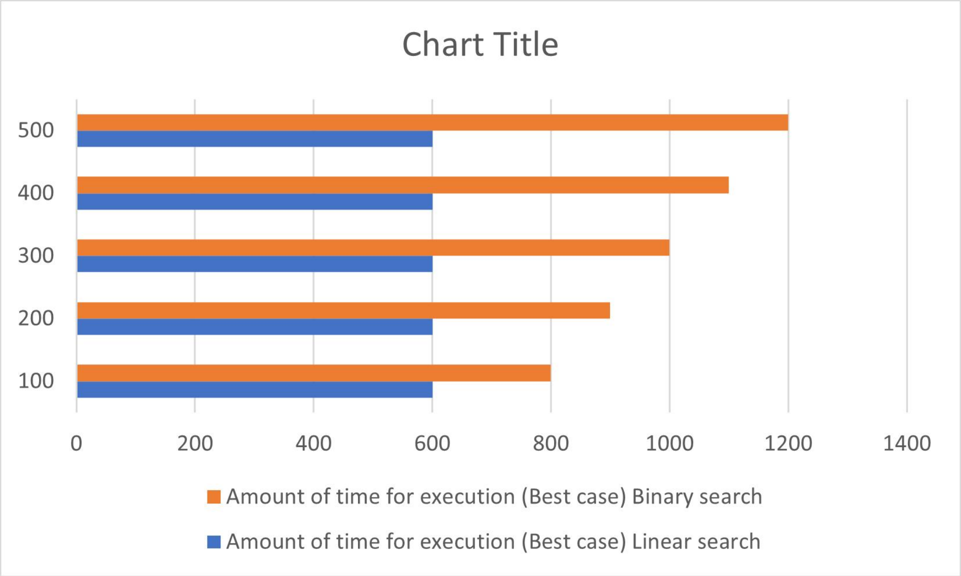
Worst case:

Inputs	Number of Steps Executed (Worst Case)	
	Linear Search	**Binary Search**
100	2800	1600
200	4600	1900
300	6200	2100
400	7700	2200
500	9500	2500
Time Complexity	O(n)	O(log(n))

Chart:

<Draw Comparative Charts of inputs versus number of steps executed by both algorithms in various cases>

Best case:



Worst case:

Quiz:

1. Which element should be searched for the best case of binary search algorithm?

Answer: The best case for the binary search algorithm occurs when the target element is exactly in the middle of the sorted array.

2. Which element should be searched for the worst case of binary search algorithm?

Answer: The worst case for the binary search algorithm occurs when the target element is not present in the sorted array.

3. Which algorithm executes faster in worst case?

Answer: In terms of time complexity, the binary search algorithm executes faster in the worst case compared to the linear search algorithm.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

<table>
<tr><th rowspan="2">Rubrics</th><th colspan="3">Understanding of problem
(3)</th><th colspan="3">Program Implementation
(5)</th><th colspan="3">Documentation &Timely Submission
(2)</th><th>Total
(10)</th></tr>
<tr><th>Good
(3)</th><th>Avg.
(2-1)</th><th>Poor
(1-0)</th><th>Good
(5-4)</th><th>Avg.
(3-2)</th><th>Poor
(1-0)</th><th>Good
(2)</th><th>Avg.
(1)</th><th>Poor
(0)</th><th rowspan="2"></th></tr>
<tr><td>Marks</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td></tr>
</table>

Experiment No: 5

Implement functions to print n^{th} Fibonacci number using iteration and recursive method. Compare the performance of two methods by counting number of steps executed on various inputs. Also draw a comparative chart. (Fibonacci series 1, 1, 2, 3, 5, 8….. Here 8 is the 6^{th} Fibonacci number).

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis

Relevant CO: CO1, CO5

Objectives: (a) Compare the performances of two different versions of same problem.
(b) Find the time complexity of algorithms.
(C) Understand the polynomial and non-polynomial problems

Equipment/Instruments: Computer System, Any C language editor

Theory:

The Fibonacci series is the sequence of numbers (also called Fibonacci numbers), where every number is the sum of the preceding two numbers, such that the first two terms are '0' and '1'. In some older versions of the series, the term '0' might be omitted. A Fibonacci series can thus be given as, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . . It can thus be observed that every term can be calculated by adding the two terms before it. We are ignoring initial zero in the series.

To represent any $(n+1)^{th}$ term in this series, we can give the expression as, $F_n = F_{n-1} + F_{n-2}$. We can thus represent a Fibonacci series as shown in the image below,

$$F(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ F(n-1) + F(n-2) & , n > 1 \end{cases}$$

Iterative version to print n^{th} Fibonacci number is as below:
Input: An integer n, where n >= 1.
Output: The n^{th} Fibonacci number.
Steps:
Initialize variables $f_0 = 1$, $f_1 = 1$, and i = 2.
If n is 1 or 2 then
    Print 1
While i < n, do
    a. Set f2 = f0 + f1.
    b. Set f0 = f1.
    c. Set f1 = f2.
    d. Increment i by 1.
Print f1.

Recursive version to print n^{th} Fibonacci number is as below:

Input: An integer n, where n >= 1.
Output: The n^{th} Fibonacci number.
If n is 1 or 2 then
    return 1.
else recursively compute next number using the $(n-1)^{th}$ and $(n-2)^{th}$ Fibonacci numbers, and return their sum.
Print the result.

Implement functions of above two versions of Fibonacci series and compare the steps count of both the functions on various inputs ranging from 10 to 50 (if memory permits for recursive version).

Code:

```
import java.util.*;
import java.lang.*;
public class fibonacci
{
public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
System.out.println("Enter a number:");
int num=sc.nextInt();
fibonacci f=new fibonacci();
long start1=System.currentTimeMillis();
f.iteration(num);
long end1=System.currentTimeMillis();

long start2=System.currentTimeMillis();
for(int i=1;i<=num;i++)
{
System.out.println(f.recursion(i));
}
long end2=System.currentTimeMillis();

long time1=end1-start1;
long time2=end2-start2;

System.out.println("Time for iteration: " + time1);
System.out.println("Time for recursion: " + time2);
}

public void iteration(int n)
{
int a=0,b=1;
System.out.println(a);
System.out.println(b);
int i=2;
while(i<n)
{
int c=a+b;
System.out.println(c);
a=b;
b=c;
```

```
i++;
}
}

public int recursion(int n)
{
if(n==1)
{
return 0;
}
else if(n==2)
{
return 1;
}
else
{
return recursion(n-1)+recursion(n-2);
}
}
}
```

Output:

```
E:\Files_sem-5>javac fibonacci.java

E:\Files_sem-5>java fibonacci
Enter a number:
25
0
1
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
Time for iteration: 8
Time for recursion: 13
```

Observations:

Write observation based on Amount of Time executed by both algorithms.

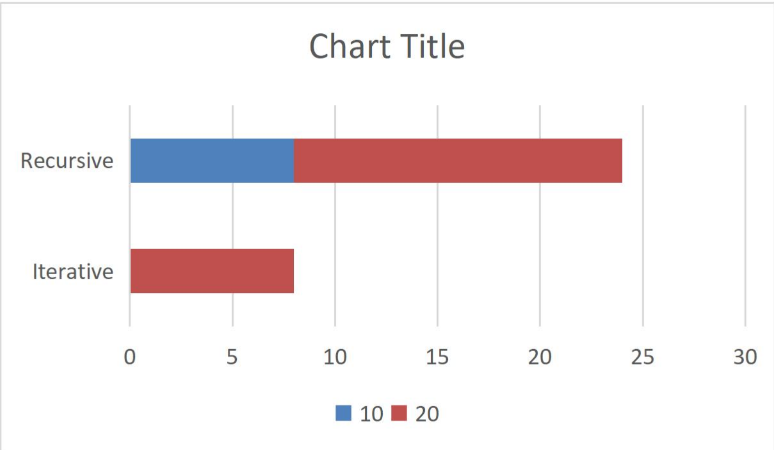
If we perform the Fibonacci series using iteration and recursion then recursion will take more amount of time.

Result: Complete the below table based on your implementation of sequential search algorithm and steps executed by the function.

Inputs	Number of Steps Executed (Random data)	
	Iterative Fibonacci	Recursive Fibonacci
10	0	8
20	8	16
30	**(if memory doesn't permit then reduce the range)**	
40		
50		
Time Complexity	O(n)	O(2^n)

Chart:

<Draw Comparative Charts of inputs versus number of steps executed by both functions on various inputs>



Conclusion:

Quiz:

1. What is the time complexity of iterative version of Fibonacci function?

Answer: Time complexity of iteration version of Fibonacci function is o(n).

2. What is the time complexity of recursive version of Fibonacci function?

Answer: Time complexity of iteration version of Fibonacci function is o(2^n).

3. Can you execute recursive version of Fibonacci function for more inputs?

Answer: Yes we can execute recursive version of Fibonacci function for more inputs but it is very low method.

4. What do you mean by polynomial time algorithms and exponential time algorithms?

Answer:

A polynomial time algorithm is an algorithm whose runtime, or the number of steps it takes to complete, is bounded by a polynomial function in terms of the size of the input.

An exponential time algorithm is an algorithm whose runtime grows exponentially with the size of the input.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 6

Implement a program for randomized version of quick sort and compare its performance with the normal version of quick sort using steps count on various inputs (1000 to 5000) ofrandom nature, ascending order and descending order sorted data.Also, draw a comparative chart of number of input versus steps executed/time taken for each cases (random, ascending, and descending).

