Clarification: Attendance at your 1hr. Wednesday or Friday lab/recitation section meeting is a required part of the course. Attendance is taken. Attendance and credits for recitation work assigned by the instructors are recorded and will contribute to your grade. Projects, lecture content, and exam design are based on the assumption that students participate in the 1hr section meetings. (If for some reason you miss a section meeting that you are registered for, you can participate in a different one if you inform your lead instructors Minoo Aminian and Stephen Englert.) The occasional training exercises are homework to be done outside of your section meeting. The office hour periods are for consulting with the professor or TAs to get assistance with project work, training exercises, studying, grade appeals, registration advice, etc.

Observing Activations with ddd

The objective of this week’s recitation lesson is to discuss and study simple recursive programs and to observe the activation records. Activation records, also known as (execution) stack frames are discussed in the beginning 6 pages into DSO Chapter 9 on recursion.

The instructions below describe how to observe these activation records using the write_vertical() function for an example (beginning of Chapter 9). So read all these pages first!

You will find an expanded version of the “vertical” program in ITSUNIX directory ~/acsi310/Lab4. Study the source file vertical.cxx Copy, build (remember to use the -g option!), and run it under the debugger.

To analyze a different program which your TA might assign or suggest, it would be wise to insert line breaks so that the bodies of control statements and each expression statement are on separate lines. That way, the debugger can stop at convenient spots. I also modified write_vertical() to include additional local extant (automatic) variables for you to see in each activation record.

As usual, set break main first, then activate run. Scroll to the body of write_vertical() and set another breakpoint, or simply command “break write_vertical”. Set a third breakpoint before the last line of code in write_vertical(). Finally, click or command continue to get the body of main() to run.

Observe how the program waits for you to type a positive number input. After you enter a sample number, the second breakpoint will be hit. At that point, pull down the “Status” menu and select “Backtrace”.

Give the “continue” command a few times and observe what happens. The “backtrace” is really a report of what activation records, also called stack frames, are currently in the execution stack. To see a more detailed record, give the “backtrace” and the “where” commands in the text command window.

Pull down the “Data” menu and command ddd to “Display local variables” and “Display Arguments” For a really detailed view of the current stack frame, give the text command “info frame” You can command ddd to put this frame information “on display” by checking “All about selected frame” in the “Status Displays” window that appears when you active “Status Displays” on the “Data” menu.
Observe how our one local variable write_verticals_LOCAL changes as you progress through running the program. Sometimes use the “next” command instead of “continue”. When does a display change indicate that the display shows a DIFFERENT VARIABLE (from a different activation)? When does it show a CHANGE OF VALUE in the SAME variable? Tip: Put the following expression on display: “&write_verticals_LOCAL” (Remember that “&” signifies “Address Of” when it is a prefix operator in front of a variable in an expression. It does not signify “Reference parameter” in this context.)

You probably will have to restart the program several times using the “run” command to observe what we are looking for. It is handy that all the breakpoint and display settings are kept for successive program runs.

Another important exercise: When the Backtrace shows several frames of write_vertical() activations, experiment with the “Up” and “Down” operations to observe the changes in the display window.

When you are debugging for real, the “Up” operation is handy for finding out about the parts of the program that called functions leading to the spot where the debugger stopped the program. This is important to know when you want to figure out why your program crashed and left a “core” file.

The “Down” operation lets you again view things closer to the bug or crash manifestation.

Warning 1: Going “Down” connotes going “deeper” into function activation nesting—But that means going towards the TOP of the execution stack.

A stack is also called a first-in last-out (fifo) store. The earliest started activations are the last to return and go away.

Warning 2: Bugs that corrupt the activation records in the execution stack can cause the debugger’s “Up” command to fail or give corrupted results. The same bugs cause manifestations to appear when activations return, since the corrupted stack frames are then accessed. A common way to corrupt activation records is to use an excessively large positive subscript in a local extant array. (That is because the top of the execution stack uses smaller memory addresses than the bottom, and increasing subscripts correspond to increasing memory addresses.) In other words, the function that stores data into an array element using an excessively large subscript may appear to work properly; but the functions that had called it, directly or indirectly, will fail miserably after the actually faulty function returns.

