Problem Statement

In this programming assignment you will implement a system sort utility that contains a fully optimized quick sort algorithm. This system sort utility will give you a chance to apply many patterns and reusable components that we covered in class.

Part One

The heart of this assignment is the sort() function, which you must write. In part one, you will implement a generic sort() template function using C++ templates. The interface for this function should look as follows:

```cpp
// Sort <array> in place.
// ARRAY::size() determines the length of <array>.
// ARRAY::operator[] to get/set array values
// ARRAY::TYPE defines array elements and temporary variables.
// ARRAY::TYPE operator< orders the data.

template <class ARRAY>
void sort (ARRAY &array);
```

You can use your generic sort routine in conjunction with the generic Array class you wrote earlier in the semester as follows:

```cpp
{ 
    Array<int> int_array (10);
    Array<double> double_array (100);
    Array<Employee> employee_array (1000);

    // Randomly initialize int_array and double_array.
    init (int_array, double_array, employee_array);

    // Make copies.
    Array<int> int_copy (int_array);
    Array<double> double_copy (double_array);
    Array<Employee> employee_copy (employee_array);

    sort (int_array);
    sort (double_array);
    sort (employee_array);

    // Print the output and make sure it’s sorted.
}
```

You should use insertion sort since it’s very simple and you’ll need it for part two. If you’re having problems understanding how the templatized sort() function works, I recommend that you write a non-templatized sort() function first, which just sorts and array of integers, i.e.,
// Sort <array> in place.
void sort (Array<int> &array);

and then generalize it so that it works properly with the ARRAY template described above.

If you finish this section early I strongly suggest that you start on part 2 since it will require much more
effort to complete.

**Part Two**

In part two, you will modify your sort() function to implement quick sort. **Quick sort** is a comparison
and exchange sort that has an average case running time of $O(n \log n)$. The quick sort algorithm can be found
in most algorithm and data structure textbooks, as well as via online search engines.

In addition to being correct, your solution must also implement the following **four** optimizations to the
basic quick sort algorithm:

- **Use an explicit stack**: Use an explicit stack, rather than recursive procedure calls, to store unsorted
  (LO, HI) pairs. This reduces the overhead associated with recursive calls.

- **Guarantee $O(\log n)$ stack utilization**: push the larger of the 2 partitions onto the stack, and always
  sort the smaller partition first. This guarantees an upper limit of $O(\log n)$ stack space required for the
  algorithm. Your stack should have the following interface:

  ```cpp
  template <class T, size_t SIZE>
  class Fixed_Stack
  {
  public:
    Fixed_Stack (void);
    // ... typical push()/pop() operations
  
  private:
    size_t top_;    
    T stack_[SIZE];
  }
  ```

  This “fixed sized stack” can be a variant of the AStack you implemented in programming assignment 2,
  with an additional optimization for the fact that you needn’t have more than $\log_2 \left( \text{BITS} \left( \text{sizeof} \left( \text{size}_t \right) \right) \right)$ elements on the stack. To make your code portable, therefore, you need to parameterize
  your Fixed_Stack using the BITS (sizeof (size_t)) macro from /usr/include/values.h
  on UNIX. Those of you using Windows will need to include the following in your program somewhere:

  ```
  #define BITSPERBYTE 8
  #define BITS(type) (BITSPERBYTE * (long)sizeof (type))
  ```

- **Use median-of-3 pivot selection**: Choose the median-of-3 pivot element for the partition phase. This
decreases the likelihood of consistently picking the worst pivot value. Note that the pivot selection
strategy is likely to change, so design your program accordingly, i.e., using the Strategy pattern we
discussed in class.

- **Use insertion sort for small items**: The overhead associated with quick sort increases as the partition
  size becomes very small.\(^1\) Thus, discontinue partition ordering once the size of the partition falls below

---

\(^1\)This is especially true of recursive implementations.
some pre-determined limit (e.g., 10 elements). This will result in a collection of partitions that are
unsorted within a partition and ordered between partitions. Use a single cleanup pass using insertion
sort to finish sorting the data. Insertion sort works well on data that is almost completely sorted.

If you finish this section early I strongly suggest that you start on part 3 since it will require substantially
more time that the first two parts.