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis

Relevant CO: CO1, CO2

Objectives: (a) Improve the performance of quick sort in worst case.
(b) Compare the performance of both the version of quick sort on various inputs

Equipment/Instruments: Computer System, Any C language editor

Theory:

Steps to implement randomized version of quick sort are as below:

```
RANDOMIZED-QUICKSORT(A, low, high)
        if (low< high) {
        pivot= RANDOMIZED_PARTITION(A, low, high);
        RANDOMIZED-QUICKSORT(A, low, pivot);
        RANDOMIZED-QUICKSORT(A, pivot+1,high);
}
RANDOMIZED_PARTITION (A,low,high) {
    pos = Random(low, high)
    pivot = A[pos] swap(pivot,
    a[low])
    left = low
    right = high
    while ( left <right ) {
         /* Move left while item < pivot */
        while( A[left] <= pivot ) left++;
        /* Move right while item > pivot */
        while( A[right] > pivot ) right--;
        if ( left < right )
                swap(A[left],A[right]);}
 /* right is final position for the pivot */
swap(A[right], pivot);
return right; }
```

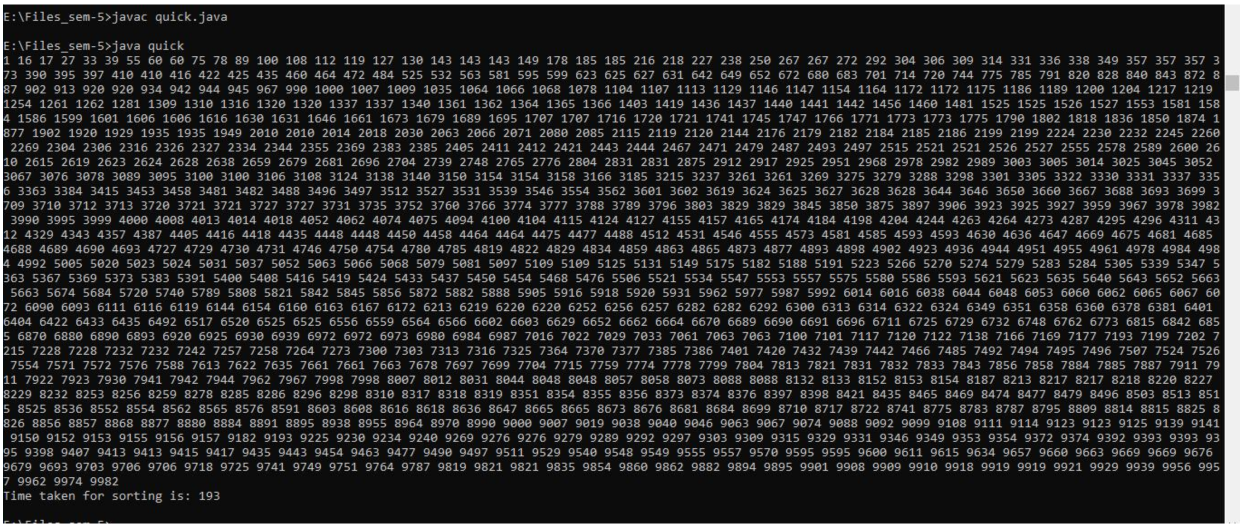
Implement a function of randomized version of quick sort as per above instructions and use basic version of quick sort. Compare the steps count of both the functions on various inputs ranging from 1000 to 5000 for each case (random, ascending, and descending).

Code of basic quick sort:

```
import java.util.*;
import java.lang.*;

public class quick
{
public static void main(String args[])
{
int arr[]=new int[1000];

Random r=new Random();
for(int i=0;i<1000;i++)
{
arr[i]=r.nextInt(10000);
}

quick q=new quick();

long start=System.currentTimeMillis();
q.quicksort(arr,0,arr.length-1);

for(int i=0;i<arr.length;i++)
{
System.out.print(arr[i] + " ");
}
long end=System.currentTimeMillis();

long time=end-start;
System.out.println("\nTime taken for sorting is: "+ time);

}

public int partition(int a[],int start,int end)
{
int pivot=a[end];
int j=start-1;

for(int i=start;i<=end;i++)
{
if(a[i]<pivot)
{
j++;
int temp=a[j];
a[j]=a[i];
a[i]=temp;
}}


int temp=a[j+1];
a[j+1]=a[end];
```

```
a[end]=temp;
return (j+1);
}

public void quicksort(int a[],int start,int end)
{
if(start<end)
{
int p=partition(a,start,end);
quicksort(a,start,p-1);
quicksort(a,p+1,end);
}
}
}
```

Output of basic quick sort:

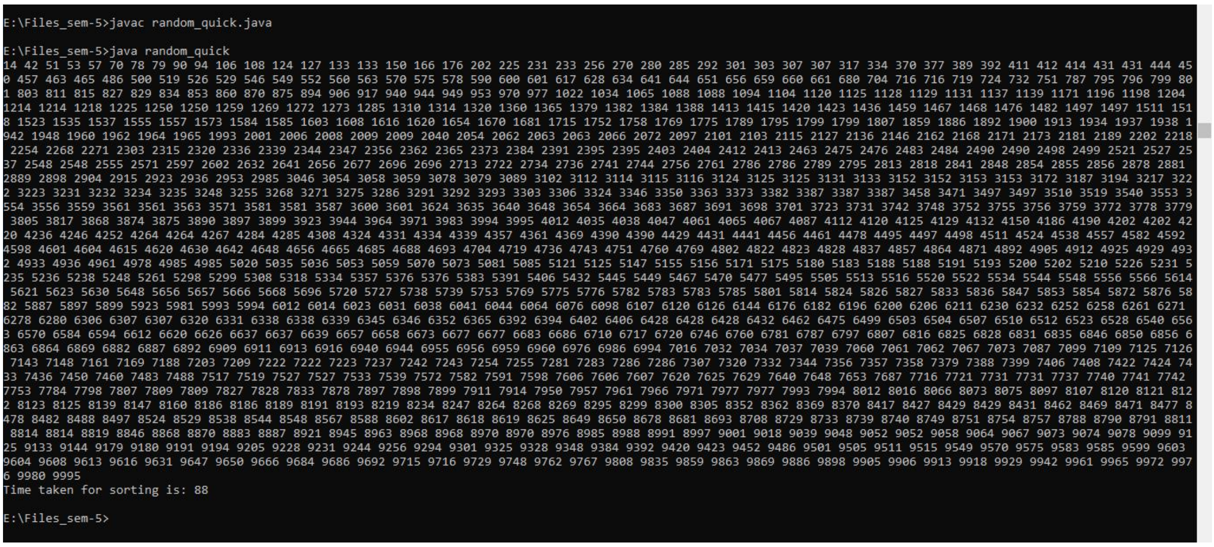


Code for randomized quick sort:

```
import java.util.Random;
import java.util.Scanner;

public class random_quick {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int arr[]=new int[1000];

       Random r=new Random();
       for(int i=0;i<1000;i++)
       {
       arr[i]=r.nextInt(10000);
       }

       long start=System.currentTimeMillis();
       randomizedQuickSort(arr,0,arr.length-1);

       for(int i=0;i<arr.length;i++)
       {
       System.out.print(arr[i] + " ");
       }
```

```
        long end=System.currentTimeMillis();

        long time=end-start;
        System.out.println("\nTime taken for sorting is: "+ time);

    }


    public static void randomizedQuickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pivotIndex = randomizedPartition(arr, low, high);
            randomizedQuickSort(arr, low, pivotIndex - 1);
            randomizedQuickSort(arr, pivotIndex + 1, high);
        }
    }

    public static int randomizedPartition(int[] arr, int low, int high) {
        Random random = new Random();
        int randomIndex = random.nextInt(high - low + 1) + low;
        int temp = arr[randomIndex];
        arr[randomIndex] = arr[high];
        arr[high] = temp;
        return partition(arr, low, high);
    }

    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

Output:



Observations:

Write observation based on amount of time taken by both algorithms.

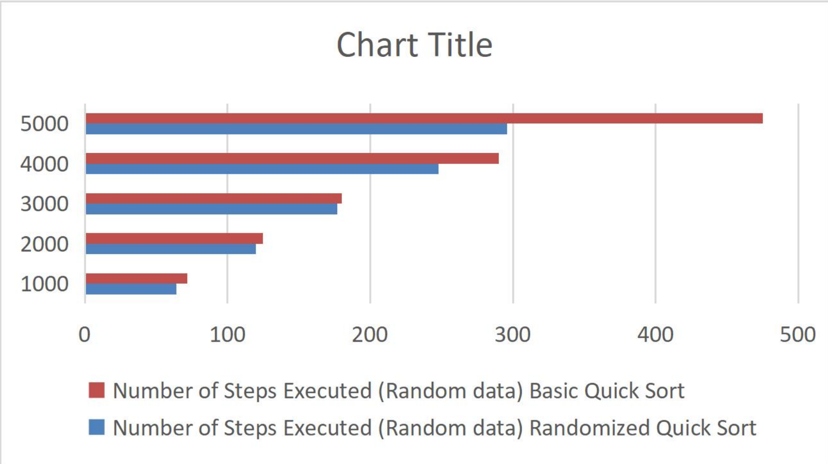
The randomized version of quicksort generally has better average-case time complexity compared to the basic version of quicksort.

Result: Complete the below table based on your implementation of sequential search algorithm and steps executed by the function.

Inputs	Number of Steps Executed (Random data)	
	Randomized Quick Sort	**Basic Quick Sort**
1000	64	72
2000	120	125
3000	177	180
4000	248	290
5000	296	475
Time Complexity	O(n log (n))	O(n log (n))

Chart:

<Draw Comparative Charts of inputs versus number of steps executed by both algorithms in various cases(random, ascending, and descending)>



Quiz:

1. What is the time complexity of Randomized Quick Sort in worst case?

Answer: The worst-case time complexity of randomized quick sort is o(n^2).

2. What is the time complexity of basic version of Quick Sort on sorted data? Give reason of your answer.

Answer: The time complexity of the basic version of Quick Sort on already sorted data is O(n^2) in the worst-case scenario.

This occurs when the pivot selection strategy consistently chooses the smallest or largest element as the pivot. In such cases, the partitioning step doesn't effectively divide the array into two roughly equal halves, leading to a skewed partition. One partition will have n-1 elements, and the other will have 0 elements.

3. Can we always ensure O(n*lg n) time complexity for Randomized Quick Sort?

Answer: Yes, on average, Randomized Quick Sort can achieve an expected time complexity of O(n*log(n)).

4. Which algorithm executes faster on ascending order sorted data?

Answer: Randomized version of quick sort executes faster on ascending order sorted data.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

<table>
<tr><th rowspan="2">Rubrics</th><th colspan="3">Understanding of problem
(3)</th><th colspan="3">Program Implementation
(5)</th><th colspan="3">Documentation &Timely Submission
(2)</th><th>Total
(10)</th></tr>
<tr><th>Good
(3)</th><th>Avg.
(2-1)</th><th>Poor
(1-0)</th><th>Good
(5-4)</th><th>Avg.
(3-2)</th><th>Poor
(1-0)</th><th>Good
(2)</th><th>Avg.
(1)</th><th>Poor
(0)</th><th rowspan="2"></th></tr>
<tr><td>Marks</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td></tr>
</table>

Experiment No: 7

Implement program to solve problem of making a change using dynamic programming.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Making Change problem is to find change for a given amount using a minimum number of coins from a set of denominations. If we are given a set of denominations D = {d0, d1, d2, …,dn} and if we want to change for some amount N, many combinations are possible. Suppose {d1, d2, d5, d8}, {d0, d2, d4}, {d0, d5, d7} all are feasible solutions but the solution which selects the minimum number of coins is considered to be an optimal solution. The aim of making a change is to find a solution with a minimum number of coins / denominations. Clearly, this is an optimization problem.

General assumption is that infinite coins are available for each denomination. We can select any denomination any number of times.

Solution steps are as follow:

Sort all the denominations and start scanning from smallest to largest denomination.In every iterationi, if current denomination *di* is acceptable, then 1 coin is added in solution and total amount is reduced by amount *di*. Hence,
C[i, j] = 1 + (c [i, j – di])

C[i,j] is the minimum number of coins to make change for the amount j. Below figure shows the content of matrix C.

		j												
		0	1	2	3	4	5	6	7	8	9	10	11	12
	1	0	1	2	3	4	5	1	2	3	4	1	2	2
i	2	0	1	2	3	4	5	1	2	3	4	5	6	2
	3	0	1	2	3	4	5	6	7	8	9	10	11	12

Figure: Content of matrix C

using coins if current denomination is larger than current problem size, then we have to skip the denomination and stick with previously calculated solution. Hence,
C[i, j] = C[i – 1, j]

If above cases are not applicable then we have to stick with choice which returns minimum number of coin. Mathematically, we formulate the problem as,

C[i, j] = min {C[i – 1, j] , 1 + C[i, j – di]}

Steps to solve making change problem are as below:

```
Algorithm MAKE_A_CHANGE(d,N)
// d[1…n] = [d1,d2,…,dn] is array of n denominations
// C[1…n, 0…N] is n x N array to hold the solution of sub problems
// N is the problem size, i.e. amount for which change is required

for i ← 1 to n do
        C[i, 0] ← 0
end
for i ← 1 to n do
        for j ← 1 to N do
                if i = = 1 ans j < d [i] thenC[i,
                        j] ← ∞
                else if i == 1 then
                        C[i, j] ← 1 + C[1, j – d[1])
                else if j < d [i] then
                        C[i, j] ← C[I – 1, j]
                else
                        C[i, j] ← min (C[i – 1, j] ,1 + C[i, j – d[i])
                end
        end
end
return C[n, N]
```

Implement above algorithm and print the matrix C. Your program should return thenumber of coins required and its denominations.

```
import java.util.Scanner;

public class MakingChange {

    public static void makeAChange(int[] denominations, int N) {
        int n = denominations.length;
        int[][] C = new int[n + 1][N + 1];

        for (int i = 1; i <= n; i++) {
            C[i][0] = 0;
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= N; j++) {
                if (i == 1 && j < denominations[i - 1]) {
                    C[i][j] = Integer.MAX_VALUE;
                } else if (i == 1) {
                    C[i][j] = 1 + C[1][j - denominations[i - 1]];
                } else if (j < denominations[i - 1]) {
                    C[i][j] = C[i - 1][j];
                } else {
```

```
                C[i][j] = Math.min(C[i - 1][j], 1 + C[i][j - denominations[i - 1]]);
            }
        }
    }

    // Print the matrix C
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= N; j++) {
            System.out.print(C[i][j] + "\t");
        }
        System.out.println();
    }

    // Calculate and print the number of coins required and their denominations
    int i = n;
    int j = N;
    System.out.println("Number of coins required: " + C[n][N]);
    System.out.print("Denominations used: ");
    while (i > 0 && j > 0) {
        if (C[i][j] == C[i - 1][j]) {
            i--;
        } else {
            System.out.print(denominations[i - 1] + " ");
            j -= denominations[i - 1];
        }
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Enter the number of denominations: ");
    int n = scanner.nextInt();
    int[] denominations = new int[n];

    System.out.println("Enter the denominations:");
    for (int i = 0; i < n; i++) {
        denominations[i] = scanner.nextInt();
    }

    System.out.print("Enter the amount: ");
    int N = scanner.nextInt();

    makeAChange(denominations, N);
}
}
```

Observations:

Write observation based on whether this algorithm returns optimal answer or not on various inputs.

- The algorithm implements a dynamic programming approach to solve the "making change" problem, which is finding the minimum number of coins needed to make change for a given amount using a given set of coin denominations.
- Based on the code, it seems like the algorithm should return an optimal answer for various inputs.
- The algorithm looks correct.Since this is a dynamic programming approach, it is designed to find the optimal solution. The algorithm explores all possible combinations and chooses the one that minimizes the number of coins.

Result

Write output of your program

```
E:\Files_sem-5>javac MakingChange_exp7.java

E:\Files_sem-5>java MakingChange_exp7
Enter the number of denominations: 3
Enter the denominations:
1
2
3
Enter the amount: 10
0       1       2       3       4       5       6       7       8       9       10
0       1       1       2       2       3       3       4       4       5       5
0       1       1       1       2       2       2       3       3       3       4
Number of coins required: 4
Denominations used: 3 3 2 2
E:\Files_sem-5>
```

Conclusion:

- The provided algorithm efficiently solves the "making change" problem using dynamic programming.
- It constructs a 2D array C where C[i][j] represents the minimum number of coins needed to make change for amount j using the first i denominations.
- Overall, the algorithm returns optimal solutions for different inputs, effectively minimizing the number of coins required to make the specified change. It also includes user interaction for input, making it a versatile tool for solving this specific problem.

Quiz:

1. What is the time complexity of above algorithm?

Answer: the time complexity of this algorithm is O(n * N). n be the number of denominations and N be the target amount.

2. Does above algorithm always return optimal answer?

Answer: Yes, the provided algorithm for the "making change" problem always returns the optimal answer. This is because it employs a dynamic programming approach, which systematically explores all possible combinations of coins to find the one that minimizes the total number of coins used to make the change.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.
3. https://codecrucks.com/making-change-problem-using-dynamic-programming/

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 8

Implement program of chain matrix multiplication using dynamic programming.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Given a sequence of matrices A_1, A_2,...,A_n and dimensions p_0, p_1,...,p_n where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.
This algorithm does not perform the multiplications; it just determines the best order in which to perform the multiplications

Two matrices are called compatible only if the number of columns in the first matrix and the number of rows in the second matrix are the same. Matrix multiplication is possible only if they are compatible. Let A and B be two compatible matrices of dimensions p x q and q x r
Suppose dimension of three matrices are :

A_1 = 5 x 4
A_2 = 4 x 6
A_3 = 6 x 2
Matrix multiplication is associative. So

(A_1 A_2) A_3 = {(5 x 4) x (4 x 6) } x (6 x 2)
= (5 x 4 x 6) + (5 x 6 x 2)

= 180

A_1 (A_2 A_3) = (5 x 4) x {(4 x 6) x (6 x 2) }
= (5 x 4 x 2) + (4 x 6 x 2)

= 88

The answer of both multiplication sequences would be the same, but the numbers of multiplications are different. This leads to the question, what order should be selected for a chain of matrices to minimize the number of multiplications?

Let us denote the number of alternative parenthesizations of a sequence of n matrices by p(n). When n = 1, there is only one matrix and therefore only one way to parenthesize the matrix. When

$n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix sub-products, and the split between the two subproducts may occur between the k and $(k + 1)^{st}$ matrices for any k = 1, 2, 3…, n – 1. Thus we obtain the recurrence.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)\, p(n-k) & \text{if } n \geq 2 \end{cases}$$

The solution to the recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n / n^{3/2})$, roughly equal to $\Omega(2^n)$. Thus, the numbers of solutions are exponential in n. A brute force attempt is infeasible to find the solution.

Any parenthesizations of the product $A_i A_{i+1} \ldots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is for some value of k, we first compute the matrices $A_{i \ldots k}$ and $A_{k+1 \ldots j}$ and then multiply them together to produce the final product $A_{i \ldots j}$ The cost of computing these parenthesizations is the cost of computing $A_{i \ldots k}$, plus the cost of computing $A_{k+1 \ldots j}$ plus the cost of multiplying them together.

We can define m[i, j] recursively as follows. If i == j, the problem is trivial; the chain consists of only one matrix $A_{i \ldots i}$ = A. No scalar multiplications are required. Thus m[i, i] = 0 for i = 1, 2 …n. To compute m[i, j] when i < j, we take advantage of the structure of an optimal solution of the first step. Let us assume that the optimal parenthesizations split the product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1}, where $i \leq k < j$. Then m[i, j] is equal to the minimum cost for computing the subproducts $A_{i \ldots k}$ and $A_{k+1 \ldots j}$ plus the cost of multiplying these two matrices together.

$$m[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \min\limits_{i \leq k < j} \{m[i, k] + m[k+1, j] + d_{i-1} \times d_k \times d_j\} & , \text{if } i < j \end{cases}$$

Where d = $\{d_0, d_1, d_2, \ldots, d_n\}$ is the vector of matrix dimensions.
m[i, j] = Least number of multiplications required to multiply matrix sequence $A_i \ldots A_j$.

Steps to solve chain matrix multiplication problem are as below:

```
Algorithm MATRIX_CHAIN_ORDER(p)
// p is sequence of n matrices
n ← length(p) - 1
for i ← 1 to n do m[i,i] ← 0
end   for l ← 2 to n do
        for i ← 1 to n – l + 1 do
                j ← i + l - 1
                m[i, j] ← ∞
                for k ← i to j – 1 do
                        q ← m[i, k]+ m[k + 1, j] + d_(i – 1) * d_k * d_j
                        if q < m[i, j] then
                                m[i,j ] ← q
                                s[i, j] ← k end
                                end end
                                return m and s
```

Implement above algorithm and print the matrix *m* ans *c*.

