Tampere University of Applied Sciences

Search and Sorting integers C++

Data structures and algorithms

Leo Suzuki

Class project
May 2024

IT-engineering / SW

Project repository: https://github.com/LeoSuzu/Search_and_Sort_Cpp.git

CONTENTS

1 INTRODUCTION

This program is designed to perform and analyze the performance of various sorting and searching algorithms on a list of randomly generated numbers. It includes functionality for both sequential and binary searches, as well as insertion sort and quicksort algorithms. By comparing the performance of these algorithms under different conditions, this program aims to provide insights into their efficiency and suitability for different types of data and use cases.

Objectives

The primary objectives of this program are:

- To implement and test common sorting algorithms (insertion sort and quicksort).
- To implement and test common searching algorithms (sequential search and binary search).
- To generate random lists of integers for sorting and searching operations.
- To measure and compare the performance of these algorithms in terms of time complexity and the number of operations performed.
- To provide detailed output and analysis of the sorting and searching processes.

Features

- **Random List Generation**: The program can generate lists of random integers of specified sizes.
- **Sorting Algorithms**: Includes insertion sort and quicksort, allowing for a comparison of their performance.
- **Searching Algorithms**: Includes sequential search and binary search, with performance comparison.
- **Timing and Measurement**: Utilizes a timer to measure the time taken by each algorithm, providing accurate performance metrics.
- **Result Analysis**: Outputs detailed results including average time per run, the number of comparisons, and the number of assignments for each algorithm.

2 USAGE

The program is interactive and prompts the user for the necessary inputs such as the size of the list, the number of cycles for sorting, and the number of runs for searching. It then performs the specified operations and outputs the results in a structured format. Users can choose to run predefined test sizes or input custom values.

3 INSTALLATION GUIDE

Prerequisites

Before you can build and run the program, ensure you have the following installed on your system:

- **C++ Compiler**: Ensure you have a C++ compiler installed, such as **g++** (GNU Compiler Collection) or **clang**.
- **CMake**: A cross-platform build system that can generate build files for your platform. You can download it from cmake.org.
- **Make**: A build automation tool which is often included with development tools on Unix-based systems.

Steps to Install and Build the Program

1. **Clone the Repository:** First, clone the repository from the source.
   git clone https://github.com/LeoSuzu/Search_and_Sort_Cpp.git

   cd Search_and_Sort_Cpp
2. **Create a Build Directory**: It's a good practice to build the project in a separate directory.
   mkdir build

   cd build
3. **Generate Build Files**: Use CMake to generate the build files.
   cmake ..
4. Build the Project: Run the build command.
   make
5. **Run the Executable**: After a successful build, you will find the executable named **<YourProgramName>** in the **build** directory.
   . / YourProgramName

4 RUNNING THE PROGRAM

Once you have built the project, you can run the program using the executable **YourProgramName**. The program is interactive and will prompt you for various inputs such as the size of the list, the number of sorting cycles, and the number of search runs.

Example Usage

1. **Start the Program**: Run the executable from the terminal.
   . / YourProgramName
2. **Input List Size**: The program will prompt you to enter the size of the list.
   Size of List: 1000
3. **Input Number of Sorting Cycles**: Enter the number of cycles for sorting operations.
   How many runs / algorithms: 10
4. **Input Number of Search Runs**: Enter the number of runs for searching operations.
   How many runs / algorithms: 10
5. **View Results**: The program will execute the sorting and searching algorithms, then output the performance results.

   Initializing list

   Time to init list: 0.00123 seconds

   Sequential search cycle: 1

   ...

   Binary search cycle: 1

   ...

6. **Results Summary**: At the end, the program will provide a summary of the performance metrics for each algorithm.

5 ALGORITHMS

Sequential search algorithm:

```
searchResult seqSearch(const List<unsigned int> &list, unsigned int
key) {
    // Create an instance of searchResult to store the search results
    searchResult res;

    // Temporary variable to store the current element from the list
    unsigned int temp;

    // Loop through each element in the list
    for (unsigned int i = 0; i < list.size(); i++) {
        // Retrieve the element at the current index and store it in
temp
        list.retrieve(i, temp);

        // Increment the comparison count
        res.comparisons++;

        // Check if the retrieved element matches the key
        if (temp == key) {
            // If a match is found, set the success flag to true
            res.success = true;

            // Store the position where the key was found
            res.position = i;

            // Exit the loop as the key has been found
            break;
        }
    }

    // Return the search result containing success flag, position, and
comparisons count
    return res;
}
```