**Extra Capabilities for CS 291 Students**

If you are signed up for CS 291 please add the following extension to the original assignment:

1. Add a second “function object” generic parameter to the `sort()` function that defines `operator()`
   and can be used to control whether the sorting is done in ascending or descending order. This should
   work similar to the `less` and `greater` function objects in STL, e.g.,
   ```cpp
   // Sort in descending order: note explicit ctor for greater
   sort (int_array, greater<int> ());
   // Sort in ascending order: note explicit ctor for greater
   sort (double_array, less<double> ());
   ``
   By default, your `sort()` function should use the `less` function object.

**Part Three**

Your system sort utility should be able to process its input data very efficiently. In particular, it should
perform the minimal number of dynamic allocations for the *entire* data stream from stdin. To implement this
you’ll need to create an `Input` class that reads an arbitrarily long sequence of characters from stdin or cin
and returns a copy of these characters as a dynamically-allocated buffer. The trick to making this work is to
define a recursive helper method that stores fixed portions of the input stream on the run-time stack. The class
handouts describes the interface for the `Input` class and outlines how it works.

**Part Four**

Use the `sort()` function you wrote for part 1 and part 2 in conjunction with the reusable `Input` class
you wrote for part 3 to write a general-purpose system sort utility that sorts all lines from stdin and writes the
result to stdout. A line is a sequence of characters terminated by a newline. Your sort utility will therefore
need to sort an input sequence consisting of an array of characters.

The following is the programmer’s manual page for your system sort utility, which is inspired by the Linux
system sort utility. Your solution must support the options described below and provide at least this level of
functionality.

```
NAME
   sort - sort a file

SYNOPSIS
   sort  [-c<num> -k<num> -f -n -r -t<char>]

DESCRIPTION
   Sort sorts lines from stdin and writes the result to stdout.
   Default sort key is an entire line, which is a sequence of
   characters terminated by a newline. Default ordering is
   lexicographic by bytes in machine collating sequence. The
```
ordering is affected globally by the following options, one or more of which may appear. All options are prefixed by the ‘-‘ delimiter.

f  Fold upper case letters onto lower case (i.e. ignore case).

n  A numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value.

r  Reverse the sense of comparisons (i.e., sort by descending rather than ascending order).

k  Begin sorting at specified key <num>. Fields are nonempty nonblank strings separated by the field separator character (see the following option).

t  Specifies the field separator character, which defaults to ’ ’ (ASCII blank).

c  Begin sorting at the specified column <num> (range starts at 1) There must be NO space after the ’c’ and before the column number. If this option is specified along with the field option (-k), then the designated column is treated as the starting point within the designed field.

EXAMPLES
Print in reverse numerical order the contents of the sixth field of each line in the file.

    sort -r -n -k6 < file

Print in alphabetical order treating the 10th column as the beginning of the sort key. Capitalized words do not differ from uncapitalized.

    sort -f -c10 < file

BUGS
Strictly an "in memory sort"; only deals with fixed sized files of input that do not exceed the internal buffer sizes.

Implementation Issues for Part 4

The following are implementation issues you’ll need to consider when implementing part 4 of this assignment.

- **Building an Access Table** – Another key abstraction you’ll need for part 4 of this program is an Access_Table class. This class is an adapter for the type parameter expected by the sort() function you developed in parts 1 and 2. The Access_Table stores all the character data received from stdin by the Input class.\(^2\) In addition, the Access_Table also contains an Array that stores the beginning of each “line” from the stdin. The reason for this scheme is to save time and space by allowing efficient storage of variable length lines, as well as to permit fast swapping of lines by simply exchanging Array entries (which point to lines in the Access_Table buffer) rather than copying the lines themselves.

\(^2\) Hint, translate all newlines into the ASCII NUL character.
• **Precomputing the Access Table offsets** – In addition to storing the beginning of a line, the `Access Table Array` also stores the beginning field where the comparison takes place. The functionality of this system sort derives from its ability to specify sort keys starting at certain fields and columns. To save time during later phases of the sort, the offset of the beginning field is pre-computed when the data is initially read into the `Access Table`. All comparisons of lines made by the `sort()` function begin with the starting field (which may or may not be different from the start of the line).

• **Developing I/O adapters** – The trick to making all this work is to define an implementation of global `operator<(const Array_Adapter &, const Array_Adapter &)` that adapts the interface expected by the `sort` routine to an interface that supports the flexible comparison and exchange functionality provided by the `Access Table`. In this case, where `Array_Adapter` is the `TYPE` of the class `Array<T>` that `sort` sees by using `ARRAY::TYPE`.

We will discuss the various issues and the design patterns that can be applied to address them in class based on class handouts.

---

3 See the manual page shown above for details.