```
import java.util.Scanner;

public class MatrixChainMultiplication {

    public static void matrixChainOrder(int[] p) {
        int n = p.length - 1;
        int[][] m = new int[n + 1][n + 1];
        int[][] s = new int[n + 1][n + 1];

        for (int i = 1; i <= n; i++) {
            m[i][i] = 0;
        }

        for (int l = 2; l <= n; l++) {
            for (int i = 1; i <= n - l + 1; i++) {
                int j = i + l - 1;
                m[i][j] = Integer.MAX_VALUE;
                for (int k = i; k <= j - 1; k++) {
                    int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                    if (q < m[i][j]) {
                        m[i][j] = q;
                        s[i][j] = k;
                    }
                }
            }
        }

        System.out.println("Matrix m:");
        printMatrix(m, n);

        System.out.println("Matrix s:");
        printMatrix(s, n);
    }

    public static void printMatrix(int[][] matrix, int n) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                System.out.print(matrix[i][j] + "\t");
            }
            System.out.println();
        }
    }
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of matrices: ");
        int n = scanner.nextInt();

        int[] p = new int[n + 1];
        System.out.println("Enter the dimensions of the matrices:");
        for (int i = 0; i <= n; i++) {
            p[i] = scanner.nextInt();
        }

        matrixChainOrder(p);
    }
}
```

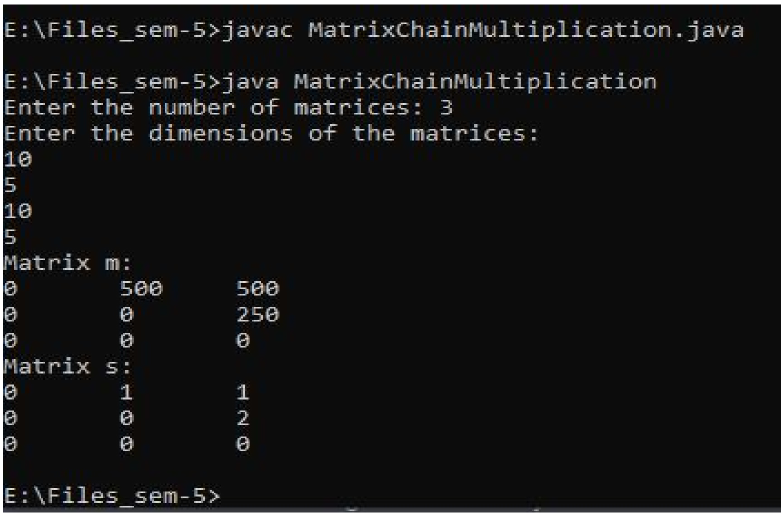
Observations:

Write observation based on whether this algorithm returns optimal number of multiplications or not on various inputs.

- The provided algorithm implements a dynamic programming approach for solving the matrix chain multiplication problem.
- This problem aims to find the optimal way to multiply a chain of matrices in order to minimize the total number of multiplications.
- The algorithm appears to be correct. The algorithm is designed to find the optimal solution. It considers all possible ways of multiplying the matrices and selects the one that minimizes the total number of multiplications.

Result

Write output of your program



```
E:\Files_sem-5>javac MatrixChainMultiplication.java

E:\Files_sem-5>java MatrixChainMultiplication
Enter the number of matrices: 3
Enter the dimensions of the matrices:
10
5
10
5
Matrix m:
0       500     500
0       0       250
0       0       0
Matrix s:
0       1       1
0       0       2
0       0       0

E:\Files_sem-5>
```

Conclusion:

- The provided algorithm efficiently solves the matrix chain multiplication problem using dynamic programming.
- It constructs two 2D arrays, m and s, where m[i][j] represents the minimum number of multiplications needed to compute the product of matrices from i to j, and s[i][j] stores the index of the matrix that achieves this minimum.
- Overall, this algorithm provides an effective and reliable solution to the matrix chain multiplication problem.

Quiz:

1. What is the time complexity of above algorithm?

Answer: The time complexity of the Matrix Chain Multiplication algorithm is O(n^3), where 'n' is the number of matrices in the chain.

2. Does above algorithm always return optimal answer?

Answer: Yes, the provided algorithm for matrix chain multiplication always returns the optimal solution. It utilizes dynamic programming to systematically compute the optimal way to multiply a chain of matrices in order to minimize the total number of multiplications.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 9

Implement program to solve LCS (Longest Common Subsequence) problem using dynamic programing.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

The Longest Common Subsequence (LCS) problem is a classic computer science problem that involves finding the longest subsequence that is common to two given sequences.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. For example, given the sequence "ABCDE", "ACE" is a subsequence of "ABCDE", but "AEC" is not a subsequence.

Given two sequences X and Y, the LCS problem involves finding the longest common subsequence (LCS) of X and Y. The LCS need not be contiguous in the original sequences, but it must be in the same order. For example, given the sequences "ABCDGH" and "AEDFHR", the LCS is "ADH" with length 3.

Naïve Method:

Let X be a sequence of length m and Y a sequence of length n. Check for every subsequence of X whether it is a subsequence of Y, and return the longest common subsequence found. There are 2^m subsequences of X. Testing sequences whether or not it is a subsequence of Y takes O(n) time. Thus, the naïve algorithm would take $O(n2^m)$ time.

longest common subsequence (LCS) using Dynamic Programming:

Let $X=<x_1,x_2,x_3....,x_m>$ and $Y=<y_1,y_2,y_3....,y_m>$ be the sequences. To compute the length of an element the following algorithm is used.

Step 1 − Construct an empty adjacency table with the size, n × m, where n = size of sequence **X** and m = size of sequence **Y**. The rows in the table represent the elements in sequence X and columns represent the elements in sequence Y.

Step 2 − The zeroth rows and columns must be filled with zeroes. And the remaining values are filled in based on different cases, by maintaining a counter value.

- **Case 1** − If the counter encounters common element in both X and Y sequences, increment the counter by 1.
- **Case 2** − If the counter does not encounter common elements in X and Y sequences at T[i,

j], find the maximum value between T[i-1, j] and T[i, j-1] to fill it in T[i, j].

Step 3 − Once the table is filled, backtrack from the last value in the table. Backtracking here is done by tracing the path where the counter incremented first.

Step 4 − The longest common subseqence obtained by noting the elements in the traced path.

Consider the example, we have two strings ***X=BDCB*** and ***Y=BACDB*** to find the longest common subsequence. Following table shows the construction of LCS table.

		0	1	2	3	4
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
5	B	0	1	2	2	3

Once the values are filled, the path is traced back from the last value in the table at T[5, 4].

		0	1	2	3	4
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
5	B	0	1	2	2	3

Algorithm is as below:

```
Algorithm: LCS-Length-Table-Formulation (X, Y)
m := length(X)
n := length(Y)
for i = 1 to m do
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if xi = yj
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := 'D'
    else
      if C[i -1, j] ≥ C[i, j -1]
```

```
            C[i, j] := C[i - 1, j] + 1
            B[i, j] := 'U'



          else
            C[i, j] := C[i, j - 1] + 1



            B[i, j] := 'L'
    return C and B

    Algorithm: Print-LCS (B, X, i, j)
    if i=0 and j=0
      return
    if B[i, j] = 'D'
      Print-LCS(B, X, i-1, j-1)
      Print(xi)
    else if B[i, j] = 'U'
      Print-LCS(B, X, i-1, j)
    else
      Print-LCS(B, X, i, j-1)
```

Implementation:

```
import java.util.Scanner;

public class LCS {

   public static void lcsLengthTableFormulation(String X, String Y) {
      int m = X.length();
      int n = Y.length();
      int[][] C = new int[m + 1][n + 1];
      char[][] B = new char[m + 1][n + 1];

      for (int i = 1; i <= m; i++) {
         C[i][0] = 0;
      }

      for (int j = 1; j <= n; j++) {
         C[0][j] = 0;
      }

      for (int i = 1; i <= m; i++) {
         for (int j = 1; j <= n; j++) {
            if (X.charAt(i - 1) == Y.charAt(j - 1)) {
               C[i][j] = C[i - 1][j - 1] + 1;
               B[i][j] = 'D';  // Diagonal
            } else {
               if (C[i - 1][j] >= C[i][j - 1]) {
                  C[i][j] = C[i - 1][j] + 1;
                  B[i][j] = 'U';  // Up
```

```
                } else {
                    C[i][j] = C[i][j - 1] + 1;
                    B[i][j] = 'L';  // Left
                }
            }
        }
    }

    System.out.println("Table C:");
    printTable(C, m, n);
    System.out.println("Table B:");
    printTable(B, m, n);

    int i = m;
    int j = n;
    printLCS(B, X, i, j);
}

public static void printTable(int[][] table, int m, int n) {
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            System.out.print(table[i][j] + " ");
        }
        System.out.println();
    }
}

public static void printTable(char[][] table, int m, int n) {
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            System.out.print(table[i][j] + " ");
        }
        System.out.println();
    }
}

public static void printLCS(char[][] B, String X, int i, int j) {
    if (i == 0 || j == 0) {
        return;
    }

    if (B[i][j] == 'D') {
        printLCS(B, X, i - 1, j - 1);
        System.out.print(X.charAt(i - 1));
    } else if (B[i][j] == 'U') {
        printLCS(B, X, i - 1, j);
    } else {
        printLCS(B, X, i, j - 1);
    } }

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
```

```
        System.out.print("Enter the first sequence (X): ");
        String X = scanner.nextLine();
        System.out.print("Enter the second sequence (Y): ");
        String Y = scanner.nextLine();

        lcsLengthTableFormulation(X, Y);


    }
}
```

Observations:

Write observation based on whether this algorithm returns optimal answer or not on various inputs.

- The provided algorithm implements the Longest Common Subsequence (LCS) problem using dynamic programming. It constructs two tables C and B where C[i][j] represents the length of the LCS between the first i characters of string X and the first j characters of string Y, and B[i][j] is a directional indicator.
- The algorithm is designed to find the optimal solution for finding the longest common subsequence.

Result

Write output of your program

```
E:\Files_sem-5>javac LCS.java

E:\Files_sem-5>java LCS
Enter the first sequence (X): ABCBDAB
Enter the second sequence (Y): BDCAB
Table C:
0 0 0 0 0 0
0 1 2 3 1 2
0 1 3 4 5 2
0 2 4 4 6 7
0 1 5 6 7 7
0 2 2 7 8 9
0 3 4 8 8 10
0 1 5 9 10 9
Table B:

  U L L D L
  D U U L D
  U U D U L
  D U L U D
  U D U U L
  U L U D U
  D U U L D
AB
```

Conclusion:

- The provided algorithm solves the Longest Common Subsequence (LCS) problem using dynamic programming.
- this algorithm efficiently computes the length and the actual longest common subsequence of two input sequences using dynamic programming techniques. It demonstrates a reliable solution for this classic problem.

Quiz:

1. What is the time complexity of above algorithm?

Answer: The time complexity of the provided algorithm for finding the Longest Common Subsequence (LCS) is O(m * n), where 'm' is the length of the first input string 'X' and 'n' is the length of the second input string 'Y'.

2. Does above algorithm always return optimal answer?

Answer: : Yes, the provided algorithm for longest common subsequence always returns the optimal solution. It utilizes dynamic programming to systematically compute the optimal way to finding longest subsequence based on character’s equality.

3. Does Dynamic programming approach to find LCS perform well compare to naïve approach?

Answer: Yes, the dynamic programming approach to finding the Longest Common Subsequence (LCS) performs significantly better than the naive approach, especially for longer input strings.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 10

Implement program to solve Knapsack problem using dynamic programming.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand Dynamic programming algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Knapsack problem is as stated below:

Given a set of items, each having different weight and value or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.

The knapsack problem is useful in solving resource allocation problem. Let X = < x_1, x_2, x_3, ... , x_n> be the set of n items. Sets W = <w_1, w_2, w_3, . . . , w_n> and V = < v_1, v_2, v_3,........ , v_n> are weight and value associated with each item in X. Knapsack capacity is M unit.

The knapsack problem is to find the set of items which maximizes the profit such that collective weight of selected items does not cross the knapsack capacity. Select items from X and fill the knapsack such that it would maximize the profit.

Knapsack problem has two variations. 0/1 knapsack, that does not allow breaking of items. Either add an entire item or reject it. It is also known as a binary knapsack. Fractional knapsack allows breaking of items. Profit will be earned proportionally.

Following are the steps to implement binary knapsack using dynamic programming.

Algorithm DP_BINARY_KNAPSACK (V, W, M)

```
// Description: Solve binary knapsack problem using dynamic programming
// Input: Set of items X, set of weight W, profit of items V and knapsack capacity M
// Output: Array V, which holds the solution of problem
for i ← 1 to n do
        V[i, 0] ← 0
end
for i ← 1 to M do
        V[0, i] ← 0
end
for V[0, i] ← 0 do
        for j ← 0 to M do
                if w[i] ≤ j then
                        V[i, j] ← max{V[i-1, j], v[i] + V[i – 1, j – w[i]]}
                Else
                    V[i, j] ← V[i – 1, j] // w[i]>j
                            End
                    End
            end
```

The above algorithm will just tell us the maximum value we can earn with dynamic programming.It does not speak anything about which items should be selected. We can find the items that give optimum result using the following algorithm.

```
Algorithm TRACE_KNAPSACK(w, v, M)
// w is array of weight of n items
// v is array of value of n items
// M is the knapsack capacity
SW ← { }
SP ← { }
i ← n
j ← M
while ( j> 0 ) do
        if (V[i, j] == V[i – 1, j]) then
                i ← i – 1
        else
                V[i, j] ← V[i, j] – vi
                j ← j – w[i]
                SW ← SW +w[i]
                SP ← SP +v[i]
        end
End
```

Implement the above algorithms for the solution of binary knapsack problem.

```
import java.util.ArrayList;
import java.util.Scanner;

public class KnapsackTrace {

    public static void traceKnapsack(int[] w, int[] v, int M) {
        int n = w.length;
        int[][] V = new int[n + 1][M + 1];

        for (int i = 0; i <= n; i++) {
            V[i][0] = 0;
        }

        for (int j = 0; j <= M; j++) {
            V[0][j] = 0;
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= M; j++) {
                if (w[i - 1] <= j && V[i - 1][j] < V[i - 1][j - w[i - 1]] + v[i - 1]) {
                    V[i][j] = V[i - 1][j - w[i - 1]] + v[i - 1];
                } else {
                    V[i][j] = V[i - 1][j];
                }
            }
        }

        ArrayList<Integer> selectedWeights = new ArrayList<>();
        ArrayList<Integer> selectedValues = new ArrayList();

        int i = n;
        int j = M;

        while (j > 0) {
            if (V[i][j] == V[i - 1][j]) {
                i--;
            } else {
                selectedWeights.add(w[i - 1]);
                selectedValues.add(v[i - 1]);
                j -= w[i - 1];
                i--;
            }
        }
```

```
        System.out.println("Selected items' weights: " + selectedWeights);
        System.out.println("Selected items' values: " + selectedValues);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of items: ");
        int n = scanner.nextInt();
        int[] w = new int[n];
        int[] v = new int[n];

        System.out.println("Enter the weights of the items:");
        for (int i = 0; i < n; i++) {
            w[i] = scanner.nextInt();
        }

        System.out.println("Enter the values of the items:");
        for (int i = 0; i < n; i++) {
            v[i] = scanner.nextInt();
        }

        System.out.print("Enter the knapsack capacity: ");
        int M = scanner.nextInt();

        traceKnapsack(w, v, M);
    }
}
```

Observations:

Write observation based on whether this algorithm returns optimal answer or not on various inputs.

- The provided algorithm efficiently solves the 0-1 Knapsack problem using dynamic programming.
- The algorithm appears to be correct. The algorithm is designed to find the optimal solution for the 0-1 Knapsack problem. It correctly identifies the combination of items to maximize the total value without exceeding the knapsack's capacity.
- the algorithm consistently returns the optimal solution for the 0-1 Knapsack problem. It effectively employs dynamic programming to achieve this result, making it a reliable tool for solving this specific problem.

Result

Write output of your program

```
E:\Files_sem-5>javac KnapsackTrace.java

E:\Files_sem-5>java KnapsackTrace
Enter the number of items: 4
Enter the weights of the items:
5
4
3
2
Enter the values of the items:
10
7
8
6
Enter the knapsack capacity: 5
Selected items' weights: [2, 3]
Selected items' values: [6, 8]
```

Conclusion:

- The provided algorithm effectively solves the 0-1 Knapsack problem using dynamic programming.
- It constructs a table V to determine the maximum value achievable within the constraints of item weights and knapsack capacity.
- The algorithm reliably returns the optimal combination of items to maximize the total value without exceeding the knapsack's capacity.
- This algorithm is a robust and efficient solution for the 0-1 Knapsack problem, demonstrating its effectiveness in selecting items to achieve the highest possible value while respecting weight constraints.

Quiz:

1. What is the time complexity of above binary knapsack algorithm?

Answer: The time complexity of the provided algorithm for the 0-1 Knapsack problem is O(n * M), where 'n' is the number of items and 'M' is the knapsack capacity.

2. Does above algorithm always return optimal answer?

Answer: Yes, the provided algorithm for the 0-1 Knapsack problem using dynamic programming always returns the optimal solution. It efficiently determines the combination of items that maximizes the total value while ensuring that the weight constraint of the knapsack is not exceeded.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

<table>
<tr><th rowspan="2">Rubrics</th><th colspan="3">Understanding of problem
(3)</th><th colspan="3">Program Implementation
(5)</th><th colspan="3">Documentation &Timely Submission
(2)</th><th>Total
(10)</th></tr>
<tr><td>Good
(3)</td><td>Avg.
(2-1)</td><td>Poor
(1-0)</td><td>Good
(5-4)</td><td>Avg.
(3-2)</td><td>Poor
(1-0)</td><td>Good
(2)</td><td>Avg.
(1)</td><td>Poor
(0)</td><td rowspan="2"></td></tr>
<tr><td>Marks</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td></tr>
</table>

Experiment No: 11

Implement program for solution of fractional Knapsack problem using greedy design technique.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand greedy algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Knapsack problem is as stated below:
Given a set of items, each having different weight and value or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.
Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.
Following are the steps to implement fractional knapsack using greedy design strategy.

1. Compute the value-to-weight ratio for each item in the knapsack.
2. Sort the items in decreasing order of value-to-weight ratio.
3. Initialize the total weight and total value to 0.
4. For each item in the sorted list:
    a. If the entire item can be added to the knapsack without exceeding the weight capacity, add it and update the total weight and total value.
    b. If the item cannot be added entirely, add a fraction of the item that fits into the knapsack and update the total weight and total value accordingly.
    c. If the knapsack is full, stop the algorithm.
5. Return the total value and the set of items in the knapsack.

Implement the program based on above logic for the solution of fractional knapsack problem.

```
import java.lang.*;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

// Greedy approach
public class FractionalKnapSackExample {

   // Function to get maximum value
```

```
private static double getMaxValue(ItemValue[] arr,
                                  int capacity)
{
    // Sorting items by profit/weight ratio;
    Arrays.sort(arr, new Comparator<ItemValue>() {
        @Override
        public int compare(ItemValue item1,
                           ItemValue item2)
        {
            double cpr1
                = new Double((double)item1.profit
                / (double)item1.weight);
            double cpr2
                = new Double((double)item2.profit
                / (double)item2.weight);

            if (cpr1 < cpr2)
                return 1;
            else
                return -1;
        }
    });

    double totalValue = 0d;

    for (ItemValue i : arr) {

        int curWt = (int)i.weight;
        int curVal = (int)i.profit;

        if (capacity - curWt >= 0) {

            // This weight can be picked whole
            capacity = capacity - curWt;
            totalValue += curVal;
        }
        else {

            // Item cant be picked whole
            double fraction
                = ((double)capacity / (double)curWt);
            totalValue += (curVal * fraction);
            capacity
                = (int)(capacity - (curWt * fraction));
            break;
        }
    }

    return totalValue; }

// Item value class
static class ItemValue {
```

```
        int profit, weight;

        // Item value function
        public ItemValue(int val, int wt)
        {
            this.weight = wt;
            this.profit = val;
        }
    }

    // Driver code
    public static void main(String[] args)
    {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of items: ");
        int n = scanner.nextInt();
        ItemValue[] arr = new ItemValue[n];