Binary search algorithm

```
searchResult binSearch(const List<unsigned int> &list, unsigned int
key) {
    // Create an instance of searchResult to store the search results
    searchResult res;
    res.success = false; // Initialize the success flag to false

    // Initialize the bottom and top indices for the binary search
    int bottom = 0;
    int top = list.size() - 1;
    unsigned int x; // Temporary variable to store the current element
from the list

    // Loop to perform binary search
    while (bottom <= top) {
        // Calculate the middle index
        int mid = (bottom + top) / 2;

        // Retrieve the element at the middle index and store it in x
        list.retrieve(mid, x);

        // Increment the comparison count
```

```
        res.comparisons++;

        // Check if the retrieved element matches the key
        if (x == key) {
            // If a match is found, set the success flag to true
            res.success = true;

            // Store the position where the key was found
            res.position = mid;

            // Return the search result immediately
            return res;
        }
            // If the retrieved element is greater than the key,
adjust the top index
        else if (x > key) {
            top = mid - 1;
        }
            // If the retrieved element is less than the key, adjust
the bottom index
        else {
            bottom = mid + 1;
        }
    }

    // Return the search result containing the success flag, position,
and comparisons count
    return res;
}
```

Insertion sort algorithm

```
// Function to perform insertion sort on a list
sortResult insertionSort(List<unsigned int> &list, Timer &timing, bool
print) {
    // Create an instance of sortResult to store the sorting results
    sortResult res;
    unsigned int listToSort, listToCompare;

    // Reset the timer to start measuring the sorting time
    timing.reset();

    // Set the algorithm type to 'I' for Insertion Sort
    res.algorithm = 'I';

    // Start the insertion sort algorithm
    for (auto i = 1; i < list.size(); i++) {
        // Retrieve the element at position i to be sorted
        list.retrieve(i, listToSort);

        // Initialize j to the position before i
        int j = i - 1;

        // Inner loop to compare and shift elements
        while (j >= 0) {
            // Retrieve the element at position j to compare
            list.retrieve(j, listToCompare);

            // Increment the comparison count
            res.comparisons++;

            // Optionally print the comparison count every 100000
comparisons
```

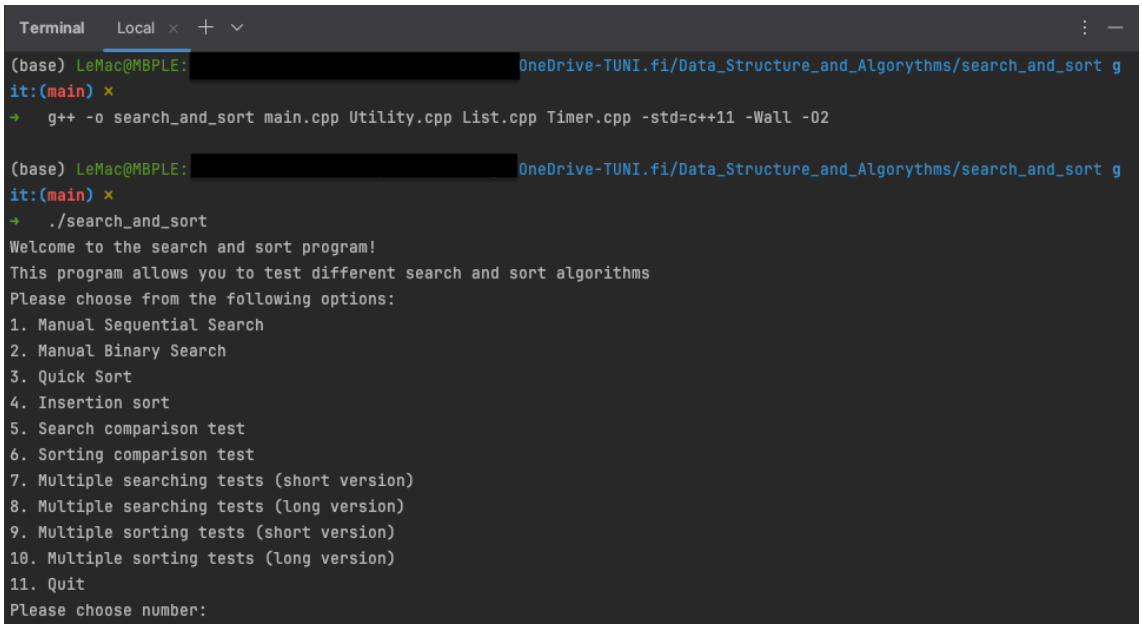
```
            if (res.comparisons % 100000 == 0 && print) {
                // Uncomment the following line to print the number of
comparisons
                // cout << "Comparisons done: " << res.comparisons <<
endl;
            }

            // If the current element is greater than or equal to the
element to sort, break the loop
            if (listToSort >= listToCompare) {
                break;
            }

            // Move the current element one position forward to make
space for the element to sort
            list.remove(j, listToCompare);
            list.insert(j + 1, listToCompare);

            // Decrement j to continue shifting elements
            j--;
        }

        // Insert the element to sort at its correct position
        list.remove(j + 1, listToCompare);  // Use a dummy variable
for the remove operation
        list.insert(j + 1, listToSort);
    }

    // Record the elapsed time for the sorting process
    res.time = timing.elapsed_time();

    // Return the sorting result containing the algorithm type,
comparison count, and elapsed time
    return res;
}
```