Software from crackers will corrupt an activation record in such a way that when that activation record is returned to, the corrupted data or return address causes the computer to execute code that the cracker supplied. So, to make your software be immune to this kind of exploit, make sure array bound subscript violations are impossible NO MATTER WHAT DATA is input to your software. Such successful protection is a bit more subtle than one might first think...

To try using ddd to find what caused a crash, next time you get a core file), run ddd as follows:

```sh
ddd executable-file-name core
```

(In the blank space below, you might want to write notes for future reference.)
Project 3

The objectives of this project are (1) giving practice with data structures consisting of a **dynamic partially-filled array** whose elements are pointers to strings, (2) implementing two sorting algorithms, one recursive, to operate on the array so each algorithm's performance (speed) can be observed and compared, and (3) further practice with modular software construction.

A build script named `build.sh` must be supplied, as it was for previous projects. When executed in its own directory, `build.sh` should produce an executable file named `sorterbench`. (Programs created or selected to use in computer performance measurements are called **benchmarks**.)

When `sorterbench` is executed, it runs first the “input phase”, which inputs lines of text from standard input (`cin`) and stores the non-empty ones in memory. It then runs the “command phase,” which performs any number of operations which the user had specified on the command line. (See below how to access the command line argument strings.) After the command phase, the program exits.

When you command a program to run with a shell, you can type space-separated strings after the program’s name\(^1\). They and the program name are **command line arguments**. For example, when you command:

```
$ sorterbench print selectionsort print
```

the command line arguments are the 4 strings “sorterbench”, “print”, “selectionsort” and “print”. These strings are made available to the program in an array of `char *` containing addresses of C-strings. The address of this array is the value of the 2nd argument (`argv` to `main(int argc, char *argv[])`). The C/C++ runtime support system marks the end of the pointer-filled array with the NULL pointer value. It also makes the first parameter to `main`, `argc` here, contain number of command line arguments; in this example, `argc`=4.

The analogous data structure will be used for storing the input lines, except the array of pointers and every C-string will be dynamically allocated; and the array of pointers will be partially filled. This data structure is sometimes called a **ragged array**.

---

\(^1\)You can command a program to run in the ddd debugger so a pop-up window appears for you to input the command line arguments. You can also type the command line arguments after the “run” keyword in the ddd text input/output window.
1 Input Phase

During the input phase, `sorterbench` reads (from standard input, `cin`) lines of input, stores non-empty lines as dynamically allocated strings, and stores the address of each string in a dynamic partially filled array. We suggest you use C-strings for storing the input lines; but you may take the option to use STL strings if you wish to study and learn their details on your own. In any case, the dynamic partially filled array MUST be filled with pointers or else the experimental sorting times will be excessive.

Here is a problem: The number of input lines is unpredictable. Here is the solution: The dynamic partially filled array will grow (by doubling in capacity) when it needs more capacity.

The lines should appear in the array in the order they were input.

Each input line is up to 90 characters in length including the newline. Your `sorterbench` could read them into a char array using the code given below which is similar to that given for Project 2.

Unlike Project 2 data input, the end of the series of input lines should be detected as the “end-of-file” condition on standard input (`cin`). For simplicity, you can terminate the input loop by code that detects this condition and other conditions such as an input line being too long. (We will not test your programs on excessively long input lines.)

**IMPORTANT FOR GETTING STARTED:** You must type one or more non-empty input lines to begin testing your program.

You must know to **TYPE Control-d on a line by itself** to terminate the inputting of text lines. (Under the default terminal settings of UNIX, Control-d typed into a terminal window activates the “end-of-file” condition on standard input, which is what C++’s `cin` uses.) **WARNING:** Do NOT code anything for detecting Control-d. The Unix terminal system does that for you.

