# 1. Implementation

## 1.1. Implementation of QuickSort with Different Pivoting Strategies

Explain your code by providing:

- Implementation Details

  Implementing QuickSort algorithm in a programming language requires a lot of details. For the algorithm to be useful, it should support a wide range of data types. Therefore, generic programming is employed in this implementation. The first template argument *T* in Figure 1.1 is given by the user, the second and the third arguments are comparator classes that can be replaced to support comparison between any type. These comparators are set to defauld by using the classes in Figure 1.5 There are two static attributes to seed the random index generator throughout the program.

```
template <typename T, typename CmpLess = Less<T>, typename CmpEqu = Equal<T>>
class QuickSort
{
public:
    QuickSort();
    QuickSort(std::vector<T> &array, char inputStrategy = 'm');
    void quickSort(std::vector<T> &A, size_t begin, size_t end);

private:
    CmpLess less;
    CmpEqu equal;
    char strategy;
    static std::random_device rd;
    static std::mt19937 gen;
    size_t partition(std::vector<T> &A, size_t begin, size_t end);
    size_t randomIndex(size_t begin, size_t end) const;
};
```

**Figure 1.1:** Interface of QuickSort class

The actual sorting algorithm is represented in Figure 1.2. The *switch* block inside the method handles the pivot choice of the user. The parameter begin indicates the first index of the sub-array, and the end parameter is the size of the sub-array. Recursive calls are done at the bottom on the partitioned sub-arrays.

```
template <typename T, typename CmpLess, typename CmpEqu>
void QuickSort<T, CmpLess, CmpEqu>::quickSort(std::vector<T> &A, size_t begin, size_t end)
{
    if (begin >= end)
        return;
    switch (this->strategy)
    {
    case 'r':
    {
        size_t randomPivotIndex = this->randomIndex(begin, end);
        swapValues(A[randomPivotIndex], A[end - 1]);
        break;
    }
    case 'm':
    {
        size_t trinity[3] = {
            this->randomIndex(begin, end),
            this->randomIndex(begin, end),
            this->randomIndex(begin, end)};
        size_t median;
        if ((trinity[0] >= trinity[1]) && (trinity[1] >= trinity[2]))
            median = trinity[1];
        else if ((trinity[1] >= trinity[0]) && (trinity[0] >= trinity[2]))
            median = trinity[0];
        else
            median = trinity[2];
        swapValues(A[median], A[end - 1]);
        break;
    }
    default:
        break;
    }
    size_t pivotIndex = this->partition(A, begin, end);
    this->quickSort(A, begin, pivotIndex);
    this->quickSort(A, pivotIndex + 1, end);
}
```

**Figure 1.2:** Sorting algorithm

The *partition* method in Figure 1.3 makes use of comparators whenever it compares the elements of the input sub-array. The pivot is assumed to be at the end of the sub-array, so the pivot should be there before the call for this method.

```
template <typename T, typename CmpLess, typename CmpEqu>
size_t QuickSort<T, CmpLess, CmpEqu>::partition(std::vector<T> &A, size_t begin, size_t end)
{
    T pivot = A[end - 1];
    size_t i = begin - 1;
    for (size_t j = begin; j < end - 1; j++)
    {
        if (this->less(pivot, A[j]) || this->equal(pivot, A[j]))
            continue;
        i++;
        swapValues(A[i], A[j]);
    }
    swapValues(A[i + 1], A[end - 1]);
    return i + 1;
}
```

**Figure 1.3:** In-place partition method

Static attributes are initialized in the global scope. The *randomIndex* method in Figure 1.4 forms a uniform distribution between the given boundaries, then returns a randomly selected index.

```
template <typename T, typename CmpLess, typename CmpEqu>
std::random_device QuickSort<T, CmpLess, CmpEqu>::rd;

template <typename T, typename CmpLess, typename CmpEqu>
std::mt19937 QuickSort<T, CmpLess, CmpEqu>::gen = std::mt19937(rd());

template <typename T, typename CmpLess, typename CmpEqu>
size_t QuickSort<T, CmpLess, CmpEqu>::randomIndex(size_t begin, size_t end) const
{
    // Create a distribution for the array.
    std::uniform_int_distribution<size_t> distribution(begin, end - 1);
    return distribution(this->gen);
}
```

**Figure 1.4:** Random index generator

```cpp
template <typename T>
class Less
{
public:
    bool operator()(const T &obj1, const T &obj2) const
    {
        return obj1 < obj2;
    }
};

template <typename T>
class Equal
{
public:
    bool operator()(const T &obj1, const T &obj2) const
    {
        return obj1 == obj2;
    }
};
```

**Figure 1.5:** Default comparators

- Related Recurrence Relation

  $T(n) = T(p) + T(n - p - 1) + \Theta(n)$ where the p is the pivot index.