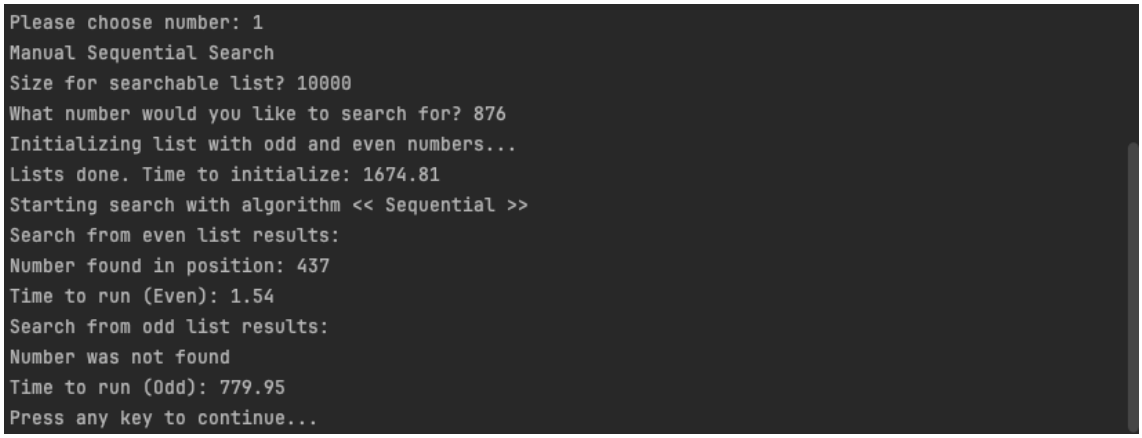
Quick sort algorithm

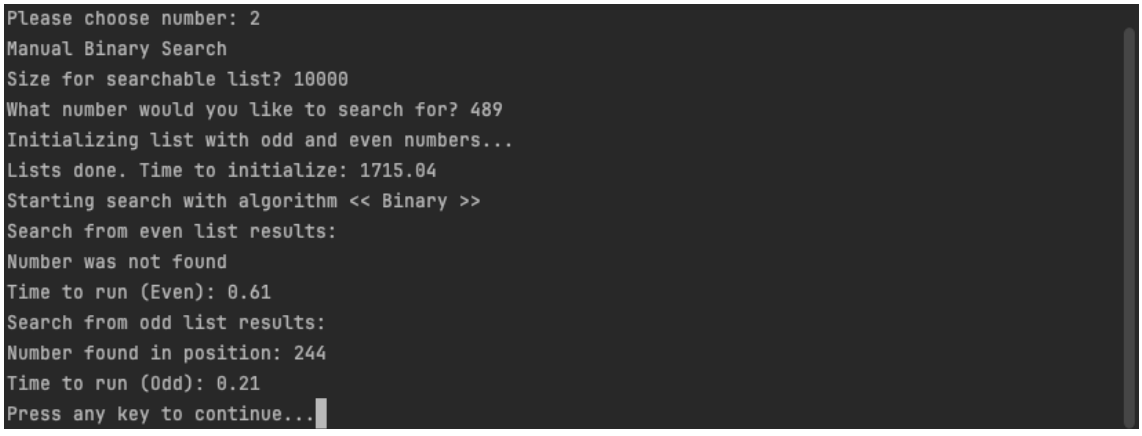
```
sortResult quickSort(List<unsigned int> &list, sortResult &result,
unsigned int startingPos, unsigned int endingPos) {
    // Base case: if the starting position is greater than or equal to
the ending position, return the result
    if (startingPos >= endingPos) {
        return result;
    }

    // Choose the starting element as the pivot
    unsigned int pivotIndex = startingPos;
    unsigned int pivot;
    list.retrieve(pivotIndex, pivot);  // Retrieve the pivot element

    // Initialize pointers for the partitioning process
    unsigned int i = startingPos + 1;
    unsigned int j = endingPos;

    // Partition the list around the pivot
    while (i <= j) {
        unsigned int listToSort, listToCompare;
        list.retrieve(i, listToSort);
        list.retrieve(j, listToCompare);
        result.comparisons++;

        // Move the pointers i and j towards each other
```