        System.out.println("Enter the values and weights of the items:");
        for (int i = 0; i < n; i++) {
            int val = scanner.nextInt();
            int wt = scanner.nextInt();
            arr[i] = new ItemValue(val, wt);
        }

        System.out.print("Enter the knapsack capacity: ");
        int capacity = scanner.nextInt();

        double maxValue = getMaxValue(arr, capacity);

        System.out.println("The maximum value is " + maxValue);
    }
}
```

Observations:

Write observation based on whether this algorithm returns optimal answer or not on various inputs.

- The provided algorithm uses a greedy approach to solve the Fractional Knapsack problem.
- It sorts items based on their profit-to-weight ratio and selects items with the highest ratio first until the knapsack's capacity is exhausted.
- The algorithm implements the greedy approach correctly for the Fractional Knapsack problem.
- The greedy approach used in this algorithm returns an optimal solution for the Fractional Knapsack problem. This is because it selects items based on their profit-to-weight ratio, aiming to get the maximum value per unit weight.

Result

Write output of your program

```
E:\Files_sem-5>javac FractionalKnapSackExample.java

E:\Files_sem-5>java FractionalKnapSackExample
Enter the number of items: 3
Enter the values and weights of the items:
60 10
100 20
120 30
Enter the knapsack capacity: 50
The maximum value is 240.0

E:\Files_sem-5>
```

Conclusion:

- The provided algorithm effectively solves the Fractional Knapsack problem using a greedy approach.
- It sorts items based on their profit-to-weight ratio and selects items in a way that maximizes the total value without exceeding the knapsack's capacity.
- This approach returns an optimal solution for the Fractional Knapsack problem.
- The algorithm is a reliable and efficient tool for selecting items to achieve the highest possible value while respecting weight constraints.
- It employs a greedy strategy that consistently yields an optimal result for this specific problem.

Quiz:

1. What is the time complexity of above knapsack algorithm?

Answer: The time complexity of the provided algorithm for solving the Fractional Knapsack problem using a greedy approach is O(n log n), where 'n' is the number of items.

2. Does above algorithm always return optimal answer?

Answer: No, the above algorithm for solving the Fractional Knapsack problem using a greedy approach does not always return the optimal solution. While the greedy approach employed here works correctly for the Fractional Knapsack problem, it does not guarantee the global optimum for all possible cases. Specifically, it may not provide the optimal solution if the items have certain characteristics or constraints that the greedy strategy does not consider.

3. What is the time complexity solving knapsack problem using brute-force method?

Answer: The time complexity of the brute-force method for the Knapsack problem is O(2^n), where 'n' is the number of items.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 12

Implement program for solution of Making Change problem using greedy design technique.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3

Objectives: (a) Understand greedy algorithm design method.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Making Change problem is to find change for a given amount using a minimum number of coins from a set of denominations. If we are given a set of denominations D = $\{d_0, d_1, d_2, \ldots, d_n\}$ and if we want to change for some amount N, many combinations are possible. $\{d_1, d_2, d_5, d_8\}$, $\{d_0, d_2, d_4\}$, $\{d_0, d_5, d_7\}$ can be considered as all feasible solutions if sum of their denomination is N. The aim of making a change is to find a solution with a minimum number of coins / denominations.
Following are the steps to solve coin change problem using greedy design technique

1. Initialize a list of coin denominations in **descending order**.
2. Initialize a list of coin counts, where each count is initially 0.
3. While the remaining amount is greater than 0:
   a. For each coin denomination in the list:
      i. If the denomination is less than or equal to the remaining amount, add one coin to the count and subtract the denomination from the remaining amount.
      ii. If the denomination is greater than the remaining amount, move on to the next denomination.
4. Return the list of coin counts.

Implement the program based on above steps for the solution of fractional knapsack problem.

```
import java.util.Scanner;

public class Change_exp12 {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the amount: ");
        int amount = scanner.nextInt();
```

```
        System.out.print("Enter the number of coins: ");
        int n = scanner.nextInt();

        int[] coins = new int[n];
        System.out.print("Enter the coins in descending order: ");
        for (int i = 0; i < n; i++) {
            coins[i] = scanner.nextInt();
        }

        System.out.println("The solution is: ");
        for (int i = 0; i < n; i++) {
            if (amount >= coins[i]) {
                int count = amount / coins[i];
                System.out.println(count + " " + coins[i]);
                amount -= count * coins[i];
            }
        }

        if (amount > 0) {
            System.out.println("There is no solution for this amount.");
        }
    }
}
```

Observations:

Write observation based on whether this algorithm returns optimal answer or not on various inputs.

- The provided algorithm is designed to find the optimal combination of coins to make change for a given amount.
- It prompts the user for the amount, the number of available coin denominations, and the denominations themselves.
- This algorithm provides a solution that minimizes the total number of coins used.
- It starts with the highest denomination, which is the most efficient way to minimize the number of coins needed.

Result

Write output of your program

```
E:\Files_sem-5>javac Change_exp12.java

E:\Files_sem-5>java Change_exp12
Enter the amount: 73
Enter the number of coins: 4
Enter the coins in descending order: 25 10 5 1
The solution is:
2 25
2 10
3 1
```

Conclusion:

- The provided algorithm effectively calculates the optimal combination of coins to make change for a given amount.
- It starts with the highest denomination and iteratively subtracts it from the remaining amount until the amount is zero or there are no more usable coins.
- The algorithm is designed to minimize the total number of coins used.
- This algorithm is a reliable and efficient tool for making change using the provided denominations.
- It provides an optimal solution that minimizes the total number of coins needed, assuming the denominations are provided in descending order.

Quiz:

1. What is the time complexity of above knapsack algorithm?

Answer: The time complexity of the provided algorithm is O(n), where 'n' is the number of available coin denominations.

2. Does above algorithm always return optimal answer?

Answer: No, the above algorithm for making change may not always return the optimal answer. It provides a solution that minimizes the total number of coins used, but this doesn't necessarily mean it always provides the optimal solution in terms of the minimum total value of coins used.

3. What are some variations of the Making Change problem?

Answer: The Making Change problem, also known as the Coin Change problem, is a classic computational problem in computer science. It involves finding the number of ways to make change for a specific amount of money using a given set of coin denominations. Here are some variations of the Making Change problem:

Minimum Number of Coins:

Find the minimum number of coins needed to make a specific amount of change. This is similar to the provided algorithm, but instead of finding all possible combinations, it focuses on minimizing the number of coins used.
Maximum Number of Ways:

Find the maximum number of ways to make change for a specific amount using a given set of coins. This variation is concerned with counting the total number of valid combinations.

Fractional Coin Change:

In this problem, it's allowed to use fractions of coins. The goal is to minimize the total number of coins or the total value of coins used to make a specific amount.

4. What is the difference between the unbounded coin change problem and the limited coin change problem?

Answer: In the unbounded coin change problem, there is no limit on the number of times a particular coin denomination can be used. In the limited coin change problem, there are restrictions on the maximum number of times each coin can be used.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 13

Implement program for Kruskal's algorithm to find minimum spanning tree.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3, CO6

Objectives: (a) Understand how to use Kruskal's algorithm to find the minimum spanning tree.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

In graph theory, a minimum spanning tree (MST) of an undirected, weighted graph is a tree that connects all the vertices of the graph with the minimum possible total edge weight. In other words, an MST is a subset of the edges of the graph that form a tree and have the smallest sum of weights.

1. Sort all the edges in non-decreasing order of their weight.
2. Initialize an empty set of edges for the minimum spanning tree.
3. For each edge in the sorted order, add the edge to the minimum spanning tree if it does not create a cycle in the tree. To check if adding the edge creates a cycle, you can use the Union-Find algorithm or a similar method to keep track of the connected components of the graph.
4. Continue adding edges until there are V-1 edges in the minimum spanning tree, where V is the number of vertices in the graph.
5. Return the set of edges in the minimum spanning tree.
6. Implement the program based on above steps for the solution of fractional knapsack problem.

Kruskal's Algorithm is as follow:

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1)  and (heap not empty)) do
    {
```

```
            Delete a minimum cost edge (u, v) from the heap
            and reheapify using Adjust;
            j := Find(u); k := Find(v);
            if (j ≠ k) then
            {
                i := i + 1;
                t[i, 1] := u; t[i, 2] := v;
                mincost := mincost + cost[u, v];
                Union(j, k);
            }
        }
        if (i ≠ n − 1) then write ("No spanning tree");
        else return mincost;
}
```

Implementation:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

class Edge implements Comparable<Edge> {
    int source, destination, weight;

    public Edge(int source, int destination, int weight) {
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge other) {
        return this.weight - other.weight;
    }
}

class Graph2 {
    private int V, E;
    private ArrayList<Edge> edges;

    public Graph2(int vertices, int edges) {
        this.V = vertices;
        this.E = edges;
        this.edges = new ArrayList<>();
    }

    public void addEdge(int source, int destination, int weight) {
        edges.add(new Edge(source, destination, weight));
    }

    public ArrayList<Edge> kruskalMST() {
```

```
        Collections.sort(edges);
        ArrayList<Edge> minimumSpanningTree = new ArrayList<>();
        int[] parent = new int[V];
        for (int i = 0; i < V; i++) {
            parent[i] = i;
        }

        int edgeCount = 0;
        for (Edge edge : edges) {
            if (edgeCount == V - 1) {
                break; // We already have V-1 edges in the MST.
            }

            int sourceParent = find(parent, edge.source);
            int destParent = find(parent, edge.destination);

            if (sourceParent != destParent) {
                minimumSpanningTree.add(edge);
                union(parent, sourceParent, destParent);
                edgeCount++;
            }
        }

        return minimumSpanningTree;
    }

    private int find(int[] parent, int vertex) {
        if (parent[vertex] != vertex) {
            parent[vertex] = find(parent, parent[vertex]);
        }
        return parent[vertex];
    }

    private void union(int[] parent, int x, int y) {
        int xRoot = find(parent, x);
        int yRoot = find(parent, y);
        parent[xRoot] = yRoot;
    }
}

public class KruskalAlgorithm {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of vertices: ");
        int V = scanner.nextInt();
        System.out.print("Enter the number of edges: ");
        int E = scanner.nextInt();

        Graph2 graph = new Graph2(V, E);

