**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI**

First Semester 2011-2012

CS / IS C363 Data Structures and Algorithms

[Section 1]

Lab Sheet – 3 [Duration: 165 minutes]

General Instructions for Programming

Linux command to download the labsheet and support files to the local machine

```
$scp  -r <username>@blade3:~/lab2 .
```

Linux command to upload all the program files back to the server.

```
$ scp -r lab2 <username>@blade3:~
```

**`<username>` and password are given in each lab.**

1. All inputs to the program must be either (a) command line arguments (b) or read from a file (other than stdin). DO NOT READ anything from stdin and DO NOT USE PROMPTS like "Please enter a number …".
2. You are required to write the output to a file (other than stdout) and errors if any to a different output channel (stderr or another file).
3. Use standard C coding conventions for multi-file programs. Separate the following: interfaces of functions (use a ".h" file), data type definitions (use another ".h" file), ADT / algorithm implementation (use a ".c" file), and driver/test code (use another ".c" code). In general, each module has to be written in **separate** c files.
4. All files related to a lab **must** be put inside a single directory by the name of the lab (lab1, lab2, etc.).
5. Valid makefile must be present in the directory.
6. Ensure that all the code written by you are compiling correctly. Preferably use gcc with the options **`-W -Wall -O2`**, while compiling your code.
7. Do all your coding in the directory you downloaded, and upload the same directory back to server at the end of lab.

## Problem

**Brief:**

Shell sort is a sorting algorithm that is a generalization of insertion sort, which exploits the fact that insertion sort works efficiently on an input that is already almost sorted. It improves on insertion sort by allowing the comparison and exchange of elements that are far apart. The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted. Given a recursive pseudo code for shell sort, your task is to convert it to an iterative version using a single stack. **You must use the exact algorithm given here, and only do direct transformation to iterative version, using the stack technique.**

### Background:

The principle of Shell sort is to rearrange the file so that looking at every h^th element yields a sorted file. Such file is called h-sorted. If the file is then k-sorted for some other integer k, then the file remains h-sorted. For instance, if a list was 5-sorted and then 3-sorted, the list is now not only 3-sorted, but both 5- and 3-sorted. If this were not true, the algorithm would undo work that it had done in previous iterations, and would not achieve such a low running time.

### Pseudo code:

A rough pseudo code for shell sort is given below.

```
Procedure Shell_Sort_Rec (t[n], n, h)
temprory variables: aux, i
begin
    If h > 0 Then
    Begin
        If n > h Then
             begin
                  Shell_Sort_Rec (t,n - h,h); //recursive call
                  If t[n] < t[n - h] Then
                  Begin
                     aux:= t[n];
                     i := n;
                     Repeat while (i >= h) And (aux < t[i - h]);
                     Begin
                        t[i] := t[i - h];
                        i := i - h;

                     End;
                     t[i] := aux;
                  End;
              End;
        Shell_Sort_Rec (t,n,h / 3); // tail recursive call
    End;
End;
```

**Running Example of the algorithm.**

Consider a list of numbers like `[13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10]`. If we started with a step-size of 5, we could visualize this as breaking the list of numbers into a table with 5 columns. This would look like this:

```
13 14 94 33 82
25 59 94 65 23
45 27 73 25 39
10
```

We then sort each column, which gives us

```
10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45
```

When read back as a single list of numbers, we get `[10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45]`. Here, the 10 which was all the way at the end, has moved all the way to the beginning. This list is then again sorted using a 3-gap sort as shown below.

```
10 14 73
25 23 13
27 94 33
39 25 59
94 65 82
45
```

Which gives us

```
10 14 13
25 23 33
27 25 59
39 65 73
45 94 82
94
```

Now all that remains to be done is a 1-gap sort (simple insertion sort).

**Description:**

The program must take as input a file containing the list of numbers, say **input.txt**. The program must then sort the algorithm and store the result in an output file output.txt.
You are already provided a shell script named **random.sh** that will generate n random numbers for you.
For ex to generate 1000:
**$ sh random.sh 1000 > input.txt**

**Header file** for operations: (`SortOps.h`)

**MAXN**: defined initially as **10000**

(a) **`int shellSort(int t[], int n, int h)`**: This function implements the shell sort algorthm. For the initial call, the value of h is $2^{16}$ -1 and value of n is the total size of the array.
(b) **`int FillList(int t[], int size)`**: This function reads input file **input.txt** and stores it in the array t.
(c) **`int printList(int t[], int size)`**: This function writes the sorted result to the file **input.txt**.

**Please note that the value of n has to be less than or equal to 10000.**

**Step 1:**
Implement **`shellSort`** function for the recursive algorithm explained above in a file named **shell_rec.c**
**Step 2:**
Implement **`shellSort`** function by converting the recursive algorithm explained above into an iterative version using stacks, in the file named **shell_iter.c**
**Step 3:**
Implement the driver file **`sort.c,`** which reads the list of files from the files input.txt, performs the shellSort operation, and then stores the result in output.txt. This file will contain functions (b) and (c) described above, for the operations specified for those functions.
**Step 4:**
Compile the executable **`sort_rec,`** which uses the recursive variant. Observe the time required for this version of program
**Step 5:**
Compile the executable **`sort_iter,`** which uses the iterative variant. Observe the time required for this version of program
**Step 6:**
Compile the results observed by you in a text file named **results.txt**

**Support Files: `random.sh`, `SortOps.h`, sample `Stack` ADT files.**
**Deliverable: `StackOps.h`, `StackOps.c`, `Stack.h`, `shell_rec.c`, `shell_iter.c`, `results.txt`**

**While uploading, remove the input.txt files generated by you first and then only upload.**