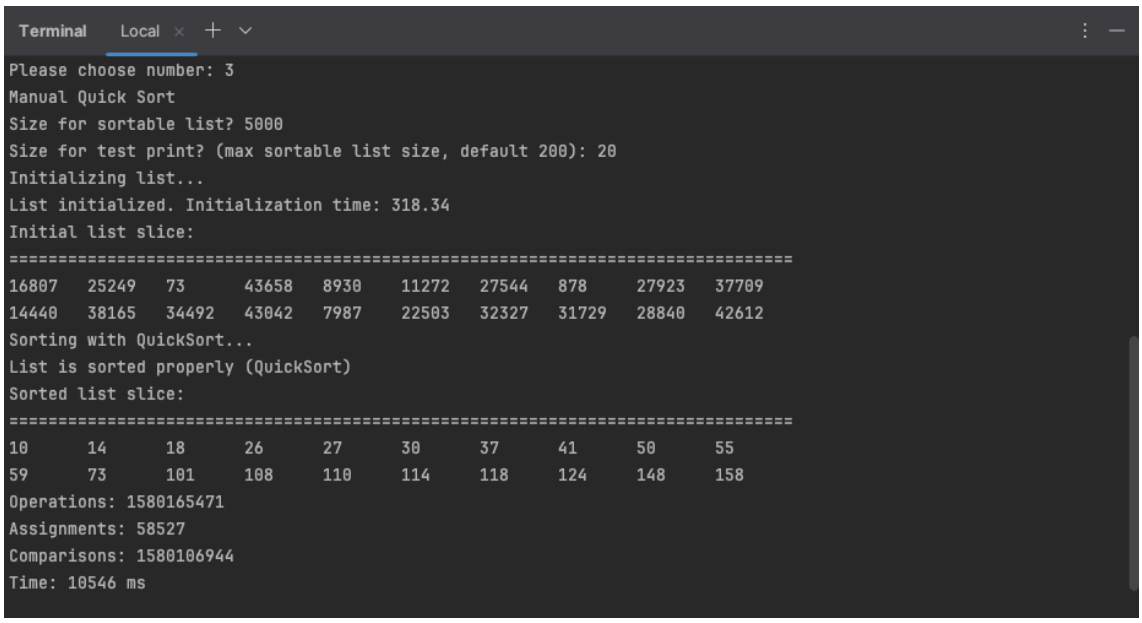
```
        if (listToSort <= pivot) {
            i++;
        } else if (listToCompare >= pivot) {
            j--;
        } else {
            // Swap elements at positions i and j
            list.remove(i, listToSort);
            list.insert(i, listToCompare);
            list.remove(j, listToCompare);
            list.insert(j, listToSort);
            i++;
            j--;
            result.assignments += 2;  // Increment the assignment
count
        }
    }

    // Move the pivot to its correct position
    list.remove(startingPos, pivot);
    list.insert(j, pivot);
    result.assignments++;  // Increment the assignment count

    // Recursively sort the left and right sublists
    if (j > startingPos) result = quickSort(list, result, startingPos,
j - 1);  // Sort the left sublist
    if (j < endingPos) result = quickSort(list, result, j + 1,
endingPos);  // Sort the right sublist

    // Return the sorting result containing the algorithm type,
comparison count, and elapsed time
    return result;
}
```

