UE14CS255: Design and Analysis of Algorithms Laboratory (0-0-2-1) Students Lab Manual

Table of Contents

Input/Output files and Sample source-code:

Ex #21 (Week 13): Greedy Technique: Prim's or Kruskal's algorithm

Ex #22 (Week 13): Greedy Technique: Dijkstra's algorithm

List of Exercises

# of Credits: 1 **# of Weeks: 13**

Week	Exercises
1	Introduction to the lab environment. - Compile and execution of a C program in Linux. - Handling Input-Output formats with large number of test-cases.
2	Brute Force: Implementation of **Sequential Search** algorithm Brute Force: Implementation of **String Matching** algorithm
3	Brute Force: Implementation of **Selection Sort** algorithm Brute Force: Implementation of **Bubble Sort** algorithm
4	Decrease and conquer: Implementation of **Johnson-Trotter** algorithm to generate permutations Brute Force: Solution for **Traveling Salesperson Problem**
5	Divide and Conquer: Implementation of **Merge Sort** Divide and Conquer: Implementation of **Quick Sort**
6	Divide and Conquer: Implementation of **Binary Search** Divide and Conquer: **Karatsuba** algorithm for **Multiplication of Large Integers**
7	Divide and Conquer: Implementation of **Binary Tree Traversal** algorithms Decrease and conquer: Implementation of **Insertion Sort** algorithm
8	Decrease and conquer: Demonstration of **BFS** and **DFS** algorithms Decrease and conquer: **Topological Sorting** of vertexes in a digraph
9	Transform and Conquer: Implementation of **Heap Sort** algorithm Space and Time Tradeoffs: Implementation of **Sorting by Distribution Counting** algorithm
10	On the spot problem to solve, which uses one or more the concepts learnt so far.
11	Space and Time Tradeoffs: Implementation of **Horspool's** or **Boyer-Moore** algorithm Dynamic Programming: Solution for the **Knapsack Problem**
12	Dynamic Programming: Implementation of **Warshall's** algorithm Dynamic Programming: Implementation of **Floyd's** algorithm
13	Greedy Technique: Implementation of **Prim's** and **Kruskal's** algorithm Greedy Technique: Implementation of **Dijkstra's** algorithm

General Instructions for Students

1. Before coming to the lab,
    a. Read the questions well enough.
    b. Prepare/download the input/output files for the test-cases.
    c. Write the program and see if it works with the given test-cases.
    d. Write the questions in the observation book.
    e. Carry input/output files and your source-code into your memory-stick/usb-drive.
2. In the lab,
    a. If you've got permission to show output of previous lab, do it in the first ten minutes. DON'T spend much time on the previous week's exercises.
    b. Listen to any more instructions from the lab instructors.
    c. Execute and show output to the lab instructor. On approval, write the output (or any specific things asked) in your observation book following the question. Get it signed by the lab instructor.
    d. If the lab instructor asks for any more changes, do it.
    e. Answer the Viva questions, if asked.
    f. You are NOT expected to carry forward the exercises for the following week. On special cases, you should get permission from the lab instructor to show the output in the first ten minutes of the next week's lab.
3. Write a program for the question at home and test against the input/output files with commands similar to the following:
    a. Compile
    ```
    :~$ gcc -Wall PES16GCD.c -lrt
    ```
    b. Execute
    ```
    :~$ ./a.out < PES16GCD_ip2.txt > my_PES16GCD_op2.txt
    ```
    c. Test for correctness
    ```
    :~$ diff -wB my_PES16GCD_op2.txt PES16GCD_op2.txt
    100001c100001
    < Execution time: 56.689300 millisec.
    ---
    >
    :~$
    ```
4. Write the exercise questions and output for a smaller input in your observation book and get it signed by the lab instructor.
5. Answer the questions asked by the lab instructors, which could potentially be counted as viva.

Frequently Asked Questions (FAQs)

1. Why does initializing array doesn't work after reading its size by scanf()?

```
a.      scanf("%d",&n);
        student s[n];
```

Don't do this. Generally, keep all the declarative statements above any executable statement because decision about declarative statements is made at compile-time. So, s[n] could have initialized with some "unpredictable" value of n, not the you've fed into. A better way of achieving the same is,

```
        student *s;
        scanf("%d",&n);
        s = (student *) malloc(n * sizeof(student));
```

Make sure you "`free(s);`" when you don't need that piece of memory block.

2. Why doesn't floor() and ceil() functions work the way we expect in the following code?

```
p = ((int)(floor(n/2)));
q = ((int)(ceil(n/2)));
```

   a. If n is an int, then the statement with ceil() doesn't work the way you expect. That's because n/2 is an integer division operation which is automatically floored. So, you'll never get the actual ceil when n is odd. A correct way of achieving the same is as follows.

```
p = n/2;
q = ((int)(ceil(n/2.0)));
```

   b. Another way of achieving the same is:

```
p = n/2;
q = n - p;
```

3. How to allocate a 2D array and pass it across functions safely?
   a. Here's how I do it. A 2D array of ints would have datatype as "**`int **`**". Allocate an array of r number of "**`int *`**", where r is the number of rows. That's a pointer for each row. Now allocate c number of "**`int`**" for each row, where c is the number of columns. That way accessing an element **`a[i][j]`** is equivalent to **`*(*(a+i)+j)`**.
   b.

```
//Allocates 2D array of ints having r rows and c columns
int** alloc_array2d_int(int r, int c) {
      int **a;
      int i;
      a = (int **) malloc(r * sizeof(int *)); //r rows
      for(i = 0; i < c; i++) {
            a[i]=(int*)malloc(c*sizeof(int)); //cols for ith
row
      }
      return a;
}

//Frees an allocated 2D array having r rows
void free_array2d(int **a, int r) {
```

```
        int i;
        for(i = 0; i < r; i++) {
              free(a[i]);
        }
        free(a);
}
```

4. What do we gain by using these input/output files? It was simpler to give inputs manually.
    a. Large inputs cannot be tested by manually typing in the terminal. Large inputs are necessary to observe the time taken by the algorithms. Algorithmic analysis is all for executing faster on large inputs.
    b. Multiple test-cases covering border cases can be tested in one go.
    c. To observe order of growth functions you studies in theory by increasing the input size systematically and measuring the executing time.
    d. My philosophy is “If it can be automated, it should be automated”.
    e. Sooner than later you need to get used to it for other projects, interviews, and programming contests.
5. I copied from sample input from a pdf file of the lab manual into a text file like PES16_ip1.txt. Now, my program is not reading the input as expected. What am I missing?
    a. This has happened to a few students when the input had minus symbol like -999999. The pdf file must have rendered it as a special character to display beautifully. When you copied and pasted to a text file, that could a different ascii character, which is not even displayed as a character. When you move the cursor by keyboard arrow keys, you’ll notice a missing a char there. Deleting such instances has solved the issue.
6. How to measure the time taken to execute a function?
    a. Refer Guidelines for solving an exercise:. Steps 5 and 6 explain the usage of clock_gettime() function to measure the execution time of a code segment.
7. I want to see what’s happening during the execution step by step with changing values of the variables. How to debug like that?
    a. Extra printf statements help to catch the state of variables at the instances you want to observe. Make sure to put ‘\n’ at end of every such printf statements because sometimes printf statements doesn’t display on the terminal until a full line is sent to the stdout.
    b. Use gdb, which is GNU debugger. It’s a tool which takes an executable file and runs it in a controlled way, which allows you to analyze what happens step by step. For that, while compiling using gcc, add an option -g to your command. That adds necessary debugging information to your executable file (a.out in most of your cases). Then run gdb command with a.out as a command line parameter. That puts you into a gdb prompt. gdb has its own commands, which work in gdb prompt. Learn more about how to debug in gdb.
8. How to write a modular program with multiple c files and header file?
    a. Refer Keeping your core functions in a library: (Optional), where in we’ve explained it for the sample exercise of finding GCD of two non-negative integers.
9. How to read a full line as a string with spaces?
    a. `// to read a string into str up to 1000 characters.`

```
// The string will have '\0' in the end and not '\n'.
char str[1001];
scanf(" %[^\n]", str);
```