- Time and Space Complexity

  Time complexity of the last element strategy is $\Theta(n^2)$, and $\Theta(nlog(n))$ in average for the randomized strategies. The space complexity is $\Theta(1)$ for any strategy. Because there is no additional array is reserved in the memory.

Run your code with the configurations given in Table 1.1. Provide the execution time in nanoseconds (ns).

| | **Population1** | **Population2** | **Population3** | **Population4** |
|---|---|---|---|---|
| **Last Element** | 391.027.205 | 5.894.538.341 | 3.335.891.711 | 12.739.972 |
| **Random Element** | 13.137.718 | 10.586.840 | 10.993.082 | 12.502.034 |
| **Median of 3** | 12.162.844 | 11.544.581 | 12.115.785 | 13.210.216 |

**Table 1.1:** Comparison of different pivoting strategies on input data.

Discuss the outcome of your experiments.

Instead of relying on the last element, the pivot can be selected randomly to achieve faster time complexities.

## 1.2. Hybrid Implementation of QuickSort and InsertionSort

Explain your code by providing:

- Implementation Details

  The QuickSort algorithm is composed with the InsertionSort algorithm. In Figure 1.6, an additional attribute *k* is presented to hold the threshold value that is specified by the user.

```
template <typename T, typename CmpLess = Less<T>, typename CmpEqu = Equal<T>>
class QuickSort
{
public:
    QuickSort();
    QuickSort(std::vector<T> &array, char inputStrategy = 'm', size_t threshold = 1);
    void quickSort(std::vector<T> &A, size_t begin, size_t end);
    void insertionSort(std::vector<T> &A, size_t begin, size_t end) const;

private:
    CmpLess less;
    CmpEqu equal;
    size_t k;
    char strategy;
    static std::random_device rd;
    static std::mt19937 gen;
    size_t partition(std::vector<T> &A, size_t begin, size_t end);
    size_t randomIndex(size_t begin, size_t end) const;
};
```

**Figure 1.6:** Hybrid sort class interface

The insertionSort method in Figure 1.7 makes use of the comparators provided by the user (or default if the user does not provide any). Since the variable j is unsigned, it could not be checked for $j >= 0$ inside the *while* condition. Therefore, an additional *if* statement is introduced afterwards to complete the last iteration.

```
template <typename T, typename CmpLess, typename CmpEqu>
void QuickSort<T, CmpLess, CmpEqu>::insertionSort(std::vector<T> &A, size_t begin, size_t end) const
{
    size_t i, j;
    T key;
    for (i = begin + 1; i < end; i++)
    {
        key = A[i];
        j = i - 1;
        while (j > 0 && this->less(key, A[j]))
        {
            A[j + 1] = A[j];
            j--;
        }
        if (this->less(A[j], key) || this->equal(A[j], key))
            A[j + 1] = key;
        else
        {
            A[j + 1] = A[j];
            A[j] = key;
        }
    }
}
```

**Figure 1.7:** Insertion sort method

The bridge between the two sorting algorithms is constructed in the *quickSort* method in Figure 1.8 by providing an *if* statement.

```
template <typename T, typename CmpLess, typename CmpEqu, typename StringVal>
void QuickSort<T, CmpLess, CmpEqu, StringVal>::quickSort(std::vector<T> &A, size_t begin, size_t end)
{
    if (begin >= end)
        return;
    else if (end - begin <= this->k)
    {
        this->insertionSort(A, begin, end);
        return;
    }
```

**Figure 1.8:** Threshold check

- Related Recurrence Relation

  $T(n) = T(k) + T(p-k) + T(n-p-1) + \Theta(n+k)$ where the p is the pivot index, and k is the threshold value. It is assumed that p > k

- Time and Space Complexity

  Worst-case complexities for the hybrid sort algorithm is same as the ordinary QuickSort algorithm. However, since the InsertionSort algorithm has a best-case time complexity of $\Omega(n)$, the hybrid sort algorithm runs faster on average.

Run your code with different values of k. Record the execution time in nanoseconds (ns) on Table 1.2. Provide the execution time in nanoseconds (ns).

| Threshold (k) | Population4 |
|---|---|
| 1 | 12.736.435 |
| 5 | 11.717.028 |
| 10 | 10.938.999 |
| 20 | 10.793.503 |
| 40 | 13.177.674 |
| 60 | 13.000.527 |
| 100 | 15.220.446 |
| 150 | 18.110.829 |
| 200 | 20.257.048 |
| 250 | 23.433.264 |
| 300 | 28.204.071 |

**Table 1.2:** Comparison of different thresholds on the hybrid QuickSort algorithm on input data.

Discuss the outcome of your experiments.

Even if the InsertionSort algorithm has a worse worst-case time complexity than the average time complexity of the QuickSort algorithm, it operates faster on smaller or nearly sorted arrays. The results in the Table 1.2 proves that, by selecting a good threshold value, it is possible to get a faster solution.