**An unusual requirement:** Your program should “ignore”, i.e., skip over and not store, empty input lines (which consist of the newline character only). Remember that `istream::getline()` replaces the newline with the null character when it copies the input line to your programs buffer array. The reason is you will be invited to try your program on large text files supplied by Project Guttenberg [http://www.gutenberg.net](http://www.gutenberg.net) and they have a lot of boring empty lines. It is amusing to see just how fast certain algorithms on computers really are, and what happens to great literature plus publisher’s notices when their lines are sorted together lexicographically. More seriously, this project is the start for writing concordance computing software, a higher level assignment.
To help you get started, here is the code for the loop to read the input lines:

```c
const int INBUFSIZE = 90;
char inbuf[INBUFSIZE];
int n_lines = 0;
// Allocating declaration of the partially-filled array object
// where the input lines will be stored...
/* RECORD THE TIME BEFORE INPUT READING STARTS */
while( cin.getline(inbuf, INBUFSIZE) )
{
    if( inbuf[0] != '\0' )
    {
        // INV: The input line is non-empty, since the first char!=the null char.
        /* Write code here to allocate a string of the right length
         * and store its address just after the end of filled part
         * of your partially filled array.
         */
        n_lines++;
    }
}
/* RECORD THE TIME WHEN INPUT READING ENDS */
cout << "sorterbench read " << n_lines << " non-empty data lines" << endl;
/* PRINT THE REPORT OF WHEN THE ABOVE OPERATION STARTED AND ENDED,
   AND THE AMOUNT OF TIME USED. */
cout << "Time used for input reading=" << /* expression for time used */;```

See below the instructions to access, store and calculate with the time-of-day.
Command Phase

The pseudo-code for the command phase is as follows (this code would be inside the body of main(int argc, char *argv[])):