6 SCREENSHOTS

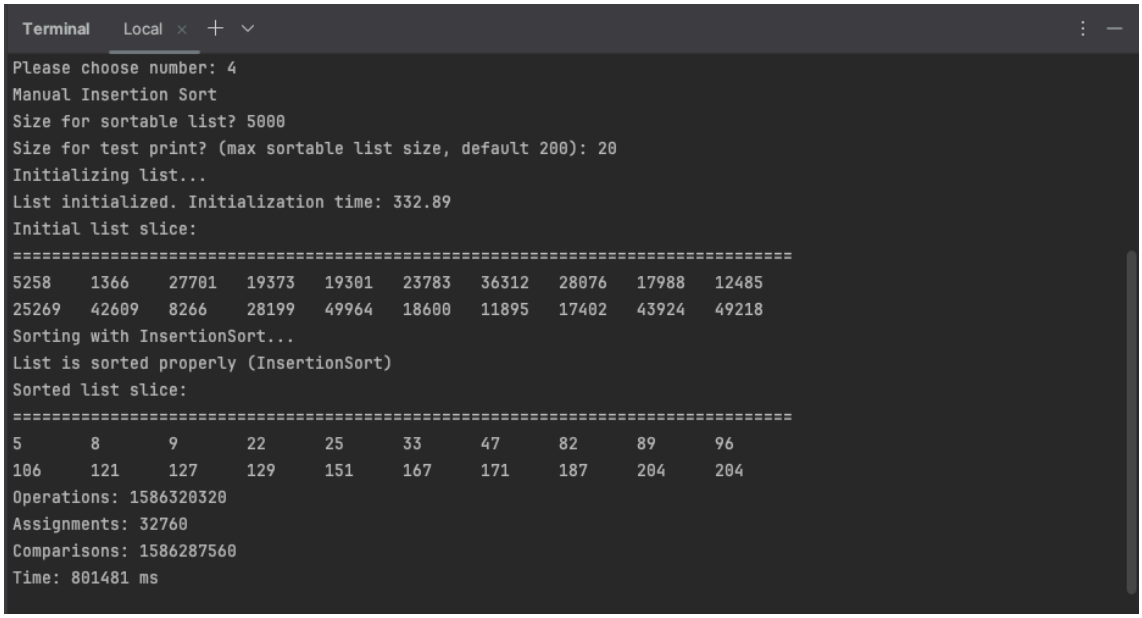








```
Terminal  Local
Please choose number: 3
Manual Quick Sort
Size for sortable list? 5000
Size for test print? (max sortable list size, default 200): 20
Initializing list...
List initialized. Initialization time: 318.34
Initial list slice:
================================================================================
16807   25249   73      43658   8930    11272   27544   878     27923   37709
14440   38165   34492   43042   7987    22503   32327   31729   28840   42612
Sorting with QuickSort...
List is sorted properly (QuickSort)
Sorted list slice:
================================================================================
10      14      18      26      27      30      37      41      50      55
59      73      101     108     110     114     118     124     148     158
Operations: 1580165471
Assignments: 58527
Comparisons: 1580106944
Time: 10546 ms
```



```
Terminal  Local
Please choose number: 4
Manual Insertion Sort
Size for sortable list? 5000
Size for test print? (max sortable list size, default 200): 20
Initializing list...
List initialized. Initialization time: 332.89
Initial list slice:
================================================================================
5258    1366    27701   19373   19301   23783   36312   28076   17988   12485
25269   42609   8266    28199   49964   18600   11895   17402   43924   49218
Sorting with InsertionSort...
List is sorted properly (InsertionSort)
Sorted list slice:
================================================================================
5       8       9       22      25      33      47      82      89      96
106     121     127     129     151     167     171     187     204     204
Operations: 1586320320
Assignments: 32760
Comparisons: 1586287560
Time: 801481 ms
```

```
Please choose number: 5
Search comparison test (manual)
Size of list: 50000
Initializing lists
Time to init lists for Even and Odd: 40102.8
How many runs / algorithm: 5
```

```
Search done
Searching setting:

**********************
Searches per algorithm: 5
Length of lists: 50000
Largest possible number in list: 100000
Total time: 120374

Sequential Search:
**********************
Status: Successful
Avg time per run: 4134.27
Avg comparisons per search: 18759
Searches: 5

Status: Unsuccessful
Avg time per run: 19931
Avg comparisons per search: 50000
Searches: 5

Binary Search:
**********************
Status: Successful
Avg time per run: 4.66
Avg comparisons per search: 14
Searches: 5

Status: Unsuccessful
Avg time per run: 4.872
Avg comparisons per search: 15
Searches: 5

Press any key to continue...
```

7 TROUBLESHOOTING

- **Compiler Errors**: Ensure that your compiler supports C++11 or later. Update your compiler if necessary.
- **CMake Errors**: Ensure that CMake is correctly installed and added to your system's PATH.
- **Library Dependencies**: Ensure all required libraries are available on your system. Install any missing dependencies as indicated by error messages.
  If you encounter any issues, refer to the documentation or seek help from the community or support channels associated with the repository.

Leo Suzuki

Class project May 2024

IT-engineering / SW

Project repository: https://github.com/LeoSuzu/Search_and_Sort_Cpp.git