10. Will the modifications asked in the lab be counted for ISA?
    a. Unless explicitly mentioned it as optional, it'll be counted for ISA.
11. Do we get a grace time of a week to show the output?
    a. No, for the exercises of the week, you should show the output in the lab session. Most of the heavy-lifting should be done at home. There will even be an online judge for you to try out at home. On special cases, you should get permission from the lab instructor to show the output in the first ten minutes of the next week's lab.
12. What should be written in the observation book?
    a. Write the exercise question in the observation book at home. During the lab, after showing output to the lab instructor, write the output for a smaller input (anything in particular asked by the lab instructor) in your observation book and get it signed by the lab instructor.
13. Any guidelines for solving a problem?
    a. Refer Guidelines for solving an exercise:

Sample Exercise (Week 1): Find GCD using Euclid's algorithm

An online version of the below problem is at http://www.spoj.com/problems/PES16GCD/

Write a program to find gcd (greatest common divisor) of two non-negative integers using Euclid's algorithm.

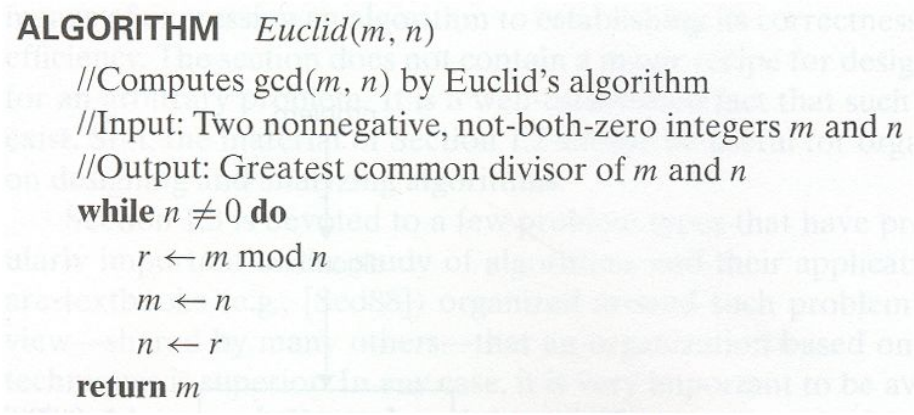
Input:
Input begins with t ($1 \leq t \leq 100,000$) of number of test-cases in the first line and the test-cases are in the following lines. Each test-case has m and n ($0 \leq m, n \leq 2^{20}$) on a single line separated by a space.

Output:
For each test-case, print the gcd of m and n. Print the total time taken in milliseconds.

Example:
Input:
5
60 24
100 101
120 420
0 123
123 0

Output:
12
1
60
123
123
1.234567 milliseconds (Note: This value is only representative)

Algorithm: Euclid's algorithm



ALGORITHM *Euclid*(m, n)

```
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

Input/Output files and Sample source-code:

Sample input file: PES16GCD_ip1.txt
Expected Output for the Sample input: PES16GCD_op1.txt
Large input file: PES16GCD_ip2.txt
Expected Output for the large input: PES16GCD_op2.txt

Sample Source-code of the solution:
PES16GCD.c.

Guidelines for solving a problem:

Step 1: Create sample input/output files from the question.
Example:
Question: Sample Exercise (Week 1): Find GCD using Euclid's algorithm
Sample input file: PES16GCD_ip1.txt
Sample output file: PES16GCD_op1.txt

Step 2: Write code, which works for the sample input/output files. That is, it should input from the sample input file and produce output equivalent to the given sample output file.
Example:
Source-code of the solution: PES16GCD_spoj.c.

Step 3: Test against the sample input/output files. The following commands may help you.

```
:~$ gcc -Wall PES16GCD_spoj.c
```
Command to compile the source-file to get an executable file a.out.
Any "gcc" version should be good enough for us.
Command-line option "**`-Wall`**" enables all the warnings to be shown to you when it compiles (compiles and links). Make sure you resolve all the errors and warnings before attempting to execute.

```
:~$ ./a.out < PES16GCD_ip1.txt > my_PES16GCD_op1.txt
```
Command to execute with PES16GCD_ip1.txt as the input file. For the current process, the "stdin" file will be mapped to PES16GCD_ip1.txt instead of the keyboard and hence scanf() will read from PES16GCD_ip1.txt instead of the keyboard. Similarly, "stdout" file will be mapped to my_PES16GCD_op1.txt instead of the display terminal and hence printf() will write into my_PES16GCD_op1.txt instead of the display terminal.
The above command should execute and terminate without expecting any keyboard input and displaying onto the terminal.

```
:~$ diff -wB my_PES16GCD_op1.txt PES16GCD_op1.txt
```
Command to see if there is any difference between your output file my_PES16GCD_op1.txt and the expected output file PES16GCD_op1.txt. If the above command returns back to shell without displaying anythings, then your output the expected output match. The option -wB makes it to ignore the differences in whitespaces (like spaces and tabs) and blank lines.

```
:~$ ./a.out < PES16GCD_ip2.txt > my_PES16GCD_op2.txt
:~$ diff -wB my_PES16GCD_op2.txt PES16GCD_op2.txt
```
Commands to test against another input file. In this case, it has large number of test cases. Failing in this cases means some boundary cases are not handled by you including the case of handling just more test cases.

Step 4: (Optional) Submit the above source-code (PES16GCD_spoj.c) to the online judge at http://www.spoj.com/problems/PES16GCD/. It should be accepted. Possible failures are:

1. Compiler Error.
2. Runtime Error.
3. TLE (Time Limit Exceeded) because your code took lot more time than the time limit set by us.
4. NZEC (Non Zero Exit Code) because you missed to return 0 at the end of main().
5. Wrong output because either your answer is wrong or your output is matching the format of the expected output.

Step 5:
Write code for the lab. One extra thing you'll need to do here is typically measuring the absolute time (in milliseconds) your algorithm has taken.
Example:
Source-code of the solution: PES16GCD.c.

Step 6:
Test against the sample input/output files. Your terminal may look like this if everything goes fine. The diff command shows some diff, which is about the execution time and the expected output file didn't have it. Command-line option "**`-lrt`**" in gcc command is to explicitly mention the library, which has the definition of the function **clock_gettime()**. Declaration of the function was in **time.h**.