```cpp
int i_arg = 1; //index of the first command-line argument to process,
    //since argv[0] is the program name.
while( argv[ i_arg ] != NULL )
{
    if ( strcmp( argv[i_arg], "print" )==0 )
    {
        /* RECORD THE TIME WHEN PRINTING STARTS */
        /* ... Print all the lines in the partially filled array */
        /* RECORD THE TIME WHEN PRINTING ENDS */
        /* CALCULATE THE TIME USED FOR PRINTING */
        /* PRINT THE REPORT OF WHEN THE ABOVE OPERATION STARTED AND ENDED,
           AND THE AMOUNT OF TIME USED. */
        cout << "Time used for printing=" << /* expression for time used */;
    }
    else if ( strcmp( argv[i_arg], "selectionsort" )==0 )
    {
        /* RECORD THE TIME WHEN SELECTION SORTING STARTS */
        ....
        /* Run the selection sort algorithm on the partially filled array */
        /* RECORD THE TIME WHEN SELECTION SORTING ENDS */
        /* CALCULATE THE TIME USED FOR SELECTION SORTING */
        /* PRINT THE REPORT OF WHEN THE ABOVE OPERATION STARTED AND ENDED,
           AND THE AMOUNT OF TIME USED. */
        cout << "The wallclock ..."); // See instructions for the exact format required.
    }
    else if ( strcmp( argv[i_arg], "mergesort" )==0 )
    {
        /* code similar to the above but for mergesort */
    } else if ( strcmp( argv[i_arg], "..." )==0 )
    /* optional additional algorithms to experiment with */
    { /* code similar to the above but for optional algorithms */
    } else
    {
        cerr << argv[0] << ": Unrecognized argument." << endl;
        return( 1);
    }
}
//INV: The commands carried out are those given by argv[i_arg] and
// its predecessors.
i_arg++; /*argv[i_arg] points to first UNPROCESSED argument OR it
    // equals NULL. */
```
cout << argv[0] << " exiting normally. Enjoy your sorting." << endl;
return( 0 );
} // end of main()

Details about sorting

The selection sort and recursive mergesort algorithms described in class and in DSO chapter 13 should be implemented by code your write yourself, not by library sorting functions.

C-string library functionstrcmp() or equivalent (for STL) should be used to compare strings. It does so-called lexicographic comparisons on strings based on numerical comparisons of the ASCII character codes. The lexicographically smaller strings should be be printed before the larger strings when the sorted array is printed.

The sorted data should replace the current data. So, if the user names two sort algorithms in a row, the sorted data produced by the first is sorted (uselessly) again. (We can therefore time a storing algorithm on input that is already sorted.)

Note that the sorting operations must not print the lines, either before or after sorting. Instead, the program should print a report of the time that elapsed during the sorting operation.

Selection sort merely moves string addresses from one array element to another. Mergesort involves dynamically allocating more arrays for temporary use...details are given in the textbook. So, one advantage of selection sort over mergesort, when the data is in an array, is that temporary arrays are not used. Your selection sort implementation must not use new at all. As shown in DSO Ch. 13, new and delete[] will be used in mergesort. (Later in the semester you will learn \(O(n \log n)\) sorting algorithms that do not arrays or other large structures in addition to the structures to store the inputted data. Ambitious students might add these algorithms to their sorterbenchs to find out how much they really help.)

Selection sort in an array uses two index variables: One keeps track of the position within the sorted sequence where the element now being selected will finally be put. The other keeps track of the positions of elements to be compared with the so-far-smallest element among those not placed in their final positions. See the Lecture notes on using selection sort to sort the characters in an array. Both index variables are used to traverse the array.

Selection sort operates by comparing certain pairs of sequence items and swaps them if they are out of order. The most efficient way to swap C-strings (or other implementations) in a linked list is to swap the addresses of the strings stored in the array. So, you must use this most efficient way.

Mergesort

(Study recursion and the recitation lesson on it first!)

The mergesort operation provides the same functionality as selectionsort, except it uses the mergesort algorithm to do the sorting. Mergesort for arrays is described in section 13.2 of DSO.

For this project, the mergesort algorithm must be implemented using recursive code. Recursion was introduced in the lecture. When we apply, in the simplest way, the “divide-and-conquer” pattern or principle to the sorting problem to obtain a recursive sorting algorithm, the algorithm we get is mergesort. This topic will be covered in the next lecture.
Like for `selectionsort`, the C-string library function `strcmp()` or its equivalent must be used to compare the C-strings or other strings for sorting. The string comparison operation used in the merge operation **must be exactly the same** as the one used for selection sort, so that differences in elapsed time will not depend on how the strings were compared.

Of course, `mergesort` must report the the `gettimeofday()` results and their difference just as `selectionsort` did, except for name of the algorithm is “mergesort”.

`mergesort` will be tested the same way as `selectionsort`

We will also check that your actually implemented and called a recursive `mergesort` algorithm, rather than cheat by calling the selection sort code again (or a library function)! (We will also be able to tell by the time reports.)

**Timing in Unix/Linux**

To capture a record of what time it is either before or after doing one operation, call the `gettimeofday()` function. This time value will eventually be printed and used to calculate elapsed time. (Why shouldn’t it be printed before the operation is started?)

Use the following following code: (Study `man gettimeofday` output to find out what it means.)

```
#include <sys/time.h>
#include <iostream>
using namespace std;
...
struct timeval time_started;
struct timeval time_finished;
gettimeofday( &time_started, 0 );
...//do the operation
gettimeofday( &time_finished, 0 );
```

AFTER the operation is completed, print the (1) time-of-day before the operation had started, (2) the time-of-day after the operation had finished and (3) the difference between these two times. Use code like::

