# CS 291: Software Design Studio

Final Programming Assignment
Part 1 due Monday, November 15th, 2004
Part 2 due Monday, December 29st, 2004

# Problem Statement

Learning how to write effective test programs is an important skill. The objective of this assignment is to develop a set of C++ classes that enable a tester to check efficiently whether the sort routine you wrote for the previous assignment works correctly. This assignment will give you more experience working with C++, abstract data types (ADTs), program families, and design patterns.

From a high-level, the system will work as follows:

- Two arrays are given as input. One array is the original array that was passed as input to the sort routine. The is the output of the sort routine (*i.e.,* it is the "potentially" sort array).
- The main purpose of your program is to check that the potentially sorted array is actually an ordered permutation of the original input array. This is surprisingly tricky since the sort routine can fail for many reasons (*e.g.,* it may fail to sort correctly or it may accidentally change values).

The general structure of your program should look something like this (lots of detailed omitted):

```
int main (int argc, char *argv[])
{
  size_t size = atoi (argv[1]);

  Array<int> original (size);
  Array<int> potentially_sorted (size);

  randomly_generate_input_data (original);
  sort (original, potentially_sorted);

  if (check_sort (original, potentially_sorted, size) == 0)
    cout << "array is not sorted!" << endl;
  else
    cout << "array is sorted correctly" << endl;
  return 0;
}
```

The following are constraints on your solution (*i.e.,* the "forces"):

- Your solution should be time and space efficient. For example, it should not take more time to check than to sort in the first place!
- Do not assume the existence of a "correct" sorting algorithm. In particular, the algorithm you select to solve this problem much be as simple or simpler than writing the sort routine in the first place (Quis costodiet ipsos custodes).
- Your solution should work for any type of data, but you can assume you'll only be sorting integers for the purposes of your test program (this makes certain things easier...).

# Observations

Depending on the properties of the data in the original array, you will want to use different strategies to check whether the array was sorted correctly. For example, if the data values in the array are "small" integral values (where small relative to the size of the array) you'll use a different strategy than if the values are large and/or non-integral values. Likewise, if the original array has no duplicate values you can use a different strategy than if it has duplicates. Thus, once you start to design your solution you'll find that there it forms a "program family." This gives us an opportunity to use design patterns and C++ classes and templates to reuse as much effort as possible across multiple different strategies.

The key to making an efficient and flexible solution is to develop a "search structure" abstract base class (ABC) that contains pure virtual methods such as `insert` and `remove`. The following is an example:

```
template <class T>
class Search_Structure
// TITLE
//   Defines an abstract base class for inserting and removing
//   elements of type <T> from a collection.
//
// DESCRIPTION
//   This class only defines an interface and relies on derived
//   classes to supply an implementation.
{
public:
  virtual int insert (const T &new_item) = 0;
  // Insert <new_item> into the search structure.

  virtual int remove (const T &existing_item) = 0;
  // Remove <existing_item> from the search structure.
  // Return 0 if is there, else -1.

  virtual ~Search_Structure (void) = 0;
};
```

You will implement different subclasses of this search structure ABC, depending on the characteristics of the array data you are checking.

The following are the types of subclasses that you will need:

- `Range_Vector` – This version is useful for sorting "small" ranges of integral values, where small is defined as <= total size. This solution has $O(N)$ time complexity and is space efficient as long as the range is reasonable.

  ```
  class Range_Vector : public Search_Structure<long>
  {
    /* ... */
  };
  ```

- `Binary_Search_Nodups` (version 1) – This version does works for arrays with arbrary ranges. However, it not handle duplicates and has $O(n \lg n)$ time complexity and is space efficient.

  ```
  template <class T>
  class Binary_Search_Nodups : public Search_Structure<T>
  {
    /* ... */
  };
  ```

- Binary_Search_Dups (version 2) – This version also works for arrays whose values have arbitrary ranges. It does handle duplicates and remains $O(n \lg n)$ in its time complexity. However, it maybe somewhat less space efficient than version 1 (depending on number of duplicates and how clever you are ;-)).

  ```
  template <class T>
  class Binary_Search_Dups : public Search_Structure<T>
  {
    /* ... */
  };
  ```

## Part One

For part one of this assignment you'll implement the Range_Vector, Binary_Search_Nodups, and Binary_Search_Dups.

## Part Two

For part two of this assignment you'll implement the driver program that performs the checksort strategy.