```
:~$ gcc -Wall PES16GCD.c -lrt
:~$ ./a.out < PES16GCD_ip2.txt > my_PES16GCD_op2.txt
:~$ diff -wB my_PES16GCD_op2.txt PES16GCD_op2.txt
100001c100001
< Execution time: 56.689300 millisec.
---
>
:~$
```

Keeping your core functions in a library: (Optional)

Instead of writing everything in one source-file like in PES16GCD.c, it's a good practice to keep the core functions in a separate file (we call such a thing as library) and use it your demo program.
Example:
PES16_daa_lib.c
PES16_daa_lib.h
PES16GCD_demo.c

The following snapshot of my terminal should demonstrate the essential commands you need.

```
:~/$ gcc -Wall -c PES16_daa_lib.c
:~/$ gcc -Wall PES16GCD_demo.c PES16_daa_lib.o -lrt
:~/$ ./a.out < PES16GCD_ip2.txt > my_PES16GCD_op2.txt
:~/$ diff -wB my_PES16GCD_op2.txt PES16GCD_op2.txt
100001c100001
```

```
< Execution time: 63.751966 millisec.
---
>
:~$
```

For those who are looking for even more details, the following snapshot of my terminal may clarify some more concepts.

```
:~$ gcc -Wall -c PES16_daa_lib.c
:~$ nm PES16_daa_lib.o
0000000000000078 T gcd
0000000000000000 T time_elapsed
:~$
:~$ gcc -Wall -c PES16GCD_demo.c
:~$ nm PES16GCD_demo.o
                 U __isoc99_scanf
                 U clock_gettime
                 U gcd
0000000000000000 T main
                 U printf
                 U time_elapsed
:~$
:~$ gcc -Wall PES16GCD_demo.o PES16_daa_lib.o -lrt
:~$ ./a.out < PES16GCD_ip2.txt > my_PES16GCD_op2.txt
:~$ diff -wB my_PES16GCD_op2.txt PES16GCD_op2.txt
100001c100001
< Execution time: 60.979241 millisec.
---
>
:~$
```

Notes:

Command-line option "**`-c`**" in gcc command compiles the source file, but does not link it and hence it outputs the respective "**`.o`**" file, which we call as an object file. Object file is a binary file, which is neither a source file nor an executable file. It's just a compiled version of a set of functions. Linker takes the object file (possibly with some more object files) and libraries to link them to an "**executable file**", which has an entry through a `main()` function. We can a "**`.a`**" file as a library file, which is just a collection of object files. Command-line option "**`-lrt`**" refers to a library file, which has an object file having the definition of `clock_gettime()` function.

Ex #1 (Week 2): Brute Force: Sequential Search algorithm

An online version of the below problem is at http://www.spoj.com/problems/PES16SEQ/

Write a program to search for an element in an array of n elements using sequential/linear search algorithm.

Input:
Input begins with t ($1 \leq t \leq 100$) of number of test-cases in the first line and the test-cases are in the following lines. Each test-case begins with n ($1 \leq n \leq 2^{20}$) of number of integers ($-2^{31} \leq$ integer $\leq 2^{31} - 1$) in the array in a single line and followed by n lines having an integer in each line and the integer to be searched in a new line.

Output:
For each test-case, print the index ($0 \leq$ index $\leq n-1$) of the first appearance of the search element in the array in a new line. Print '-1' if the element is not found in the array. Print the total time taken in milliseconds.

Example:
Input:
3
4
999999
0
-999999
1234
1234
4
999999
0
-999999
1234
-1234
4
999999
0
-999999
1234
999999

Output:
3
-1
0
1234.567 (Note: This value in milliseconds may be different for each execution)

Algorithm: Sequential Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Input/Output files and Sample source-code:

Sample input file: PES16SEQ_ip1.txt
Expected Output for the Sample input: PES16SEQ_op1.txt
Large input file: PES16SEQ_ip2.txt
Expected Output for the large input: PES16SEQ_op2.txt

The following snapshot of my terminal may help you.

```
:~$ gcc -Wall PES16SEQ.c -lrt
:~$ ./a.out < PES16SEQ_ip2.txt > my_PES16SEQ_op2.txt
:~$ diff -wB my_PES16SEQ_op2.txt PES16SEQ_op2.txt
4c4
< Execution time: 153.137028 millisec.
---
>
:~$
```

Source-code of the solution:
PES16SEQ.c

Ex #2 (Week 2): Brute Force: String Matching algorithm

An online version of the below problem is at http://www.spoj.com/problems/PES16STR/

Write a program to search for a substring (pattern) in a longer string (text) using brute-force string matching algorithm.

Input:
Input begins with t ($1 \leq t \leq 100$) of number of test-cases in the first line and the test-cases in the following lines. Each test-case begins with a text ($1 \leq$ length of the text $\leq 2^{20}$) in a single line and the pattern to be searched in a new line. The text and pattern could have spaces as characters.

Output:
For each test-case, print the index ($0 \leq$ index $<$ length of the text) of the beginning the pattern in the text in a new line. Print '-1' if the substring is not found in the text. Print the total time taken in milliseconds.

Example:
Input:
2
Discrete Mathematics and Logic is a prerequisite for Design and Analysis of Algorithms.
and
Discrete Mathematics and Logic is a prerequisite for Design and Analysis of Algorithms.
Data and

Output:
21
-1
1234.567 (Note: This value in milliseconds may be different for each execution)

Algorithm: Brute-Force String Matching

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Input/Output files and Sample source-code:

Sample input file: PES16STR_ip1.txt
Expected Output for the Sample input: PES16STR_op1.txt
Large input file: PES16STR_ip2.txt
Expected Output for the large input: PES16STR_op2.txt

Source-code of the solution:
PES16STR.c

Ex #3 (Week 3): Brute Force: Selection Sort algorithm

Write a program to sort an array of integers in ascending order using Selection Sort algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PES16SO1/

Input:
Input begins with n ($1 \leq n \leq 2^{20}$) of number integers in the array in a single line and followed by n lines having an integer in each line.

Output:
Print the sorted array with one integer in each new line in ascending order and print the time taken (in milliseconds) to sort the array in a new line.

Example:
Input:
4
999999
0
-999999
1234

Output:
-999999
0
1234
999999
12.3 millisec (Note: This value is only representative)

Algorithm: Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

Input/Output files and Sample source-code:

Sample input file: PES16sort_ip1.txt
Expected Output for the Sample input: PES16sort_op1.txt
Large input / output files:
PES16sort_ipRank1k.txt / PES16sort_opRand1k.txt
PES16sort_ipRank2k.txt / PES16sort_opRand2k.txt
PES16sort_ipRank4k.txt / PES16sort_opRand4k.txt
PES16sort_ipRank8k.txt / PES16sort_opRand8k.txt
PES16sort_ipRank16k.txt / PES16sort_opRand16k.txt
PES16sort_ipRank32k.txt / PES16sort_opRand32k.txt
PES16sort_ipRank64k.txt / PES16sort_opRand64k.txt
PES16sort_ipRank128k.txt / PES16sort_opRand128k.txt
PES16sort_ipRank256k.txt / PES16sort_opRand256k.txt
PES16sort_ipRank512k.txt / PES16sort_opRand512k.txt
PES16sort_ipRank1024k.txt / PES16sort_opRand1024k.txt

Note down the execution time for the above 11 inputs of random integers of size 2^{10}, 2^{11}, .., 2^{20}. Write a table of the durations in your observation book against the input size. Plot a graph with size on the x-axis and duration on the y-axis. Plot the the graph both on a linear graph sheet and also a logarithmic graph sheet.

Source-code of the solution:
PES16SelectionSort.c.

Ex #4 (Week 3): Brute Force: Bubble Sort algorithm

Write a program to sort an array of integers in ascending order using Bubble Sort algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PES16SO1/

Input:
Input begins with n ($1 \leq n \leq 2^{20}$) of number integers in the array in a single line and followed by n lines having an integer in each line.

Output:
Print the sorted array with one integer in each new line in ascending order and print the time taken (in milliseconds) to sort the array in a new line.

Example:
Input:
4
999999
0
-999999
1234

Output:
-999999
0
1234
999999
12.3 millisec (Note: This value is only representative)

Algorithm: Bubble Sort

ALGORITHM *BubbleSort*$(A[0..n-1])$