```
cout << "Time before selectionsort="
 << time_started.tv_sec << "seconds+
 << time_started.tv_usec << "microseconds" << endl;
cout << "Time after selection sort=" //..... etc..
```

This will enable you to observe exactly how time elapsed during the period when the computer was busy running your selection sort code. We can also check that your calculation of the elapsed time by subtraction of the start minus the finish time is correct.

To compute the number of microseconds of time that elapsed between the two time of day reports, you will have to code a combination of addition, subtraction and multiplication operations on the two fields of each of the two `struct timeval`'s filled by `gettimeofday()`. (Your program will have to multiply something by a million (1000000) since there are 1 million microseconds in 1 second.) Print the result exactly as follows:

The wallclock time to run `selectionsort` on `<n>` elements was `<mmm>` microseconds.
Here <n> signifies the number of lines in the sequence (that was sorted) and <mmmm> is the number of microseconds, both printed as decimal integers.

You and our staff will be able to test your sorting algorithm implementations by specifying print, selectionsort (or mergesort), and print again as command line arguments, and then typing some lines.

When you or we type the lines, we can terminate the input operation by typing Control-d on a line by itself.

If a print operation immediately after input doesn’t report the very same non-empty lines that were input for the test, your program will be rejected right away and get 0 points for functionality.

**How to Do This Project; Some Graded Items!**

All project specifications and rules from the handout on doing programming projects, and Project 2 apply to Project 3 as well. (RCS databases, build script, header/implementation/main file separation, pre and post conditions to specify function documentation written in header files, turning in a directory, etc.)

Full credit for modularization requires at least 3 different implementation files for (1) the user interface/input phase reader (2) the selection sort algorithm implementation and (3) the merge sort algorithm implementation. You are encouraged to use more, but that’s optional.

You might fashion your Project 3 program from your Project 2 program, or start from scratch.

Reminder: selectionsort and mergesort MUST NOT PRINT the sorted lines, only the time reports. To see the sorted results, you must put print in the command line. My reasons are to make time reports for input, sorting and printing be separate and to make the count of lines sorted and time reports easy to see when the number of lines is quite large (thousands or more).

Submit a single directory to project name Proj3 using “turnin-csi310 -c csi310 -p Proj3 Directory ”. Verify with “turnin-csi310 -v -c csi310 -p Proj3”.

**Future Work**

You can easily make this project program sort the lines from a file and report the time to do it by:

1. Prepare the file named for example “testinput.txt”

2. Supply the file as input to your project program by redirecting the standard input to sorterbench from the file. This is done by the following shell command:
   
   sorterbench mergesort < testinput.txt
   
   If you followed the instructions, only the line count, day time and elapsed time reports will be printed.

   **WARNING:** Do NOT write code to detect the “<” character or use the filename!! The shell will do that for you. The shell will remove the “<” and filename before preparing the command line arguments for access by main(). The shell will try to open the input file. If that is successful, the shell will arrange to run your sorterbench program so standard input (i.e., cin) reads from the input file instead of the terminal window.
3. If you actually want a sorted output file, command:

```
sorterbench mergesort print < testinput.txt > testoutput.txt
```

(and the edit away the time reports if you don’t want them)

Making your program into a concordance generator: Add an operation to (logically) replace each input line with many lines: One line for each occurrence of a word within one input line. The new (logical) lines will have a copy or reference to the word of the occurrence at the beginning. You should also change any upper case letter there to the corresponding lower case letter. When the new lines are sorted, you will get a concordance! It would help to attach the line number to each new logical line and print that too. An example of input and the resulting logical lines is:

Things equal to one thing
are equal to one another

things: Things equal to one thing (1)
equal: Things equal to one thing (1)
to: Things equal to one thing (1)
one: Things equal to one thing (1)
thing: Things equal to one thing (1)
are: are equal to one another (2)
equal: are equal to one another (2)
to: are equal to one another (2)
one: are equal to one another (2)
another: are equal to one another (2)

The result of sorting is:

another: are equal to one another (2)
are: are equal to one another (2)
equal: Things equal to one thing (1)
equal: are equal to one another (2)
one: Things equal to one thing (1)
one: are equal to one another (2)
thing: Things equal to one thing (1)
things: Things equal to one thing (1)
to: Things equal to one thing (1)
to: are equal to one another (2)

(Upper case letters are before lower case letters in ASCII code numeric order.)

Here is a more cool way to format the output:

```
are equal to one >another
   >are equal to one another
   Things >equal to one thing
      are >equal to one another
      Things equal to >one thing
         are equal to >one another

etc.
```