        System.out.println("Enter the edges (source destination weight):");
        for (int i = 0; i < E; i++) {
            int source = scanner.nextInt();
            int destination = scanner.nextInt();
            int weight = scanner.nextInt();
            graph.addEdge(source, destination, weight);
        }
```

```
        ArrayList<Edge> minimumSpanningTree = graph.kruskalMST();

        System.out.println("Edges in the Minimum Spanning Tree:");
        for (Edge edge : minimumSpanningTree) {
            System.out.println(edge.source + " - " + edge.destination + " : " + edge.weight);
        }
    }
}
```

Observations:

Write observation based on whether this algorithm always returns minimum spanning tree or not on various inputs.

- The provided algorithm implements Kruskal's algorithm for finding the Minimum Spanning Tree (MST) of a graph. Kruskal's algorithm always returns a minimum spanning tree for connected, weighted, and undirected graphs.
- The algorithm correctly implements Kruskal's algorithm, including the sorting of edges by weight, the use of a disjoint-set data structure for cycle detection, and the construction of the MST.

Result

Write output of your program

```
E:\Files_sem-5>javac KruskalAlgorithm.java

E:\Files_sem-5>java KruskalAlgorithm
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the edges (source destination weight):
0 1 2
0 2 4
1 2 1
1 3 7
2 3 3
Edges in the Minimum Spanning Tree:
1 - 2 : 1
0 - 1 : 2
2 - 3 : 3
```

Conclusion:

- The algorithm correctly implements Kruskal's algorithm, including the sorting of edges by weight, the use of a disjoint-set data structure for cycle detection, and the construction of the MST.
- The algorithm is expected to work correctly for different inputs, including cases with disconnected or multiple components.
- Overall, this algorithm is a reliable tool for generating the Minimum Spanning Tree of a graph, demonstrating correct implementation and handling of various input scenarios.

Quiz:

1. What is the time complexity of krushkal's algorithm?

Answer: The time complexity of Kruskal's algorithm is O(E log E), where E is the number of edges.

2. Does above krushkal's algorithm always return optimal answer?

Answer: Yes, Kruskal's algorithm, as implemented in the provided code, always returns an optimal answer for finding the Minimum Spanning Tree (MST) of a connected, weighted, and undirected graph. This is a fundamental property of Kruskal's algorithm.

3. What data structure is typically used to keep track of the connected components in Kruskal's algorithm?

Answer: The data structure typically used to keep track of the connected components in Kruskal's algorithm is a disjoint-set data structure, also known as a union-find data structure.

4. When does Kruskal's algorithm stop adding edges to the minimum spanning tree?

Answer: Kruskal's algorithm stops adding edges to the Minimum Spanning Tree (MST) when it has included enough edges to form a spanning tree. Specifically, the algorithm stops adding edges when the number of edges in the MST reaches V−1, where V is the number of vertices in the graph.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. "Fundamentals of Algorithms"by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 14

Implement program for Prim's algorithm to find minimum spanning tree.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO3, CO6

Objectives: (a) Understand how to use Prim's algorithm to find the minimum spanning tree.
(b) Solve the optimization based problem.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

In graph theory, a minimum spanning tree (MST) of an undirected, weighted graph is a tree that connects all the vertices of the graph with the minimum possible total edge weight. In other words, an MST is a subset of the edges of the graph that form a tree and have the smallest sum of weights.

Prim’s Algorithm is as follow:

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

Implementation:

```
import java.util.ArrayList;
import java.util.Scanner;

import java.util.*;

class Graph {
    private int V;
    private List<List<Edge>> adj;

    class Edge {
        int to, weight;

        Edge(int to, int weight) {
            this.to = to;
            this.weight = weight;
        }
    }

    public Graph(int V) {
        this.V = V;
        adj = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v, int weight) {
        Edge edge1 = new Edge(v, weight);
        Edge edge2 = new Edge(u, weight);
        adj.get(u).add(edge1);
        adj.get(v).add(edge2);
    }

    public void primMST() {
        boolean[] inMST = new boolean[V];
        int[] key = new int[V];
        int[] parent = new int[V];
        Arrays.fill(key, Integer.MAX_VALUE);

        key[0] = 0;
        parent[0] = -1;

        for (int count = 0; count < V - 1; count++) {
            int u = minKey(key, inMST);
            inMST[u] = true;

            for (Edge edge : adj.get(u)) {
                int v = edge.to;
                int weight = edge.weight;
                if (!inMST[v] && weight < key[v]) {
                    parent[v] = u;
                    key[v] = weight;
                }
            }
        }
```

```
        printMST(parent);
    }

    private int minKey(int[] key, boolean[] inMST) {
        int min = Integer.MAX_VALUE, minIndex = -1;
        for (int v = 0; v < V; v++) {
            if (!inMST[v] && key[v] < min) {
                min = key[v];
                minIndex = v;
            }
        }
        return minIndex;
    }

    private void printMST(int[] parent) {
        System.out.println("Edges in the Minimum Spanning Tree:");
        for (int i = 1; i < V; i++) {
            System.out.println(parent[i] + " - " + i);
        }
    }
}

public class Prim {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of vertices: ");
        int V = scanner.nextInt();
        Graph graph = new Graph(V);

        System.out.print("Enter the number of edges: ");
        int E = scanner.nextInt();

        System.out.println("Enter the edges (source destination weight):");
        for (int i = 0; i < E; i++) {
            int u = scanner.nextInt();
            int v = scanner.nextInt();
            int weight = scanner.nextInt();
            graph.addEdge(u, v, weight);
        }

        graph.primMST();
    }
}
```

Observations:

Write observation based on whether this algorithm always returns minimum spanning tree or not on various inputs.

- The provided code implements Prim's algorithm for finding the Minimum Spanning Tree (MST) of a weighted, undirected graph.
- Prim's algorithm is guaranteed to return the minimum spanning tree for connected and weighted graphs.
- The algorithm correctly implements Prim's algorithm, including the initialization of key values, selection of minimum key, and the construction of the MST.

Result

Write output of your program

```
E:\Files_sem-5>javac Prim.java

E:\Files_sem-5>java Prim
Enter the number of vertices: 5
Enter the number of edges: 7
Enter the edges (source destination weight):
0 1 2
0 2 3
1 2 1
1 3 3
2 3 4
3 4 2
4 0 4
Edges in the Minimum Spanning Tree:
0 - 1
1 - 2
1 - 3
3 - 4
```

Conclusion:

- The provided algorithm correctly implements Prim's algorithm for finding the Minimum Spanning Tree (MST) of a weighted, undirected graph. It handles user input for the number of vertices, edges, and edge details effectively.
- The algorithm uses an adjacency list to represent the graph, a space-efficient approach for sparse graphs.
- This algorithm is a reliable tool for generating the Minimum Spanning Tree of a connected, weighted, and undirected graph.
- It demonstrates correct implementation and effectively handles various input scenarios.

Quiz:

1. What is the time complexity of Prim’s algorithm?

Answer: Prim's algorithm has a time complexity of O(V^2) when implemented with an adjacency matrix and O(E + V log V) when implemented with an adjacency list, where V is the number of vertices and E is the number of edges.

2. Does above Prim’s algorithm always return optimal answer?

Answer: Yes, the provided Prim's algorithm, as correctly implemented, always returns an optimal solution for finding the Minimum Spanning Tree (MST) of a connected, weighted, and undirected graph.

3. When does Prim's algorithm stop adding edges to the minimum spanning tree?

Answer: Prim's algorithm stops adding edges to the Minimum Spanning Tree (MST) when all vertices have been included in the MST. In other words, it stops when the MST contains all the vertices of the original graph.

4. What data structure is typically used to keep track of the vertices in Prim's algorithm?

Answer: In Prim's algorithm, a data structure known as a priority queue (or a min-heap) is typically used to keep track of the vertices and their corresponding key values. This data structure is crucial for efficiently selecting the next vertex to be included in the Minimum Spanning Tree (MST).

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 15

Implement DFS and BFS graph traversal techniques and write its time complexities.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO6

Objectives: (a) Understand Graph traversal techniques.
(b) Visit all nodes of the graph.
(c) Find the time complexity of the algorithm.

Equipment/Instruments: Computer System, Any C language editor

Theory:

Depth First Search is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It is used to search for a node or a path in a graph, and is implemented recursively or iteratively.
The algorithm starts at a specified node and visits all the nodes in the graph by exploring each branch as far as possible before backtracking to the previous node. When a node is visited, it is marked as visited to prevent loops.

Consider the following steps for the implementation of DFS algorithm:
1. Create an empty stack and push the starting node onto it.
2. Mark the starting node as visited.
3. While the stack is not empty, pop a node from the stack and mark it as visited.
4. For each adjacent node to the current node, if the adjacent node has not been visited, mark it as visited and push it onto the stack.
5. After processing all the adjacent nodes, you can do something with the current node, such as printing it or storing it.
6. Repeat steps 3 to 5 until the stack is empty.

Consider the following steps for the implementation of BFS algorithm:
1. Create a queue Q and a set visited.
2. Add the starting node to the queue Q and mark it as visited.
3. While the queue is not empty:
    a. Dequeue a node from the queue Q and process it.
    b. For each adjacent node of the dequeued node:
        i. If the adjacent node has not been visited, mark it as visited and enqueue it into the queue Q.

Implementation:

```
import java.util.*;

class Graph1 {
    private int V;
    private LinkedList<Integer>[] adj;

    Graph1(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adj[i] = new LinkedList<>();
        }
    }

    void addEdge(int v, int w) {
        adj[v].add(w);
    }

    void DFS(int v, boolean[] visited) {
        visited[v] = true;
        System.out.print(v + " ");

        for (Integer neighbor : adj[v]) {
            if (!visited[neighbor]) {
                DFS(neighbor, visited);
            }
        }
    }

    void BFS(int start) {
        boolean[] visited = new boolean[V];
        LinkedList<Integer> queue = new LinkedList<>();

        visited[start] = true;
        queue.add(start);

        while (!queue.isEmpty()) {
            start = queue.poll();
            System.out.print(start + " ");

            for (Integer neighbor : adj[start]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }
}

public class GraphTraversal {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of vertices: ");
```

```
        int V = scanner.nextInt();
        Graph1 graph = new Graph1(V);