```
//Sorts a given array by bubble sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    for j ← 0 to n − 2 − i do
        if A[j + 1] < A[j]  swap A[j] and A[j + 1]
```

Input/Output files and Sample source-code:

(Note: Same files from the exercise of Selection Sort)
Sample input file: PES16sort_ip1.txt
Expected Output for the Sample input: PES16sort_op1.txt
Large input / output files:
PES16sort_ipRank1k.txt / PES16sort_opRand1k.txt
PES16sort_ipRank2k.txt / PES16sort_opRand2k.txt
PES16sort_ipRank4k.txt / PES16sort_opRand4k.txt
PES16sort_ipRank8k.txt / PES16sort_opRand8k.txt
PES16sort_ipRank16k.txt / PES16sort_opRand16k.txt
PES16sort_ipRank32k.txt / PES16sort_opRand32k.txt
PES16sort_ipRank64k.txt / PES16sort_opRand64k.txt
PES16sort_ipRank128k.txt / PES16sort_opRand128k.txt
PES16sort_ipRank256k.txt / PES16sort_opRand256k.txt
PES16sort_ipRank512k.txt / PES16sort_opRand512k.txt
PES16sort_ipRank1024k.txt / PES16sort_opRand1024k.txt

Note down the execution time for the above 11 inputs of random integers of size 2^{10}, 2^{11}, .., 2^{20}. Write a table of the durations in your observation book against the input size. Plot a graph with size on the x-axis and duration on the y-axis. Plot the the graph both on a linear graph sheet and also a logarithmic graph sheet.

Source-code of the solution:
PES16BubbleSort.c.

Ex #5 (Week 4): Decrease and conquer: Johnson-Trotter algorithm

Write a program to print all permutations of n distinct symbols using the Johnson-Trotter algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PES16JT/

Input:
Input begins with t ($1 \le t \le 9$) of number of test-cases in the first line and a test-case in each of the following lines. A test-case to have a positive integer n ($1 \le n \le 9$) in a single line.

Output:
For each test-case, print all the permutations one per line in the order of generation of permutations by the Johnson-Trotter algorithm. A permutation is symbols from 1 to n separated by a space. Print the total time taken in milliseconds.

Example:
Input:
3
1
2
3

Output:
1
1 2
2 1
1 2 3
1 3 2
3 1 2
3 2 1
2 3 1
2 1 3
1.23 millisec (Note: This value is only representative)

Algorithm: Johnson-Trotter algorithm

ALGORITHM *JohnsonTrotter(n)*

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
find its largest mobile element k
swap k with the adjacent element k's arrow points to
reverse the direction of all the elements that are larger than k
add the new permutation to the list

```
Algorithm Johnson-Trotter(n)
//Input: Non-negative integer n
//Output: Return the list L of permutations of symbols 1 thru n
      for i ← 1 to n
            p[i] ← i
            d[i] ← left

      Add p to the list L
      m ← GetLargestMobileElement(p, d)
      while(m ≠ -1)
            if (d[m] = left) i ← m-1
            else i ← m+1
            swap p[m] and p[i]
            change direction of the symbols which are greater than p[i]
            Add p to the list L
            m ← GetLargestMobileElement(p, d)
End of Algorithm Johnson-Trotter

Procedure GetLargestMobileElement(p, d)
//Input: Permutation p and corresponding directions d
//Output: Returns the index (1 thru n) of the largest mobile element.
//          -1, if not found.
      m ← -1
      for i ← 1 to n
            if(d[i] = left) and (i > 1) and (p[i] > p[i-1]))
                  if(m = -1) m ← i
                  else if (p[i] > p[m]) m ← i
            else if(d[i] = right) and (i < n) and (p[i] > p[i+1]))
                  if(m = -1) m ← i
                  else if(p[i] > p[m]) m ← i
      return m
End of Procedure GetLargestMobileElement
```

Input/Output files and Sample source-code:

Sample input file: PES16JT_ip1.txt

Expected Output for the Sample input: PES16JT_op1.txt
Large input / output files: PES16JT_ip2.txt / PES16JT_op2.txt

Source-code of the solution:
PES16JohnsonTrotter_Sample.c (It's a sample code of the implementation of the algorithm. Before opening it, you are encouraged to implement the algorithm on your own. Open after you write the code or when you find it hard to implement)

PES16JohnsonTrotter.c

Ex #6 (Week 4): Brute Force: Traveling Salesperson Problem

Write a program to find the shortest path from the first vertex traversing all other vertexes in a complete graph and returning back to the first vertex. In other words, find a solution to a Travelling Salesperson Problem using an exhaustive search algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PES16TSP/

Input:
Input begins with n ($1 \leq n \leq 11$) of number of vertexes of a graph. The following n lines to have the cost matrix of the graph with n non-negative integers ($0 \leq cost \leq 100$) in each line separated by spaces.

Output:
Print the cost of the shortest circuit starting at the first vertex and traversing through all other vertexes of the complete graph. Print the total time taken in milliseconds.

Example:
Input:

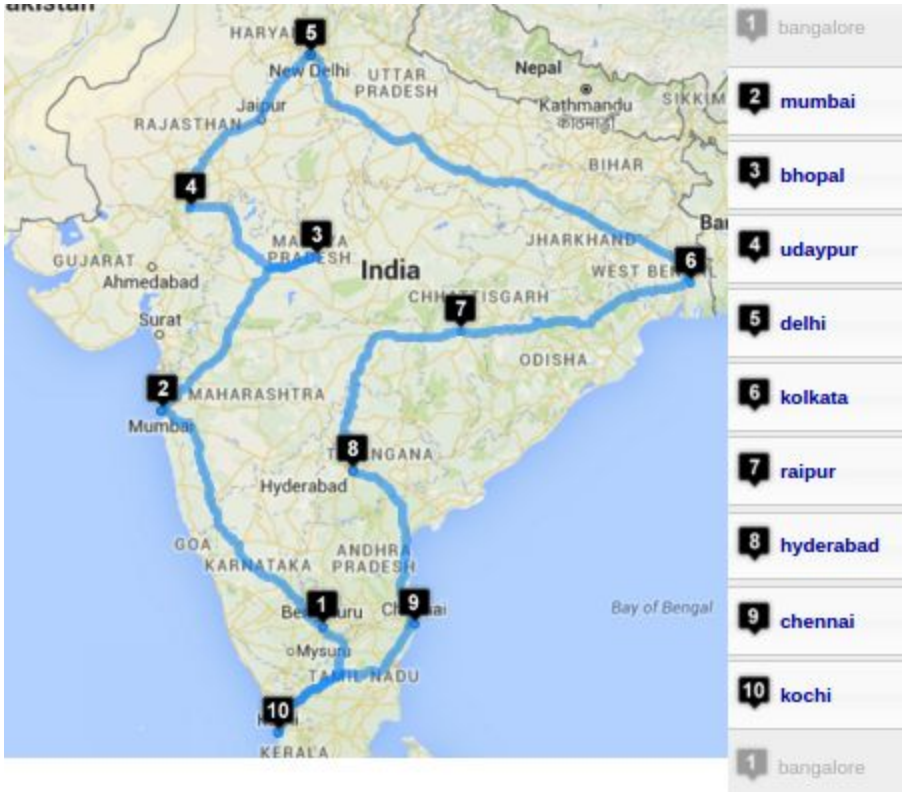
```
5
0       100     125     100     75
100     0       50      75      125
125     50      0       100     125
100     75      100     0       50
75      125     125     50      0
```

Output:
375
12.3 millisec (Note: This value is only representative)

