**3353 Fall 2022**

**Program 1**

**Shell Sort**

Deadline: 9/24 (Sat) 11:59pm

You are given a file named "c1.h". It contains a class name C1, which is described below:

- Members (private):
  - int arr[3]: an array of 3 integers
- Constructors
  - C1(): New object, with all members set to 0
  - C1(int, int, int): New object, with the three parameters assigned to the array.
  - C1(const C1& t): Copy constructor, create a copy of t
- Overloaded operators
  - C1 & operator==(const C1& t): Assignment.
  - bool operator<(const C1& t)L return true is the object is less than t, false if otherwise
    - Currently, it is defined as having the smaller value in the first element of the array. Ties are broken with the second, and the third element respectively.
- Methods
  - void Print(): Print the value of the array to standard output
- Static (class) variable:
  - static int compareCount: initialize to 0. Will be incremented every time the "<" operator is called.
  - Notice that this is public, so you can change its value in your program.

You are not allowed to change the content of c1.h

**Part 1: Task – Shell Sort**

You are to implement the following function:

- void ShellSort(vector<C1>& varray, int code): Implement Shell Sort to sort the objects in varray in increasing order.
  - varray: the array to be sorted
  - code: an integer denoting how the hlist array (as in the source code) is to be formed:
    - 0: hlist should be [1] (i.e. essentially insertion sort)
    - 1: hlist should be [$2^k$ $2^{(k-1)}$ $2^{(k-2)}$ … 2 1] where k is the maximum number such that $2^k$ is still (strictly) smaller than the number of objects to be sorted. (Notice that in this case $1 = 2^0$)
    - 2: hlist should be [$2^k$-1 $2^{(k-1)}$-1 $2^{(k-2)}$-1 … 3 1] where k is the maximum number such that $2^k - 1$ is still (strictly) smaller than the number of objects to be sorted. (Notice that in this case $1 = 2^1 - 1$)
    - 3: hlist should be [$_kC_2$, $_{(k-1)}C_2$, $_{(k-2)}C_2$ … 3 1], where k is k is the maximum number such that $_kC_2$ is still (strictly) smaller than the number of objects to be sorted. (Notice that in this case $1 = {}_2C_2$)

- Your function should first output the value to hlist (via cout), before you do the sorting

You are welcomed to write a main program to test your function. You don't have to hand in that program.

**Task 2 – Investigate the sort's performance**

You are to investigate the performance of Shell Sort using various hlist values. In this program, you should use the number of comparisons (< method) that is called as a measurement of the performance (as number of swaps is bounded by the number of comparisons).

You should write a program that generate random list of objects of C1, sort them using your ShellSort() function (with different hlist) and record the number of comparisons.

You should tried for cases where there are 500, 1000, 1500, 2000, 2500, …. , all the way up to 5000 objects. For each case you should generate 24 different random sets, sort each of them and record the number of comparisons made (by examining the compareCount variable (remember to reset it to 0 before starting a second test). You should calculate the average number of comparisons for both cases and plot a graph to show the performance of various methods (the y-axis should be the number of comparisons, and the x-axis should be the number of objects to be sorted).

You should prepare two graphs, one for the average over all 24 runs; the other is for 20 cases (with the two best and worst runs removed).

**What to hand in**

You should upload a zip file that contains the following files

- A file named "shellsort.cpp" that contains ONLY the source code for the ShellSort() method
- A file named "main.cpp" which contains ONLY the code you run to test your case
- A pdf file named "report_<last name>_<first name>.pdf" that contains the two graphs. For each graph you should also have 1-2 paragraph to compare the performance of various choice of hlist for ShellSort(). Your discussion should focus not only on which one is faster, but how fast does the number of comparison grow with respect to the number of objects to be sorted. If you prefer, you can have more graphs to illustrate your point.

Notice these should be the ONLY things that is in the zip file. If you file contains more files (e.g. the whole build library, including object files and executables) than points will be deducted.