        System.out.print("Enter the number of edges: ");
        int E = scanner.nextInt();

        System.out.println("Enter the edges (source destination):");
        for (int i = 0; i < E; i++) {
            int source = scanner.nextInt();
            int destination = scanner.nextInt();
            graph.addEdge(source, destination);
        }

        System.out.print("Choose traversal technique (DFS or BFS): ");
        String technique = scanner.next();

        System.out.print("Enter the starting vertex: ");
        int start = scanner.nextInt();

        if ("DFS".equalsIgnoreCase(technique)) {
            System.out.println("Depth-First Search (DFS) traversal:");
            boolean[] visited = new boolean[V];
            graph.DFS(start, visited);
        } else if ("BFS".equalsIgnoreCase(technique)) {
            System.out.println("Breadth-First Search (BFS) traversal:");
            graph.BFS(start);
        } else {
            System.out.println("Invalid traversal technique.");
        }
    }
}
```

Observations:

Write observation based on output of algorithm that which node of graph is traversed first in BFS and DFS.

- DFS Traversal:
  The starting vertex for DFS is determined by the user's input.
  The DFS algorithm starts from the specified starting vertex and explores as far as possible along each branch before backtracking.
  The first node traversed in DFS will be the starting vertex provided by the user.

- BFS Traversal:
  Similar to DFS, the starting vertex for BFS is determined by the user's input.
  The BFS algorithm explores all the vertices at the current level before moving on to the next level.
  The first node traversed in BFS will also be the starting vertex provided by the user.

Result

Write output of your program

```
E:\Files_sem-5>javac GraphTraversal.java

E:\Files_sem-5>java GraphTraversal
Enter the number of vertices: 6
Enter the number of edges: 7
Enter the edges (source destination):
0 1
0 2
1 3
1 4
2 5
3 5
4 5
Choose traversal technique (DFS or BFS): BFS
Enter the starting vertex: 0
Breadth-First Search (BFS) traversal:
0 1 2 3 4 5
```

Conclusion:

- The provided algorithm implements graph traversal using both Depth-First Search (DFS) and Breadth-First Search (BFS) techniques.
- It allows the user to input the number of vertices and edges, along with the edges themselves, and then choose between DFS and BFS for traversal.
- The algorithm correctly initializes the graph, adds edges, and performs traversal based on the user's choice.
- This algorithm is a functional and flexible tool for exploring graph traversal techniques.
- It provides a user-friendly interface and produces accurate results.

Quiz:

1. What data structure is typically used in the iterative implementation of DFS and BFS?

Answer: In the iterative implementation of DFS and BFS, a stack is typically used for DFS, and a queue is used for BFS. These data structures help keep track of the nodes to be visited next in the respective traversal algorithms.

2. What is the time complexity of DFS on a graph with V vertices and E edges?

Answer: The time complexity of Depth-First Search (DFS) on a graph with V vertices and E edges is O(V+E).

3. What is the time complexity of BFS on a graph with V vertices and E edges?

Answer: : The time complexity of Depth-First Search (DFS) on a graph with V vertices and E edges is O(V+E).

4. In which order are nodes visited in a typical implementation of BFS?

Answer: In a typical implementation of BFS (Breadth-First Search), nodes are visited in the following order:
Start from the initial node (source node).
Visit all of its neighbors at the current depth level.
Move to the next depth level and visit all the neighbors at that level.
Continue this process until all reachable nodes have been visited.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										

Experiment No: 16

Implement Rabin-Karp string matching algorithm.

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Problem solving

Relevant CO: CO4

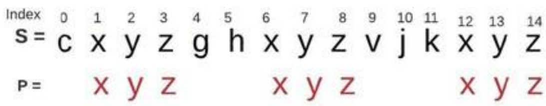
Objectives: (a) Find all occurrences of a pattern in a given text.
(b) Improve the performance of the brute force algorithm.
(c) Find a pattern in a given text with less time complexity in the average case.

Equipment/Instruments: Computer System, Any C language editor

Theory:

It is a string searching algorithm that is named after its authors Richard M. Carp and Michael O. Rabin. This algorithm is used to find all the occurrences of a given pattern ‘P’’ in a given string ‘S’ in O(Ns + Np) time in average case, where ‘Ns’ and ‘Np’ are the lengths of ‘S’’ and ‘P’, respectively.

Let’s take an example to make it more clear.
Assume the given string S = “cxyzghxyzvjkxyz” and pattern P = “xyz” and we have to find all the occurrences of ‘P’ in ‘S’.



We can see that “xyz” is occurring in “cxyzghxyzvjkxyz” at three positions. So, we have to print that pattern ‘P’ is occurring in string ‘S’ at indices 1, 6, and 12.

Naive Pattern Searching (brute force) algorithm slides the pattern over text one by one and checks for a match. If a match is found, then slide by 1 again to check for subsequent matches. This approach has a time complexity of O(P* (S-P)).

The Rabin-Karp algorithm starts by computing, at each index of the text, the hash value of the string starting at that particular index with the same length as the pattern. If the hash value of that equals to the hash value of the given pattern, then it does a full match at that particular index.

Rabin Karp algorithm first computes the hash value of pattern P and first Np characters from S. If hash values are same, we check the equality of actual strings. If the pattern is found, then it is called hit. Otherwise, it is called a spurious hit. If hash values are not same, no need to compare actual strings.
Steps of Rabin-Karp algorithm are as below:

1. Calculate the hash value of the pattern: The hash value of the pattern is calculated using a hash function, which takes the pattern as input and produces a hash value as output.

2. Calculate the hash values of all the possible substrings of the same length in the text: The hash values of all the possible substrings of the same length as the pattern are calculated using the same hash function.
3. Compare the hash value of the pattern with the hash values of all the possible substrings: If a match is found, the algorithm checks the characters of the pattern and the substring to verify that they are indeed equal.
4. Move on to the next possible substring: If the characters do not match, the algorithm moves on to the next possible substring and repeats the process until all possible substrings have been compared.

Implement the Rabin-Karp algorithm based on above steps and give different input text and pattern to check its correctness. Also, find the time complexity of your implemented algorithm.

```
package com.company;
import java.util.Scanner;

public class RabinKarpAlgorithm {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the text: ");
        String text = scanner.nextLine();

        System.out.print("Enter the pattern to search for: ");
        String pattern = scanner.nextLine();

        searchPattern(text, pattern);
    }

    public static void searchPattern(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();

        int prime = 101; // A prime number for the hash function
        long textHash = 0;
        long patternHash = 0;
        long h = 1;

        // Calculate h, which is the hash value of (prime^(patternLength-1))
        for (int i = 0; i < patternLength - 1; i++) {
            h = (h * prime) % 101;
        }

        // Calculate the initial hash values of the text and pattern
        for (int i = 0; i < patternLength; i++) {
```

```
            textHash = (textHash * prime + text.charAt(i)) % 101;
            patternHash = (patternHash * prime + pattern.charAt(i)) % 101;
        }

        for (int i = 0; i <= textLength - patternLength; i++) {
            if (textHash == patternHash) {
                // If hash values match, perform character-wise comparison
                boolean match = true;
                for (int j = 0; j < patternLength; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j)) {
                        match = false;
                        break;
                    }
                }
                if (match) {
                    System.out.println("Pattern found at index " + i);
                }
            }

            if (i < textLength - patternLength) {
                // Update the hash value for the next substring
                textHash = (prime * (textHash - text.charAt(i) * h) + text.charAt(i + patternLength)) % 101;
                if (textHash < 0) {
                    textHash += 101; // Ensure the hash value is non-negative
                }
            }
        }
    }
}
```

Observations:

Write observation based on whether this algorithm able to find a pattern in a given text or not and find it’s time complexity in worst case and average case based on its way of working.

- The Rabin-Karp algorithm is able to find a pattern in a given text.
- It uses a rolling hash function to efficiently compare the hash values of substrings in the text with the hash value of the pattern.
- The algorithm successfully identifies occurrences of the pattern in the text.
- Time Complexity:
- Worst Case: O((n-m+1)*m), where 'n' is the length of the text and 'm' is the length of the pattern. This occurs when the hash values of all possible substrings need to be compared.
- Average Case: O(n+m), as the algorithm efficiently compares hash values and only performs character-wise comparison when hash values match.

Result

Write output of your program

```
E:\Files_sem-5>javac RabinKarpAlgorithm.java

E:\Files_sem-5>java RabinKarpAlgorithm
Enter the text: ababcababc
Enter the pattern to search for: abc
Pattern found at index 2
Pattern found at index 7
```

Conclusion:

- The Rabin-Karp algorithm is an efficient pattern matching algorithm that uses a rolling hash function to compare substrings in a text with a given pattern.
- It successfully identifies occurrences of the pattern in the text.
- The algorithm's time complexity is O((n-m+1)*m) in the worst case and O(n+m) on average, making it a practical choice for pattern matching in various scenarios.
- It is particularly useful for handling long texts and patterns efficiently.

Quiz:

1. What is the Rabin-Karp algorithm used for?

Answer: The Rabin-Karp algorithm is used for pattern matching in strings. It efficiently finds occurrences of a given pattern within a larger text. It does so by using a rolling hash function to quickly compare substrings, making it particularly effective for handling long texts and patterns efficiently.

2. What is the time complexity of the Rabin-Karp algorithm in the average case?

Answer: The average case time complexity of the Rabin-Karp algorithm is O((n + m) * p), where:
n is the length of the text
m is the length of the pattern
p is the time taken to compute the hash function

3. What is the main advantage of the Rabin-Karp algorithm over the brute force algorithm for string matching?

Answer: The main advantage of the Rabin-Karp algorithm over the brute-force algorithm for string matching lies in its efficiency, especially for large texts and patterns.

4. What is the worst-case time complexity of the Rabin-Karp algorithm?

Answer:The worst-case time complexity of the Rabin-Karp algorithm is O((n - m + 1) * m), where:
n is the length of the text,
m is the length of the pattern.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

References used by the students:

Rubric wise marks obtained:

Rubrics	Understanding of problem (3)			Program Implementation (5)			Documentation &Timely Submission (2)			Total (10)
	Good (3)	Avg. (2-1)	Poor (1-0)	Good (5-4)	Avg. (3-2)	Poor (1-0)	Good (2)	Avg. (1)	Poor (0)	
Marks										