```
Algorithm TravellingSalespersonProblem
//Input: nxn cost matrix A of complete graph with n vertices.
//Output: Min Cost Hamiltonian circuit.
      mincost ← Infinity
      for each permutation of (n-1) cities
            cost ← 0
            for each edge in the Hamiltonian circuit
                  cost ← cost + cost of the edge
            if (cost < mincost)
            mincost ← cost
      return mincost
```

Input/Output files and Sample source-code:

Sample input file: PES16TSP_ip1.txt
Expected Output for the Sample input: PES16TSP_op1.txt
Large input / output files:
PES16TSP_ip2.txt / PES16TSP_op2.txt
PES16TSP_ip2.txt is of the complete graph of ten selected cities in India with cost of an edge being the estimated driving time in seconds. Below pictures show the selected cities and the solution for the TSP.



Source-code of the solution:
PES16TSP.c.

Ex #7 (Week 5): Divide and Conquer: Merge Sort

Write a program to sort an array of records (a record to have a register number, name of the students and his/her CGPA) by their names using Mergesort algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PES16SO2/

Input:
Input begins with n ($1 \leq n \leq 2^{20}$) of number student records. The following n lines has a student record per line. A student record of a student has a unique register number (one alphanumeric word and 3 <= length of the word <= 10), his/her name (one alphabetic word without any spaces and 1 <= length of the name <= 20) and his/her CGPA with the precision up to two digits after decimal point (00.00 <= CGPA <= 10.00). The three fields of a record are separated by a spaces.

Output:
Print the sorted array with one student record in each new line in ascending order of student's name and print the time taken (in milliseconds) to sort the array in a new line. In case of more than one record has a common name, the relative order of the records need to be maintained (i.e., sorting has to be "stable").

Example: Input:
10
CS003 Vinay 10
CS005 Mouli 9.94
CS010 Gautham 9.94
CS020 Sneha 9.94
CS200 Mohit 9.93
CS012 Aarti 9.9
CS006 Darshan 9.88
CS002 Adithya 9.78
CS050 Karthik 9.71
CS001 Adithya 9.58

Output:
CS012 Aarti 9.9
CS002 Adithya 9.78
CS001 Adithya 9.58
CS006 Darshan 9.88
CS010 Gautham 9.94
CS050 Karthik 9.71
CS200 Mohit 9.93
CS005 Mouli 9.94
CS020 Sneha 9.94
CS003 Vinay 10
12.3 millisec (Note: This value is only representative)

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i + 1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Input/Output files and Sample source-code:

Sample input file: PES16sort_records_ip1.txt
Expected Output for the Sample input: PES16sort_records_names_op1.txt
Large input / output files: PES16sort_records_ip2.txt / PES16sort_records_names_op2.txt

Q. How is sorting an array of structs is different from sorting an array of ints?
Ans: Not much. Suppose, the struct looks like:

```
//student record
typedef struct record {
      char regno[11];//3 <= l <= 10
      char name[21];//1 <= l <= 20
      float cgpa; //0.00 to 10.00
} record;
```

When two elements of the array are compared for sorting for say “less than or equal to” condition, instead of using “<=” operator like in comparing ints, here we can use a function to compare the structs based on whatever condition we are interested. If we are sorting in ascending order by the “name” field, the comparison function would look like:

```
//if name field of first record is <= second record
int name_is_less_than_or_equal_to(record *a, record *b) {
      int c = strcmp(a->name, b->name);
      if(c <= 0) {
            return 1;
      } else {
            return 0;
      }
}
```

Source-code of the solution:
PES16Mergesort.c.

Ex #8 (Week 5): Divide and Conquer: Quick Sort

Write a program to sort an array of records (a record to have a register number, name of the students and his/her CGPA) by their register numbers using Quick Sort algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PES16SO3/

Input:
Input begins with n ($1 \leq n \leq 2^{20}$) of number student records. The following n lines has a student record per line. A student record of a student has a unique register number (one alphanumeric word and 3 <= length of the word <= 10), his/her name (one alphabetic word without any spaces and 1 <= length of the name <= 20) and his/her CGPA with the precision up to two digits after decimal point (00.00 <= CGPA <= 10.00). The three fields of a record are separated by a spaces.

Output:
Print the sorted array with one student record in each new line in ascending order of student's register number and print the time taken (in milliseconds) to sort the array in a new line.

Example: Input:
10
CS003 Vinay 10
CS005 Mouli 9.94
CS010 Gautham 9.94
CS020 Sneha 9.94
CS200 Mohit 9.93
CS012 Aarti 9.9
CS006 Darshan 9.88
CS002 Adithya 9.78
CS050 Karthik 9.71
CS001 Adithya 9.58

Output:
CS001 Adithya 9.58
CS002 Adithya 9.78
CS003 Vinay 10
CS005 Mouli 9.94
CS006 Darshan 9.88
CS010 Gautham 9.94
CS012 Aarti 9.9
CS020 Sneha 9.94
CS050 Karthik 9.71
CS200 Mohit 9.93
12.3 millisec (Note: This value is only representative)

ALGORITHM *Quicksort*$(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//       indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ←Partition(A[l..r]) //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//       as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//       indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//       this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])  //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Input/Output files and Sample source-code:

Sample input file: PES16sort_records_ip1.txt
Expected Output for the Sample input: PES16sort_records_regno_op1.txt
Large input / output files: PES16sort_records_ip2.txt / PES16sort_records_regno_op2.txt

Source-code of the solution:
PES16Quicksort.c.

Ex #9 (Week 6): Divide and Conquer: Binary Search

Write a program to search for a student-record by name of the student in a sorted (on name field) array of student-records using Binary Search algorithm.

Input:
Input begins with the size of the sorted array n ($1 <= n <= 2^{20}$) followed by n lines with each line having a student-record. In a new line, a name is given, which needs to be searched in the array of student-records against the name field.

A student-record has a unique register number (one alphanumeric word and 3 <= length of the word <= 10), his/her name (one alphabetic word without any spaces and 1 <= length of the name <= 20) and his/her CGPA with the precision up to two digits after decimal point (00.00 <= CGPA <= 10.00). The three fields of a record are separated by a spaces.

Output:
Print the index (0 ≤ index < length of the text) of the first student-record in which the given name matched with the name field of the student-record. Return -1 if the search is unsuccessful search. Print the total time taken in milliseconds.

Example:
Input:
10
CS012 Aarti 9.90
CS002 Adithya 9.78
CS001 Adithya 9.58
CS006 Darshan 9.88
CS010 Gautham 9.94
CS050 Karthik 9.71
CS200 Mohit 9.93
CS005 Mouli 9.94
CS020 Sneha 9.94
CS003 Vinay 10.00
Adithya

Output:
1
12.3 millisec (Note: This value is only representative)

Algorithm: Binary Search:

```
Algorithm BinarySearchRec(A[l..r], K)
      //Searches for the first occurrence of K in A
      if(l > r)
            return -1
      m = ⌊(l + r) / 2⌋
      if(k = A[m])
            while(m > 0 and A[m] = A[m-1])
                  m ← m - 1
            return m
      if(k < A[m])
            return BinarySearchRec(A[l..m-1], K)
      else
            return BinarySearchRec(A[m+1..r], K)
```

Input/Output files and Sample source-code:

Sample input file: PES16BIN_ip1.txt
Expected Output for the Sample input: PES16BIN_op1.txt
Large input / output files: PES16BIN_ip2.txt / PES16BIN_op2.txt

Note: Problem on Binary Search uses the similar student-records you used in Mergesort. In fact, the output of Mergesort is very similar to the input of Binary Search.

Source-code of the solution:
PES16BIN.c.

Ex #10 (Week 6): Divide and Conquer: Karatsuba Algorithm for Multiplication of Large Integers

Write a program to multiply two large integers using Karatsuba algorithm.

Input:
Input has two lines each with a large integer in base-10 ($1 \leq$ large integer $< 10^{1M}$, where 1M means 2^{20}). That is, each integer can be up to 2^{20} digits.

Output:
Print the product of the given two large integers in base-10. Note that the product may have up to 2 million (2^{21}) digits. Print the total time taken in milliseconds.

Example:
Input:
12345678
32165487

Output:
397104745215186
12.3 millisec (Note: This value is only representative)

Algorithm: Karatsuba Algorithm

```
Algorithm Karatsuba(a, b, k)
//Multiplication of large integers a and b of 2^k decimal digits
  if (k = 0) return a*b
  m = 2^k
  m2 = m/2
  a1, a2 = split_at(a, m2)
  b1, b2 = split_at(b, m2)
  p1 = Karatsuba(a1, b1, k-1)
  p2 = Karatsuba(a2, b2, k-1)
  p3 = Karatsuba((a1+a2), (b1+b2), k)
  return (p1*10^m)+((p3-p1-p2)*10^m2)+(p2)
```

Note: Problem on Karatsuba algorithm normally needs more effort from you.

1. It is expected to multiply two large integers (up to 2^{20} digits in base-10) more efficiently than the primary-school algorithm. We've given an easier version of Karatsuba algorithm which expects the inputs of length 2^k so that the splitting is always even. For that you need pre-process the original inputs to make it of length 2^k by inserting leading zeroes.
2. Obviously, using a string of characters is easier to represent these large integers. When you store a digit as a character in the string, you mostly store it as an ASCII character. For calculations, make sure you convert it to the integer value. It may look like "**c - '0'**" or "***number[i] - '0'**", where is c is of type '**char**' and number is of type '**char ***'.
3. It also needs addition of two large integers of up to 2^{21} million digits.

Input/Output files and Sample source-code:

Sample input file: PES16KAR_ip1.txt
Expected Output for the Sample input: PES16KAR_op1.txt
Large input / output files:
PES16KAR_ip2.txt / PES16KAR_op2.txt
PES16KAR_ip3.txt / PES16KAR_op3.txt
PES16KAR_ip4.txt / PES16KAR_op4.txt
PES16KAR_ip5.txt / PES16KAR_op5.txt
PES16KAR_ip6.txt / PES16KAR_op6.txt

Source-code of the solution:
PES16KAR.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #11 (Week 7): Divide and Conquer: Binary Tree Traversal

Write a program to insert a sequence of distinct integers into a Binary Search Tree (BST) and print the sequence of integers as per the post-order traversal on the BST.

Input:
Input begins with n ($1 \le n \le 2^{20}$), which is the number of integers to be inserted into the BST. It is followed by n lines each having an integer to be inserted into the BST in the given order ($-2^{30} \le$ integer $\le 2^{30}$).

Output:
Print the n integers in the sequence as per the post-order traversal on the BST, one per line. Print the total time taken in milliseconds.

Example:
Input:
4
0
999999
-999999
1234

Output:
-999999
1234
999999
0
12.3 millisec (Note: This value is only representative)

Input/Output files and Sample source-code:

Sample input file: PES16BST_ip1.txt
Expected Output for the Sample input: PES16BST_op1.txt
Large input / output files: PES16BST_ip2.txt / PES16BST_op2.txt

Source-code of the solution:
PES16BST.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #12 (Week 7): Decrease and conquer: Insertion Sort

Write a program to sort an array of integers in ascending order using Insertion Sort algorithm.

Input:
Input begins with n ($1 \leq n \leq 2^{20}$) of number integers in the array in a single line and followed by n lines having an integer in each line.

Output:
Print the sorted array with one integer in each new line in ascending order and print the time taken (in milliseconds) to sort the array in a new line.

Example:
Input:
4
999999
0
-999999
1234

Output:
-999999
0
1234
999999
12.3 millisec (Note: This value is only representative)

Input/Output files and Sample source-code:

Sample input file: PES16sort_ip1.txt
Expected Output for the Sample input: PES16sort_op1.txt
Large input / output files:
PES16sort_ipRank1k.txt / PES16sort_opRand1k.txt
PES16sort_ipRank2k.txt / PES16sort_opRand2k.txt
PES16sort_ipRank4k.txt / PES16sort_opRand4k.txt
PES16sort_ipRank8k.txt / PES16sort_opRand8k.txt
PES16sort_ipRank16k.txt / PES16sort_opRand16k.txt
PES16sort_ipRank32k.txt / PES16sort_opRand32k.txt
PES16sort_ipRank64k.txt / PES16sort_opRand64k.txt
PES16sort_ipRank128k.txt / PES16sort_opRand128k.txt
PES16sort_ipRank256k.txt / PES16sort_opRand256k.txt

Note down the execution time for the above 9 inputs of random integers of size 2^{10}, 2^{11}, .., 2^{18}. Write a table of the durations in your observation book against the input size. Copy the durations taken by the Bubble Sort and Selection Sort exercises in the adjacent columns and compare the execution times.

Source-code of the solution:
PES16InsertionSort.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #13 (Week 8): Decrease and conquer: BFS and DFS

Find the number of components in the given undirected graph using BFS or DFS algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PESADA07/

Input:
The input begins with the number t of test cases in a single line (t <= 50). Each test case begins with the number n of the order of the adjacency matrix of the undirected graph (n <= 100) followed by the adjacency matrix. An adjacency matrix is represented in n lines having n integers (0s or 1s) separated by a space in each line.

Output:
For every test case print the number of the components the graph has in a new line. Print the total time taken in milliseconds.

Example
Input:
7

1
1

2
0 0
0 0

2
0 1
1 0

2
1 1
1 1

3
0 0 0
0 0 0
0 0 0

3
1 1 1
1 0 0
1 0 0

3

0 1 0
1 0 0
0 0 0

Output:
1
2
1
1
3
1
2
12.3 millisec (Note: This value is only representative)

Input/Output files and Sample source-code:

Sample input file: PES16DFS_ip1.txt
Expected Output for the Sample input: PES16DFS_op1.txt
Large input / output files: PES16DFS_ip2.txt / PES16DFS_op2.txt

Source-code of the solution:
PES16DFS.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #14 (Week 8): Decrease and conquer: Topological Sorting

Find a solution for a topological sorting problem using either Source-Removal algorithm or DFS-based algorithm.

Input:
Input begins with the number n of the order of the adjacency matrix of a directed graph (1 <= n <= 100) followed by the adjacency matrix. An adjacency matrix is represented in n lines having n integers (0s or 1s) separated by a space in each line. The directed graph may or may not be a dag (directed acyclic graph).

Output:
If the given directed graph is not a dag, print “NA” indicating there cannot be a solution for the topological sorting. Otherwise, print the solution for the topological sorting with the indices of the vertexes in order separated by a space. Print the total time taken in milliseconds.

Example:
Input:
5
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0

Output:
1 0 2 3 4 (or 0 1 2 3 4; any possible solution is accepted)
12.3 millisec (Note: This value is only representative)

Input/Output files and Sample source-code:

Sample input file: PES16TPO_ip1.txt
Expected Output for the Sample input: PES16TPO_op1.txt
Large input / output files: PES16TPO_ip2.txt / PES16TPO_op2.txt

Source-code of the solution:
PES16TPO.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #15 (Week 9): Transform and Conquer: Heap Sort

Write a program to sort an array of records (a record to have a register number, name of the student and his/her percentage) by their register numbers in ascending order using Heap Sort algorithm.

An online version of the below problem is at http://www.spoj.com/problems/PES16SO3/

Input:
Input begins with n ($1 \leq n \leq 2^{20}$) of number student records. The following n lines has a student record per line. A student record of a student has a unique register number (one alphanumeric word and 3 ≤ length of the word ≤ 10), his/her name (one alphabetic word without any spaces and 1 ≤ length of the name ≤ 20) and his/her percentage is an integer (0 ≤ percentage ≤ 100). The three fields of a record are separated by a spaces.

Output:
Print the sorted array with one student record in each new line in ascending order of student's register numbers and print the time taken (in milliseconds) to sort the array in a new line.

Example: Input:
10
CS012 Aarti 99
CS002 Adithya 97
CS001 Adithya 95
CS006 Darshan 98
CS010 Gautham 99
CS050 Karthik 97
CS200 Mohit 99
CS005 Mouli 99
CS020 Sneha 99
CS003 Vinay 100

Output:
CS001 Adithya 95
CS002 Adithya 97
CS003 Vinay 100
CS005 Mouli 99
CS006 Darshan 98
CS010 Gautham 99
CS012 Aarti 99
CS020 Sneha 99
CS050 Karthik 97
CS200 Mohit 99
12.3 millisec (Note: This value is only representative)

Algorithm:

```
HeapSort(H[1..n])
      HeapBottomUp(H[1..n])  //Construct heap
      for i ← n downto 2 do
            MaxKeyDelete(H[1..i])

HeapBottomUp(H[1..n])
      if(n ≤ 1) return
      for i ← ⌊n/2⌋ downto 1 do
            Heapify(H, i)

MaxKeyDelete(H[1..n])
      H[1] ↔ H[n]  //H[1] has the max element. Swap it with H[n].
      Heapify(H[1..n-1], 1)  //Sift down H[1] as much as possible

//For the sub-tree rooted at k, it sifts down H[k]
//as much as possible until it becomes a heap.
Heapify(H[1..n], k)
      if(2*k > n) return  //if H[k] is a leaf
      j ← 2*k  //j points to left child of H[k]
      if(j+1 ≤ n)  //if there exists a right child of H[k]
            if(H[j+1] > H[j]) j ← j+1
      if(H[j] > H[k])  //if greater child is greater than H[k]
            H[j] ↔ H[k]
            Heapify(H, j)  //Heapify the subtree rooted at j
```

Here are the videos of Prof. Channa Bankapur explaining the Heap Construction and Heapsort:
Heap and its array representation: https://youtu.be/j68JBXBaDlA
Top-down Heap construction: https://youtu.be/CzKE2C1UkxI
Bottom-up Heap construction: https://youtu.be/OuScmMXLNsU
Heapsort: https://youtu.be/9ED4kr8NEvQ

Input/Output files and Sample source-code:

Sample input file: PES16sort_records2_ip1.txt
Expected Outputs for the Sample input: PES16sort_records2_regno_op1.txt

Large input / output files:
PES16sort_records2_ip2.txt / PES16sort_records2_regno_op2.txt

Source-code of the solution:
PES16Heapsort.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #16 (Week 9): Space and Time Tradeoffs: Sorting by Distribution Counting

Write a program to sort an array of records (a record to have a register number, name of the student and his/her percentage) by their percentages in descending order using Sorting by Distribution Counting.

Input:
Input begins with n ($1 \leq n \leq 2^{20}$) of number student records. The following n lines has a student record per line. A student record of a student has a unique register number (one alphanumeric word and $3 \leq$ length of the word ≤ 10), his/her name (one alphabetic word without any spaces and $1 \leq$ length of the name ≤ 20) and his/her percentage is an integer ($0 \leq$ percentage ≤ 100). The three fields of a record are separated by a spaces.

Output:
Print the sorted array with one student record in each new line in descending order of student's percentage and print the time taken (in milliseconds) to sort the array in a new line. In case of more than one record has the same percentage, the relative order of the records need to be maintained (i.e., sorting has to be "stable").

Example: Input:
10
CS012 Aarti 99
CS002 Adithya 97
CS001 Adithya 95
CS006 Darshan 98
CS010 Gautham 99
CS050 Karthik 97
CS200 Mohit 99
CS005 Mouli 99
CS020 Sneha 99
CS003 Vinay 100

Output:
CS003 Vinay 100
CS012 Aarti 99
CS010 Gautham 99
CS200 Mohit 99
CS005 Mouli 99
CS020 Sneha 99
CS006 Darshan 98
CS002 Adithya 97
CS050 Karthik 97
CS001 Adithya 95
12.3 millisec (Note: This value is only representative)

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1     //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]            //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

Input/Output files and Sample source-code:

Sample input file: PES16sort_records2_ip1.txt
Expected Outputs for the Sample input: PES16sort_records2_percentages_op1.txt

Large input / output files:
PES16sort_records2_ip2.txt / PES16sort_records2_percentages_op2.txt

Source-code of the solution:
PES16DistributionCounting.c. (Solution will be shared with the students after assessment for the exercise freezes.)

(Week 10) On the spot problem to solve, which uses one or more the concepts learnt so far

Questions will be shared after the week.

Ex #17 (Week 11): Space and Time Tradeoffs: Horspool's or Boyer-Moore Algorithm

Write a program to search for a substring in a long text which matches with a pattern using Horspool's or Boyer-Moore Algorithm.

Input:
Input has two strings; text and pattern.

Output:
Print the array index of the first character of the first substring which matches the pattern, otherwise -1.

Print the total time taken in milliseconds.

Example:
Input:
abcbcbcda
bcd

Output:
5
12.3 millisec (Note: This value is only representative)

Horspool's Algorithm:

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

ALGORITHM *ShiftTable*$(P[0..m-1])$

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
//          filled with shift sizes computed by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//          or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shifts
i ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else  i ← i + Table[T[i]]
return −1
```

Input/Output files and Sample source-code:

Sample input file: PES16STR_ip1.txt

Expected Output for the Sample input: PES16STR_op1.txt

Large input file: PES16STR_ip2.txt

Expected Output for the large input: PES16STR_op2.txt

Source-code of the solution:

PES16Horspool.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #18 (Week 11): Dynamic Programming: Knapsack Problem

Write a program to solve 0/1 Knapsack problem using Dynamic Programming for a given list of n items with weight w_i and value v_i , and knapsack capacity W (1 <= n <= 100). Input would have n indicating the number of items, array of weights w, array of values v, and the knapsack capacity W. Output should have optimal value for the knapsack problem.

Example:
Input:
4
1 1 4 2
1 2 10 2
5

Output:
12

Example:
Input:
5
1 2 4 3 5
10 5 41 30 52
8

Output:
82

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                     Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

Input/Output files and Sample source-code:

Sample input file: PES16Knapsack_ip1.txt
Expected Outputs for the Sample input: PES16Knapsack_op1.txt
Large input / output files: PES16Knapsack_ip2.txt / PES16Knapsack_op2.txt

Source-code of the solution:
PES16KnapsackDP.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #19 (Week 12): Dynamic Programming: Warshall's algorithm

Write a program to find the transitive closure of a directed graph having n vertices using Warshall's algorithm (1 <= n <= 100). Given graph would be in the form of an adjacency matrix and the output should be a transitive closure matrix printed one row per new line and space separated elements within a row.

Example:
Input:
4
0 0 0 1
1 0 0 0
0 0 0 1
0 1 0 0

Output:
1 1 0 1
1 1 0 1
1 1 0 1
1 1 0 1

Input/Output files and Sample source-code:

Sample input file: PES16Warshalls_ip1.txt

Expected Outputs for the Sample input: PES16Warshalls_op1.txt

Large input / output files: PES16Warshalls_ip2.txt / PES16Warshalls_op2.txt

Source-code of the solution:
PES16Warshalls.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #20 (Week 12): Dynamic Programming: Floyd's algorithm

Write a program to find all pairs shortest paths in a directed weighted graph having n vertices using Floyd's algorithm (1 <= n <= 100). Given graph would be in the form of a cost adjacency matrix and the output should be a distance matrix printed one row per new line and space separated elements within a row. For a pair of vertices with no edge between them, a weight of 1000 would be given, which is practically infinity for us (you can be sure that no other edges and the shortest paths would be of length 1000 or more). And, cost of self loops are always zero.

Example:
Input:
4
0 1000 3 1000
2 0 1000 1000
1000 7 0 1
6 1000 1000 0

Output:
0 10 3 4
2 0 5 6
7 7 0 1
6 16 9 0

Example:
Input:
5
0 3 7 1000 1000
2 0 6 8 1000
9 2 0 1000 1
2 3 4 0 1000
1000 2 1 3 0

Output:
0 3 7 11 8
2 0 6 8 7
4 2 0 4 1
2 3 4 0 5
4 2 1 3 0

Input/Output files and Sample source-code:

Sample input file: PES16Floyds_ip1.txt

Expected Outputs for the Sample input: PES16Floyds_op1.txt

Large input / output files: PES16Floyds_ip2.txt / PES16Floyds_op2.txt

Source-code of the solution:
PES16Floyds.c. (Solution will be shared with the students after assessment for the exercise freezes.)

Ex #21 (Week 13): Greedy Technique: Prim's or Kruskal's algorithm

Write a program to find the cost of the minimum spanning tree of an undirected weighted graph having n vertices using Prim's or Kruskal's algorithm (1 <= n <= 100). Given graph would be in the form of a cost adjacency matrix and the output should be just the cost of the minimum spanning tree. For pair of vertices with no edge between them, a weight of 1000 would be given, which is practically infinity for us (you can be sure that no edges and the shortest paths would be of length 1000 or more). And, cost of self loops are always zero.

Example:
Input:
5
0 3 1000 7 1000
3 0 4 2 1000
1000 4 0 5 6
7 2 5 0 4
1000 1000 6 4 0

Output:
13

Example:
Input:
4
0 2 1000 3
2 0 4 5
1000 4 0 1000
3 5 1000 0

Output:
9

Ex #22 (Week 13): Greedy Technique: Dijkstra's algorithm

Write a program to find shortest path from a source vertex to a destination vertex in a directed weighted graph having n vertices using Dijkstra's algorithm (1 <= n <= 100). Given graph would be in the form of a cost adjacency matrix. Output should consist of the shortest distance an the shortest path (sequence of vertices of the shortest path). For pair of vertices with no edge between them, a weight of 1000 would be given, which is practically infinity for us (you can be sure that no edges and the shortest paths would be of length 1000 or more). And, cost of self loops are always zero.

Example:
Input:
4
0 1000 3 1000
2 0 1000 1000
1000 7 0 1
6 1000 1000 0
2
0

Output:
7
2 3 0

Example:
Input:
5
0 3 1000 7 1000
3 0 4 2 1000
1000 4 0 5 6
7 2 5 0 4
1000 1000 6 4 0
0
4

Output:
9
0 1 3 4